

---

# **The VTK-m User's Guide**

***Release 2.1.0-241-g98168fc2***

**Kenneth Moreland**

**May 15, 2024**



# VTK-M USER'S GUIDE

<b>I</b>	<b>Getting Started</b>	<b>3</b>
1	Introduction	5
2	Building and Installing VTK-m	9
3	Quick Start	17
<b>II</b>	<b>Using VTK-m</b>	<b>23</b>
4	Base Types	25
5	VTK-m Version	33
6	Initialization	35
7	Data Sets	39
8	File I/O	77
9	Running Filters	83
10	Provided Filters	93
11	Rendering	163
12	Error Handling	203
13	Managing Devices	207
14	Timers	215
15	Implicit Functions	219
<b>III</b>	<b>Developing Algorithms</b>	<b>229</b>
16	General Approach	231
17	Basic Array Handles	235
18	Simple Worklets	247

<b>19 Basic Filter Implementation</b>	<b>253</b>
<b>IV Advanced Development</b>	<b>263</b>
20 Advanced Types	265
21 Logging	309
22 Worklet Types	317
23 Extended Filter Implementations	355
24 Worklet Error Handling	367
25 Math	369
26 Working with Cells	389
27 Memory Layout of Array Handles	407
<b>V Core Development</b>	<b>425</b>
<b>VI Appendix</b>	<b>427</b>
28 Acknowledgements	429
29 License	431
30 Index	433
Index	435

## The VTK-m User's Guide

Version 2.1.0-241-g98168fc2

Kenneth Moreland

with special contributions from Vicente Bolea, Hank Childs, Nickolas Davis, Mark Kim, James Kress, Matthew Letter, Li-Ta Lo, Robert Maynard, Sujin Philip, David Pugmire, Nick Thompson, Allison Vacanti, Abhishek Yenpure, and the VTK-m community

Moreland, K. (2023). *The VTK-m User's Guide*, Tech report ORNL/TM-2023/3182, Oak Ridge National Laboratory.

Join the VTK-m Community at <http://m.vtk.org>.



# **Part I**

## **Getting Started**





## **INTRODUCTION**

High-performance computing relies on ever finer threading. Advances in processor technology include ever greater numbers of cores, hyperthreading, accelerators with integrated blocks of cores, and special vectorized instructions, all of which require more software parallelism to achieve peak performance. Traditional visualization solutions cannot support this extreme level of concurrency. Extreme scale systems require a new programming model and a fundamental change in how we design algorithms. To address these issues we created VTK-m: the visualization toolkit for multi-/many-core architectures.

VTK-m supports a number of algorithms and the ability to design further algorithms through a top-down design with an emphasis on extreme parallelism. VTK-m also provides support for finding and building links across topologies, making it possible to perform operations that determine manifold surfaces, interpolate generated values, and find adjacencies. Although VTK-m provides a simplified high-level interface for programming, its template-based code removes the overhead of abstraction.

Table 1: Comparison of Marching Cubes implementations.

CUDA SDK 431 LOC	VTK-m 265 LOC
	

VTK-m simplifies the development of parallel scientific visualization algorithms by providing a framework of supporting functionality that allows developers to focus on visualization operations. Consider the listings in [Table 1](#) that compares the size of the implementation for the Marching Cubes algorithm in VTK-m with the equivalent reference implementation in the CUDA software development kit. Because VTK-m internally manages the parallel distribution of work and data, the VTK-m implementation is shorter and easier to maintain. Additionally, VTK-m provides data abstractions not provided by other libraries that make code written in VTK-m more versatile.

## 1.1 How to Use This Guide

This user's guide is organized into 5 parts to help guide novice to advanced users and to provide a convenient reference. [Part I \(Getting Started\)](#) provides a brief overview of using VTK-m. This part provides instructions on building VTK-m and some simple examples of using VTK-m. Users new to VTK-m are well served to read through [Part I \(Getting Started\)](#) first to become acquainted with the basic concepts.

The remaining parts, which provide detailed documentation of increasing complexity, have chapters that do not need to be read in detail. Readers will likely find it useful to skip to specific topics of interest.

Part II (Using VTK-m) dives deeper into the VTK-m library. It provides much more detail on the concepts introduced in Part I (Getting Started) and introduces new topics helpful to people who use VTK-m's existing algorithms.

Part III (Developing Algorithms) documents how to use VTK-m's framework to develop new or custom visualization algorithms. In this part we dive into the inner workings of filters and introduce the concept of a *worklet*, which is the base unit used to write a device-portable algorithm in VTK-m. Part III (Developing Algorithms) also documents many supporting functions that are helpful in implementing visualization algorithms.

Part IV (Advanced Development) explores in more detail how VTK-m manages memory and devices. This information describes how to adapt VTK-m to custom data structures and new devices.

Part V (Core Development) exposes the inner workings of VTK-m. These concepts allow you to design new algorithmic structures not already available in VTK-m.

---

### Did You Know?

In this guide we periodically use these **Did you know?** boxes to provide additional information related to the topic at hand.

---

---

### Common Errors

**Common Errors** blocks are used to highlight some of the common problems or complications you might encounter when dealing with the topic of discussion.

---



## BUILDING AND INSTALLING VTK-M

Before we begin describing how to develop with VTK-m, we have a brief overview of how to build VTK-m, optionally install it on your system, and start your own programs that use VTK-m.

### 2.1 Getting VTK-m

VTK-m is an open source software product where the code is made freely available. To get the latest released version of VTK-m, go to the VTK-m releases page:

<https://gitlab.kitware.com/vtk/vtk-m/-/releases>

From there with your favorite browser you may download the source code from any of the recent VTK-m releases in a variety of different archive files such as zip or tar gzip.

For access to the most recent work, the VTK-m development team provides public anonymous read access to their main source code repository. The main VTK-m repository on a GitLab instance hosted at Kitware, Inc. The repository can be browsed from its project web page:

<https://gitlab.kitware.com/vtk/vtk-m>

We leave access to the git hosted repository as an exercise for the user. Those interested in **git** access for the purpose of contributing to VTK-m should consult the **CONTRIBUTING** guidelines documented in the source code.

### 2.2 Configuring VTK-m

VTK-m uses a cross-platform configuration tool named CMake to simplify the configuration and building across many supported platforms. CMake is available from many package distribution systems and can also be downloaded for many platforms from <http://cmake.org>.

Most distributions of CMake come with a convenient GUI application (**cmake-gui**) that allows you to browse all of the available configuration variables and run the configuration. Many distributions also come with an alternative terminal-based version (**ccmake**), which is helpful when accessing remote systems where creating GUI windows is difficult.

One helpful feature of CMake is that it allows you to establish a build directory separate from the source directory, and the VTK-m project requires that separation. Thus, when you run CMake for the first time, you want to set the build directory to a new empty directory and the source to the downloaded or cloned files. The following example shows the steps for the case where the VTK-m source is cloned from the git repository. (If you extracted files from an archive downloaded from the VTK-m web page, the instructions are the same from the second line down.)

Example 1: Running CMake on downloaded VTK-m source (Unix commands).

```
tar xvzf ~/Downloads/vtk-m-v2.1.0.tar.gz
mkdir vtkm-build
cd vtkm-build
cmake-gui ../vtk-m-v2.1.0
```

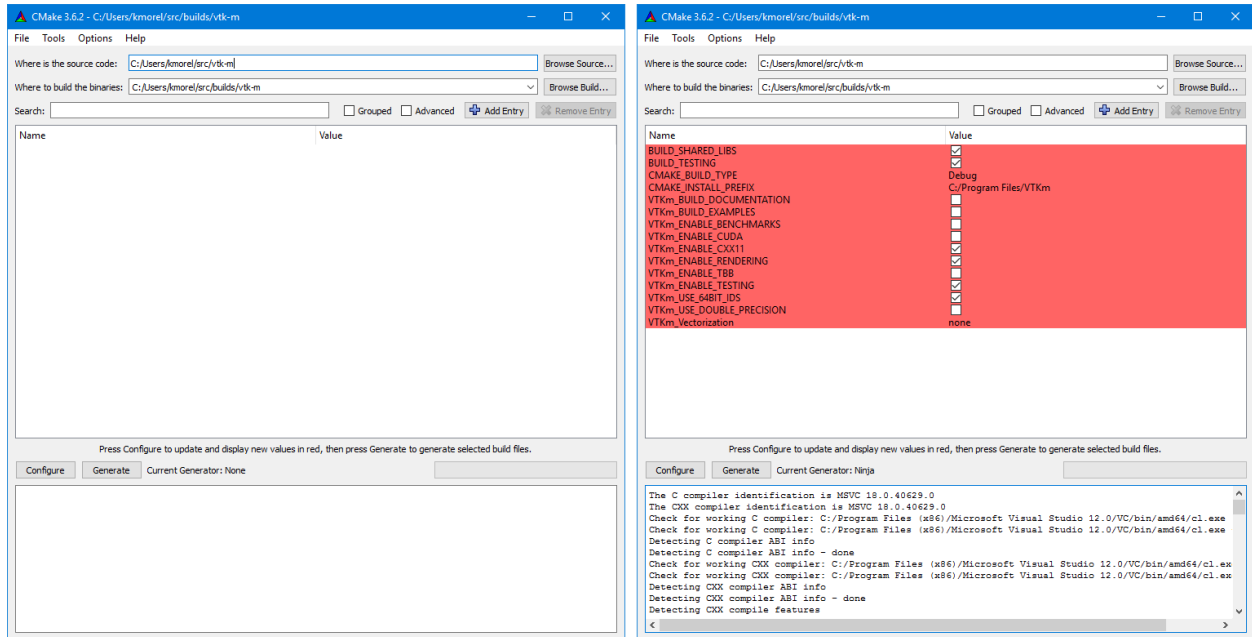


Figure 1: The CMake GUI configuring the VTK-m project. At left is the initial blank configuration. At right is the state after a configure pass.

The first time the CMake GUI runs, it initially comes up blank as shown at left in [Figure 1](#). Verify that the source and build directories are correct (located at the top of the GUI) and then click the *Configure* button near the bottom. The first time you run configure, CMake brings up a dialog box asking what generator you want for the project. This allows you to select what build system or IDE to use (e.g. make, ninja, Visual Studio). Once you click *Finish*, CMake will perform its first configuration. Don't worry if CMake gives an error about an error in this first configuration process.

## Common Errors

Most options in CMake can be reconfigured at any time, but not the compiler and build system used. These must be set the first time configure is run and cannot be subsequently changed. If you want to change the compiler or the project file types, you will need to delete everything in the build directory and start over.

After the first configuration, the CMake GUI will provide several configuration options as shown in [Figure 1](#) on the right. You now have a chance to modify the configuration of VTK-m, which allows you to modify both the behavior of the compiled VTK-m code as well as find components on your system. Using the CMake GUI is usually an iterative process where you set configuration options and re-run *Configure*. Each time you configure, CMake might find new options, which are shown in red in the GUI.

It is often the case during this iterative configuration process that configuration errors occur. This can occur after a new option is enabled but CMake does not automatically find the necessary libraries to make that feature possible. For example, to enable TBB support, you may have to first enable building TBB, configure for TBB support, and then tell

CMake where the TBB include directories and libraries are.

Once you have set all desired configuration variables and resolved any CMake errors, click the *Generate* button. This will create the build files (such as makefiles or project files depending on the generator chosen at the beginning). You can then close the CMake GUI.

There are a great number of configuration parameters available when running CMake on VTK-m. The following list contains the most common configuration parameters.

**BUILD\_SHARED\_LIBS**

Determines whether static or shared libraries are built.

**CMAKE\_BUILD\_TYPE**

Selects groups of compiler options from categories like Debug and Release. Debug builds are, obviously, easier to debug, but they run *much* slower than Release builds. Use Release builds whenever releasing production software or doing performance tests.

**CMAKE\_INSTALL\_PREFIX**

The root directory to place files when building the install target.

**VTKm\_ENABLE\_EXAMPLES**

The VTK-m repository comes with an `textfilename{examples}` directory. This macro determines whether they are built.

**VTKm\_ENABLE\_BENCHMARKS**

If on, the VTK-m build includes several benchmark programs. The benchmarks are regression tests for performance.

**VTKm\_ENABLE\_CUDA**

Determines whether VTK-m is built to run on CUDA GPU devices.

**VTKm\_ENABLE\_KOKKOS**

Determines whether VTK-m is built using the [Kokkos](#) portable library. Kokkos, can be configured to support several backends that VTK-m can leverage.

**VTKm\_ENABLE\_MPI**

Determines whether VTK-m is built with MPI support for running on distributed memory clusters.

**VTKm\_ENABLE\_OPENMP**

Determines whether VTK-m is built to run on multi-core devices using OpenMP pragmas provided by the C++ compiler.

**VTKm\_ENABLE\_RENDERING**

Determines whether to build the rendering library.

**VTKm\_ENABLE\_TBB**

Determines whether VTK-m is built to run on multi-core x86 devices using the Intel Threading Building Blocks library.

**VTKm\_ENABLE\_TESTING**

If on, the VTK-m build includes building many test programs. The VTK-m source includes hundreds of regression tests to ensure quality during development.

**VTKm\_ENABLE\_TUTORIALS**

If on, several small example programmes used for the VTK-m tutorial are built.

**VTKm\_USE\_64BIT\_IDS**

If on, then VTK-m will be compiled to use 64-bit integers to index arrays and other lists. If off, then VTK-m will use 32-bit integers. 32-bit integers take less memory but could cause failures on larger data.

### VTKm\_USE\_DOUBLE\_PRECISION

If on, then VTK-m will use double precision (64-bit) floating point numbers for calculations where the precision type is not otherwise specified. If off, then single precision (32-bit) floating point numbers are used. Regardless of this setting, VTK-m's templates will accept either type.

## 2.3 Building VTK-m

Once CMake successfully configures VTK-m and generates the files for the build system, you are ready to build VTK-m. As stated earlier, CMake supports generating configuration files for several different types of build tools. Make and ninja are common build tools, but CMake also supports building project files for several different types of integrated development environments such as Microsoft Visual Studio and Apple XCode.

The VTK-m libraries and test files are compiled when the default build is invoked. For example, if a `Makefile` was generated, the build is invoked by calling `textfilename{make}` in the build directory. Expanding on [Example 1](#)

Example 2: Using **make** to build VTK-m.

```
tar xvfz ~/Downloads/vtk-m-v2.1.0.tar.gz
mkdir vtkm-build
cd vtkm-build
cmake-gui ../vtk-m-v2.1.0
make -j
make install
```

---

### Did You Know?

`Makefile` and other project files generated by CMake support parallel builds, which run multiple compile steps simultaneously. On computers that have multiple processing cores (as do almost all modern computers), this can significantly speed up the overall compile. Some build systems require a special flag to engage parallel compiles. For example, **make** requires the `-j` flag to start parallel builds as demonstrated in [Example 2](#).

---

### Did You Know?

[Example 2](#) assumes that a make build system was generated, which is the default on most system. However, CMake supports many more build systems, which use different commands to run the build. If you are not sure what the appropriate build command is, you can run `cmake --build` to allow CMake to start the build using whatever build system is being used.

---

### Common Errors

CMake allows you to switch between several types of builds including default, Debug, and Release. Programs and libraries compiled as release builds can run *much* faster than those from other types of builds. Thus, it is important to perform Release builds of all software released for production or where runtime is a concern. Some integrated development environments such as Microsoft Visual Studio allow you to specify the different build types within the build system. But for other build programs, like **make**, you have to specify the build type in the `CMAKE_BUILD_TYPE` CMake configuration variable, which is described in [Section 2.2 \(Configuring VTK-m\)](#).

CMake creates several build “targets” that specify the group of things to build. The default target builds all of VTK-m's libraries as well as tests, examples, and benchmarks if enabled. The `test` target executes each of the VTK-m regression tests and verifies they complete successfully on the system. The `install` target copies the subset of files required to



use VTK-m to a common installation directory. The `install` target may need to be run as an administrator user if the installation directory is a system directory.

---

### Did You Know?

VTK-m contains a significant amount of regression tests. If you are not concerned with testing a build on a given system, you can turn off building the testing, benchmarks, and examples using the CMake configuration variables described in [Section 2.2 \(Configuring VTK-m\)](#). This can shorten the VTK-m compile time.

---

## 2.4 Linking to VTK-m

Ultimately, the value of VTK-m is the ability to link it into external projects that you write. The header files and libraries installed with VTK-m are typical, and thus you can link VTK-m into a software project using any type of build system. However, VTK-m comes with several CMake configuration files that simplify linking VTK-m into another project that is also managed by CMake. Thus, the documentation in this section is specifically for finding and configuring VTK-m for CMake projects.

VTK-m can be configured from an external project using the `find_package()` CMake function. The behavior and use of this function is well described in the CMake documentation. The first argument to `find_package()` is the name of the package, which in this case is `VTKm`. CMake configures this package by looking for a file named `VTKmConfig.cmake`, which will be located in the `lib/cmake/vtkm-<VTKm version>` directory of the install or build of VTK-m. The configurable CMake variable `CMAKE_PREFIX_PATH` can be set to the build or install directory, the `CMAKE_PREFIX_PATH` environment variable can likewise be set, or `cmakevar{VTKm_DIR}` can be set to the directory that contains this file.

Example 3: Loading VTK-m configuration from an external CMake project.

```
find_package(VTKm REQUIRED)
```

---

### Did You Know?

The CMake `find_package()` function also supports several features not discussed here including specifying a minimum or exact version of VTK-m and turning off some of the status messages. See the CMake documentation for more details.

---

When you load the VTK-m package in CMake, several libraries are defined. Projects building with VTK-m components should link against one or more of these libraries as appropriate, typically with the `target_link_libraries()` command.

Example 4: Linking VTK-m code into an external program.

```
find_package(VTKm REQUIRED)

add_executable(myprog myprog.cxx)
target_link_libraries(myprog vtkm::filter)
```

Several library targets are provided, but most projects will need to link in one or more of the following.

#### **vtkm::cont**

Contains the base objects used to control VTK-m.

### **vtkm::filter**

Contains VTK-m's pre-built filters. Applications that are looking to use VTK-m filters will need to link to this library. The filters are further broken up into several smaller library packages (such as `vtkm::filter_contour`, `cmake:variable`vtkm::filter_flow``, `vtkm::filter_field_transform`, and many more. `vtkm::filter` is actually a meta library that links all of these filter libraries to a CMake target.

### **vtkm::io**

Contains VTK-m's facilities for interacting with files. For example, reading and writing png, NetBPM, and VTK files.

### **vtkm::rendering**

Contains VTK-m's rendering components. This library is only available if `VTKm_ENABLE_RENDERING` is set to true.

### **vtkm::source**

Contains VTK-m's pre-built dataset generators such as Wavelet, Tangle, and Oscillator. Most applications will not need to link to this library.

---

## **Did You Know?**

The “libraries” made available in the VTK-m do more than add a library to the linker line. These libraries are actually defined as external targets that establish several compiler flags, like include file directories. Many CMake packages require you to set up other target options to compile correctly, but for VTK-m it is sufficient to simply link against the library.

---

---

## **Common Errors**

Because the VTK-m CMake libraries do more than set the link line, correcting the link libraries can do more than fix link problems. For example, if you are getting compile errors about not finding VTK-m header files, then you probably need to link to one of VTK-m's libraries to fix the problem rather than try to add the include directories yourself.

---

The following is a list of all the CMake variables defined when the `textcode{find_package}` function completes.

### **VTKm\_FOUND**

Set to true if the VTK-m CMake package is successfully loaded. If `find_package()` was not called with the `REQUIRED` option, then this variable should be checked before attempting to use VTK-m.

### **VTKm\_VERSION**

The version number of the loaded VTK-m package. This is in the form “major.minor”.

### **VTKm\_VERSION\_FULL**

The extended version number of the VTK-m package including patch and in-between-release information. This is in the form “major.minor.patch[.gitsha1]” where “gitsha” is only included if the source code is in between releases.

### **VTKm\_VERSION\_MAJOR**

The major VTK-m version number.

### **VTKm\_VERSION\_MINOR**

The minor VTK-m version number.

### **VTKm\_VERSION\_PATCH**

The patch VTK-m version number.

**VTkm\_ENABLE\_CUDA**

Set to true if VTK-m was compiled for CUDA.

**VTkm\_ENABLE\_Kokkos**

Set to true if VTK-m was compiled with Kokkos.

**VTkm\_ENABLE\_OPENMP**

Set to true if VTK-m was compiled for OpenMP.

**VTkm\_ENABLE\_TBB**

Set to true if VTK-m was compiled for TBB.

**VTkm\_ENABLE\_RENDERING**

Set to true if the VTK-m rendering library was compiled.

**VTkm\_ENABLE\_MPI**

Set to true if VTK-m was compiled with MPI support.

These package variables can be used to query whether optional components are supported before they are used in your CMake configuration.

Example 5: Using an optional component of VTK-m.

```
find_package(VTKm REQUIRED)

if (NOT VTKm::ENABLE::RENDERING)
  message(FATAL_ERROR "VTK-m must be built with rendering on.")
endif()

add_executable(myprog myprog.cxx)
target_link_libraries(myprog vtkm::cont vtkm::rendering)
```



## QUICK START

In this chapter we go through the steps to create a simple program that uses VTK-m. This “hello world” example presents only the bare minimum of features available. The remainder of this book documents dives into much greater detail.

We will call the example program we are building `VTKmQuickStart`. It will demonstrate reading data from a file, processing the data with a filter, and rendering an image of the data. Readers who are less interested in an explanation and are more interested in browsing some code can skip to [Section 3.5 \(The Full Example\)](#).

### 3.1 Initialize

The first step to using VTK-m is to initialize the library. Although initializing VTK-m is *optional*, it is recommend to allow VTK-m to configure devices and logging. Initialization is done by calling the `vtkm::cont::Initialize()` function. The `Initialize` function is defined in the `vtkm/cont/Initialize.h` header file.

`Initialize` takes the `argc` and `argv` arguments that are passed to the `main` function of your program, find any command line arguments relevant to VTK-m, and remove them from the list to make further command line argument processing easier.

Example 1: Initializing VTK-m.

```
1 int main(int argc, char* argv[])
2 {
3     vtkm::cont::Initialize(argc, argv, vtkm::cont::InitializeOptions::AddHelp);
```

Initialize has many options to customize command line argument processing. See [Chapter 6 \(Initialization\)](#) for more details.

---

**Did You Know?**

Don't have access to argc and argv? No problem. You can call `vtkm::cont::Initialize()` with no arguments.

---

## 3.2 Reading a File

VTK-m comes with a simple I/O library that can read and write files in VTK legacy format. These files have a `.vtk` extension.

VTK legacy files can be read using the `vtkm::io::VTKDataSetReader` object, which is declared in the `vtkm/io/VTKDataSetReader.h` header file. The object is constructed with a string specifying the filename (which for this example we will get from the command line). The data is then read in by calling the `vtkm::io::VTKDataSetReader::ReadDataSet()` method.

Example 2: Reading data from a VTK legacy file.

```
1 vtkm::io::VTKDataSetReader reader(argv[1]);
2 vtkm::cont::DataSet inData = reader.ReadDataSet();
```

The `ReadDataSet` method returns the data in a `vtkm::cont::DataSet` object. The structure and features of a `DataSet` object is described in [Chapter 7 \(Data Sets\)](#). For the purposes of this quick start, we will treat `DataSet` as a mostly opaque object that gets passed to and from operations in VTK-m.

More information about VTK-m's file readers and writers can be found in [Chapter 8 \(File I/O\)](#).

## 3.3 Running a Filter

Algorithms in VTK-m are encapsulated in units called *filters*. A filter takes in a `DataSet`, processes it, and returns a new `DataSet`. The returned `DataSet` often, but not always, contains data inherited from the source data.

VTK-m comes with many filters, which are documented in [Chapter 10 \(Provided Filters\)](#). For this example, we will demonstrate the use of the `vtkm::filter::MeshQuality` filter, which is defined in the `vtkm/filter/MeshQuality.h` header file. The `MeshQuality` filter will compute for each cell in the input data will compute a quantity representing some metric of the cell's shape. Several metrics are available, and in this example we will find the area of each cell.

Like all filters, `MeshQuality` contains an `Execute` method that takes an input `DataSet` and produces an output `DataSet`. It also has several methods used to set up the parameters of the execution. [Section 10.10.3 \(Mesh Quality Metrics\)](#) provides details on all the options of `MeshQuality`. Suffice it to say that in this example we instruct the filter to find the area of each cell, which it will output to a field named `area`.

Example 3: Running a filter.

```

1 vtkm::filter::mesh_info::MeshQuality cellArea;
2 cellArea.SetMetric(vtkm::filter::mesh_info::CellMetric::Area);
3 vtkm::cont::DataSet outData = cellArea.Execute(inData);

```

## 3.4 Rendering an Image

Although it is possible to leverage external rendering systems, VTK-m comes with its own self-contained image rendering algorithms. These rendering classes are completely implemented with the parallel features provided by VTK-m, so using rendering in VTK-m does not require any complex library dependencies.

Even a simple rendering scene requires setting up several parameters to establish what is to be featured in the image including what data should be rendered, how that data should be represented, where objects should be placed in space, and the qualities of the image to generate. Consequently, setting up rendering in VTK-m involves many steps. [Chapter 11 \(Rendering\)](#) goes into much detail on the ways in which a rendering scene is specified. For now, we just briefly present some boilerplate to achieve a simple rendering.

Example 4: Rendering data.

```

1 vtkm::rendering::Actor actor(
2     outData.GetCellSet(), outData.GetCoordinateSystem(), outData.GetField("area"));
3
4 vtkm::rendering::Scene scene;
5 scene.AddActor(actor);
6
7 vtkm::rendering::MapperRayTracer mapper;
8
9 vtkm::rendering::CanvasRayTracer canvas(1280, 1024);
10
11 vtkm::rendering::View3D view(scene, mapper, canvas);
12
13 view.Paint();
14
15 view.SaveAs("image.png");

```

The first step in setting up a render is to create a *scene*. A scene comprises some number of *actors*, which represent some data to be rendered in some location in space. In our case we only have one *DataSet* to render, so we simply create a single actor and add it to a scene as shown in [Example 4 lines 1 – 5](#).

The second step in setting up a render is to create a *view*. The view comprises the aforementioned scene, a *mapper*, which describes how the data are to be rendered, and a *canvas*, which holds the image buffer and other rendering context. The view is created in [Example 4, line 11](#). The image generation is then performed by calling `vtkm::rendering::View::Paint()` on the view object ([Example 4, line 13](#)). However, the rendering done by VTK-m's rendering classes is performed offscreen, which means that the result does not appear on your computer's monitor. The easiest way to see the image is to save it to an image file using the `vtkm::rendering::View::SaveAs()` method ([Example 4, line 15](#)).

## 3.5 The Full Example

Putting together the examples from the previous sections, here is a complete program for reading, processing, and rendering data with VTK-m.

Example 5: Simple example of using VTK-m.

```

1  #include <vtkm/cont/Initialize.h>
2
3  #include <vtkm/io/VTKDataSetReader.h>
4
5  #include <vtkm/filter/mesh_info/MeshQuality.h>
6
7  #include <vtkm/rendering/Actor.h>
8  #include <vtkm/rendering/CanvasRayTracer.h>
9  #include <vtkm/rendering/MapperRayTracer.h>
10 #include <vtkm/rendering/Scene.h>
11 #include <vtkm/rendering/View3D.h>
12
13 int main(int argc, char* argv[])
14 {
15     vtkm::cont::Initialize(argc, argv, vtkm::cont::InitializeOptions::AddHelp);
16
17     if (argc != 2)
18     {
19         std::cerr << "USAGE: " << argv[0] << " <file.vtk>" << std::endl;
20         return 1;
21     }
22
23     // Read in a file specified in the first command line argument.
24     vtkm::io::VTKDataSetReader reader(argv[1]);
25     vtkm::cont::DataSet inData = reader.ReadDataSet();
26
27     // Run the data through the elevation filter.
28     vtkm::filter::mesh_info::MeshQuality cellArea;
29     cellArea.SetMetric(vtkm::filter::mesh_info::CellMetric::Area);
30     vtkm::cont::DataSet outData = cellArea.Execute(inData);
31
32     // Render an image and write it out to a file.
33     vtkm::rendering::Actor actor(
34         outData.GetCellSet(), outData.GetCoordinateSystem(), outData.GetField("area"));
35
36     vtkm::rendering::Scene scene;
37     scene.AddActor(actor);
38
39     vtkm::rendering::MapperRayTracer mapper;
40
41     vtkm::rendering::CanvasRayTracer canvas(1280, 1024);
42
43     vtkm::rendering::View3D view(scene, mapper, canvas);
44
45     view.Paint();
46

```

(continues on next page)



(continued from previous page)

```
47 view.SaveAs("image.png");
48
49 return 0;
50 }
```

## 3.6 Build Configuration

Now that we have the program listed in [Example 5](#), we still need to compile it with the appropriate compilers and flags. By far the easiest way to compile VTK-m code is to use CMake. CMake commands that can be used to link code to VTK-m are discussed in [Section 2.4 \(Linking to VTK-m\)](#). The following example provides a minimal `CMakeLists.txt` required to build this program.

Example 6: `CMakeLists.txt` to build a program using VTK-m.

```
1 cmake_minimum_required(VERSION 3.13)
2 project(VTKmQuickStart CXX)
3
4 find_package(VTKm REQUIRED)
5
6 add_executable(VTKmQuickStart VTKmQuickStart.cxx)
7 target_link_libraries(VTKmQuickStart vtkm::filter vtkm::rendering)
```

The first two lines contain boilerplate for any `CMakeLists.txt` file. They all should declare the minimum CMake version required (for backward compatibility) and have a `project()` command to declare which languages are used.

The remainder of the commands find the VTK-m library, declare the program begin compiled, and link the program to the VTK-m library. These steps are described in detail in [Section 2.4 \(Linking to VTK-m\)](#).



# **Part II**

## **Using VTK-m**



## BASE TYPES

It is common for a framework to define its own types. Even the C++ standard template library defines its own base types like `std::size_t` and `std::pair`. VTK-m is no exception.

In fact VTK-m provides a great many base types. It is the general coding standard of VTK-m to not directly use the base C types like `int` and `float` and instead to use types declared in VTK-m. The rational is to precisely declare the representation of each variable to prevent future trouble.

Consider that you are programming something and you need to declare an integer variable. You would declare this variable as `int`, right? Well, maybe. In C++, the declaration `int` does not simply mean “an integer.” `int` means something much more specific than that. If you were to look up the C++11 standard, you would find that `int` is an integer represented in 32 bits with a two’s complement signed representation. In fact, a C++ compiler has no less than 8 standard integer types.

So, `int` is nowhere near as general as the code might make it seem, and treating it as such could lead to trouble. For example, consider the MPI standard, which, back in the 1990’s, implicitly selected `int` for its indexing needs. Fast forward to today where there is a need to reference buffers with more than 2 billion elements, but the standard is stuck with a data type that cannot represent sizes that big. (To be fair, it is *possible* to represent buffers this large in MPI, but it is extraordinarily awkward to do so.

Consequently, we feel that with VTK-m it is best to declare the intention of a variable with its declaration, which should help both prevent errors and future proof code. All the types presented in this chapter are declared in `vtkm/Types.h`, which is typically included either directly or indirectly by all source using VTK-m.

### 4.1 Floating Point Types

VTK-m declares 2 types to hold floating point numbers: `vtkm::Float32` and `vtkm::Float64`. These, of course, represent floating point numbers with 32-bits and 64-bits of precision, respectively. These should be used when the precision of a floating point number is predetermined.

```
using vtkm::Float32 = float
```

Base type to use for 32-bit floating-point numbers.

```
using vtkm::Float64 = double
```

Base type to use for 64-bit floating-point numbers.

When the precision of a floating point number is not predetermined, operations usually have to be overloaded or templated to work with multiple precisions. In cases where a precision must be set, but no particular precision is specified, `vtkm::FloatDefault` should be used.

```
using vtkm::FloatDefault = vtkm::Float32
```

The floating point type to use when no other precision is specified.

`vtkm::FloatDefault` will be set to either `vtkm::Float32` or `vtkm::Float64` depending on whether the CMake option `VTKm_USE_DOUBLE_PRECISION` was set when VTK-m was compiled, as discussed in [Section 2.2 \(Configuring VTK-m\)](#). Using `vtkm::FloatDefault` makes it easier for users to trade off precision and speed.

## 4.2 Integer Types

The most common use of an integer in VTK-m is to index arrays. For this purpose, the `vtkm::Id` type should be used. (The width of `vtkm::Id` is determined by the `VTKm_USE_64BIT_IDS` CMake option.)

```
using vtkm::Id = vtkm::Int64
```

Base type to use to index arrays.

This type represents an ID (index into arrays). It should be used whenever indexing data that could grow arbitrarily large.

VTK-m also has a secondary index type named `vtkm::IdComponent`, which is smaller and typically used for indexing groups of components within a thread. For example, if you had an array of 3D points, you would use `vtkm::Id` to reference each point, and you would use `vtkm::IdComponent` to reference the respective *x*, *y*, and *z* components.

```
using vtkm::IdComponent = vtkm::Int32
```

Base type to use to index small lists.

This type represents a component ID (index of component in a vector). The number of components, being a value fixed at compile time, is generally assumed to be quite small. However, we are currently using a 32-bit width integer because modern processors tend to access them more efficiently than smaller widths.

---

### Did You Know?

The VTK-m index types, `vtkm::Id` and `vtkm::IdComponent` use signed integers. This breaks with the convention of other common index types like the C++ standard template library `std::size_t`, which use unsigned integers. Unsigned integers make sense for indices as a valid index is always 0 or greater. However, doing things like iterating in a for loop backward, representing relative indices, and representing invalid values is much easier with signed integers. Thus, VTK-m chooses to use a signed integer for indexing.

---

VTK-m also has types to declare an integer of a specific width and sign. The types `vtkm::Int8`, `vtkm::Int16`, `vtkm::Int32`, and `vtkm::Int64` specify signed integers of 1, 2, 4, and 8 bytes, respectively. Likewise, the types `vtkm::UInt8`, `vtkm::UInt16`, `vtkm::UInt32`, and `vtkm::UInt64` specify unsigned integers of 1, 2, 4, and 8 bytes, respectively.

```
using vtkm::Int8 = int8_t
```

Base type to use for 8-bit signed integer numbers.

```
using vtkm::UInt8 = uint8_t
```

Base type to use for 8-bit unsigned integer numbers.

```
using vtkm::Int16 = int16_t
```

Base type to use for 16-bit signed integer numbers.

```
using vtkm::UInt16 = uint16_t
```

Base type to use for 16-bit unsigned integer numbers.

```
using vtkm::Int32 = int32_t
```

Base type to use for 32-bit signed integer numbers.

```
using vtkm::UInt32 = uint32_t
```

Base type to use for 32-bit unsigned integer numbers.

```
using vtkm::Int64 = signed long long
```

Base type to use for 64-bit signed integer numbers.

```
using vtkm::UInt64 = unsigned long long
```

Base type to use for 64-bit signed integer numbers.

## 4.3 Vector Types

Visualization algorithms also often require operations on short vectors. Arrays indexed in up to three dimensions are common. Data are often defined in 2-space and 3-space, and transformations are typically done in homogeneous coordinates of length 4. To simplify these types of operations, VTK-m provides a collection of base types to represent these short vectors, which are collectively referred to as Vec types.

`vtkm::Vec2f`, `vtkm::Vec3f`, and `vtkm::Vec4f` specify floating point vectors of 2, 3, and 4 components, respectively. The precision of the floating point numbers follows that of `vtkm::FloatDefault` (which, as documented in Section 4.1 (Floating Point Types), is specified by the `VTKm_USE_DOUBLE_PRECISION` compile option). Components of these and other Vec types can be references through the `[ ]` operator, much like a C array. A Vec also supports basic arithmetic operators so that it can be used much like its scalar-value counterparts.

```
using vtkm::Vec2f = vtkm::Vec<vtkm::FloatDefault, 2>
```

Vec2f corresponds to a 2-dimensional vector of floating point values.

Each floating point value is of the default precision (i.e. `vtkm::FloatDefault`). It is typedef for `vtkm::Vec<vtkm::FloatDefault, 2>`.

```
using vtkm::Vec3f = vtkm::Vec<vtkm::FloatDefault, 3>
```

Vec3f corresponds to a 3-dimensional vector of floating point values.

Each floating point value is of the default precision (i.e. `vtkm::FloatDefault`). It is typedef for `vtkm::Vec<vtkm::FloatDefault, 3>`.

```
using vtkm::Vec4f = vtkm::Vec<vtkm::FloatDefault, 4>
```

Vec4f corresponds to a 4-dimensional vector of floating point values.

Each floating point value is of the default precision (i.e. `vtkm::FloatDefault`). It is typedef for `vtkm::Vec<vtkm::FloatDefault, 4>`.

Example 1: Simple use of Vec objects.

```
1  vtkm::Vec2f A(1);           // A is (1, 1)
2  A[1] = 3;                  // A is (1, 3) now
3  vtkm::Vec2f B = { 4, 5 };  // B is (4, 5)
4  vtkm::Vec2f C = A + B;     // C is (5, 8)
5  vtkm::FloatDefault manhattanDistance = C[0] + C[1];
```

You can also specify the precision for each of these vector types by appending the bit size of each component. For example, `vtkm::Vec3f_32` and `vtkm::Vec3f_64` represent 3-component floating point vectors with each component being 32 bits and 64 bits respectively. Note that the precision number refers to the precision of each component, not the vector as a whole. So `vtkm::Vec3f_32` contains 3 32-bit (4-byte) floating point components, which means the entire `vtkm::Vec3f_32` requires 96 bits (12 bytes).

using `vtkm::Vec2f_32` = `vtkm::Vec<vtkm::Float32, 2>`

`Vec2f_32` corresponds to a 2-dimensional vector of 32-bit floating point values.

It is typedef for `vtkm::Vec<vtkm::Float32, 2>`.

using `vtkm::Vec2f_64` = `vtkm::Vec<vtkm::Float64, 2>`

`Vec2f_64` corresponds to a 2-dimensional vector of 64-bit floating point values.

It is typedef for `vtkm::Vec<vtkm::Float64, 2>`.

using `vtkm::Vec3f_32` = `vtkm::Vec<vtkm::Float32, 3>`

`Vec3f_32` corresponds to a 3-dimensional vector of 32-bit floating point values.

It is typedef for `vtkm::Vec<vtkm::Float32, 3>`.

using `vtkm::Vec3f_64` = `vtkm::Vec<vtkm::Float64, 3>`

`Vec3f_64` corresponds to a 3-dimensional vector of 64-bit floating point values.

It is typedef for `vtkm::Vec<vtkm::Float64, 3>`.

using `vtkm::Vec4f_32` = `vtkm::Vec<vtkm::Float32, 4>`

`Vec4f_32` corresponds to a 4-dimensional vector of 32-bit floating point values.

It is typedef for `vtkm::Vec<vtkm::Float32, 4>`.

using `vtkm::Vec4f_64` = `vtkm::Vec<vtkm::Float64, 4>`

`Vec4f_64` corresponds to a 4-dimensional vector of 64-bit floating point values.

It is typedef for `vtkm::Vec<vtkm::Float64, 4>`.

To help with indexing 2-, 3-, and 4- dimensional arrays, VTK-m provides the types `vtkm::Id2`, `vtkm::Id3`, and `vtkm::Id4`, which are `textidentifier{Vec}`s of type `vtkm::Id`. Likewise, VTK-m provides `vtkm::IdComponent2`, `vtkm::IdComponent3`, and `vtkm::IdComponent4`.

using `vtkm::Id2` = `vtkm::Vec<vtkm::Id, 2>`

`Id2` corresponds to a 2-dimensional index.

using `vtkm::Id3` = `vtkm::Vec<vtkm::Id, 3>`

`Id3` corresponds to a 3-dimensional index for 3d arrays.



Note that the precision of each index may be less than `vtkm::Id`.

```
using vtkm::Id4 = vtkm::Vec<vtkm::Id, 4>
```

Id4 corresponds to a 4-dimensional index.

```
using vtkm::IdComponent2 = vtkm::Vec<vtkm::IdComponent, 2>
```

IdComponent2 corresponds to an index to a local (small) 2-d array or equivalent.

```
using vtkm::IdComponent3 = vtkm::Vec<vtkm::IdComponent, 3>
```

IdComponent3 corresponds to an index to a local (small) 3-d array or equivalent.

```
using vtkm::IdComponent4 = vtkm::Vec<vtkm::IdComponent, 4>
```

IdComponent4 corresponds to an index to a local (small) 4-d array or equivalent.

VTK-m also provides types for `textidentifier{Vec}s` of integers of all varieties described in Section [ref{sec:IntegerTypes}](#). `vtkm::Vec2i`, `vtkm::Vec3i`, and `vtkm::Vec4i` are vectors of signed integers whereas `vtkm::Vec2ui`, `vtkm::Vec3ui`, and `vtkm::Vec4ui` are vectors of unsigned integers. All of these sport components of a width equal to `vtkm::Id`.

```
using vtkm::Vec2i = vtkm::Vec<vtkm::Id, 2>
```

Vec2i corresponds to a 2-dimensional vector of integer values.

Each integer value is of the default precision (i.e. `vtkm::Id`).

```
using vtkm::Vec3i = vtkm::Vec<vtkm::Id, 3>
```

Vec3i corresponds to a 3-dimensional vector of integer values.

Each integer value is of the default precision (i.e. `vtkm::Id`).

```
using vtkm::Vec4i = vtkm::Vec<vtkm::Id, 4>
```

Vec4i corresponds to a 4-dimensional vector of integer values.

Each integer value is of the default precision (i.e. `vtkm::Id`).

```
using vtkm::Vec2ui = vtkm::Vec<vtkm::UInt64, 2>
```

Vec2ui corresponds to a 2-dimensional vector of unsigned integer values.

Each integer value is of the default precision (following `vtkm::Id`).

```
using vtkm::Vec3ui = vtkm::Vec<vtkm::UInt64, 3>
```

Vec3ui corresponds to a 3-dimensional vector of unsigned integer values.

Each integer value is of the default precision (following `vtkm::Id`).

```
using vtkm::Vec4ui = vtkm::Vec<vtkm::UInt64, 4>
```

Vec4ui corresponds to a 4-dimensional vector of unsigned integer values.

Each integer value is of the default precision (following `vtkm::Id`).

The width can be specified by appending the desired number of bits in the same way as the floating point `textidentifier{Vec}s`. For example, `vtkm::Vec4ui_8` is a `textidentifier{Vec}` of 4 unsigned bytes.

using vtkm::Vec2i\_8 = vtkm::Vec<vtkm::Int8, 2>

Vec2i\_8 corresponds to a 2-dimensional vector of 8-bit integer values.

It is typedef for vtkm::Vec<vtkm::Int32, 2>.

using vtkm::Vec2ui\_8 = vtkm::Vec<vtkm::UInt8, 2>

Vec2ui\_8 corresponds to a 2-dimensional vector of 8-bit unsigned integer values.

It is typedef for vtkm::Vec<vtkm::UInt32, 2>.

using vtkm::Vec2i\_16 = vtkm::Vec<vtkm::Int16, 2>

Vec2i\_16 corresponds to a 2-dimensional vector of 16-bit integer values.

It is typedef for vtkm::Vec<vtkm::Int32, 2>.

using vtkm::Vec2ui\_16 = vtkm::Vec<vtkm::UInt16, 2>

Vec2ui\_16 corresponds to a 2-dimensional vector of 16-bit unsigned integer values.

It is typedef for vtkm::Vec<vtkm::UInt32, 2>.

using vtkm::Vec2i\_32 = vtkm::Vec<vtkm::Int32, 2>

Vec2i\_32 corresponds to a 2-dimensional vector of 32-bit integer values.

It is typedef for vtkm::Vec<vtkm::Int32, 2>.

using vtkm::Vec2ui\_32 = vtkm::Vec<vtkm::UInt32, 2>

Vec2ui\_32 corresponds to a 2-dimensional vector of 32-bit unsigned integer values.

It is typedef for vtkm::Vec<vtkm::UInt32, 2>.

using vtkm::Vec2i\_64 = vtkm::Vec<vtkm::Int64, 2>

Vec2i\_64 corresponds to a 2-dimensional vector of 64-bit integer values.

It is typedef for vtkm::Vec<vtkm::Int64, 2>.

using vtkm::Vec2ui\_64 = vtkm::Vec<vtkm::UInt64, 2>

Vec2ui\_64 corresponds to a 2-dimensional vector of 64-bit unsigned integer values.

It is typedef for vtkm::Vec<vtkm::UInt64, 2>.

using vtkm::Vec3i\_8 = vtkm::Vec<vtkm::Int8, 3>

Vec3i\_8 corresponds to a 3-dimensional vector of 8-bit integer values.

It is typedef for vtkm::Vec<vtkm::Int32, 3>.

using vtkm::Vec3ui\_8 = vtkm::Vec<vtkm::UInt8, 3>

Vec3ui\_8 corresponds to a 3-dimensional vector of 8-bit unsigned integer values.

It is typedef for vtkm::Vec<vtkm::UInt32, 3>.

using vtkm::Vec3i\_16 = vtkm::Vec<vtkm::Int16, 3>

Vec3i\_16 corresponds to a 3-dimensional vector of 16-bit integer values.

It is typedef for vtkm::Vec<vtkm::Int32, 3>.

using vtkm::Vec3ui\_16 = vtkm::Vec<vtkm::UInt16, 3>

Vec3ui\_16 corresponds to a 3-dimensional vector of 16-bit unsigned integer values.

It is typedef for vtkm::Vec<vtkm::UInt32, 3>.

using vtkm::Vec3i\_32 = vtkm::Vec<vtkm::Int32, 3>

Vec3i\_32 corresponds to a 3-dimensional vector of 32-bit integer values.

It is typedef for vtkm::Vec<vtkm::Int32, 3>.

using vtkm::Vec3ui\_32 = vtkm::Vec<vtkm::UInt32, 3>

Vec3ui\_32 corresponds to a 3-dimensional vector of 32-bit unsigned integer values.

It is typedef for vtkm::Vec<vtkm::UInt32, 3>.

using vtkm::Vec3i\_64 = vtkm::Vec<vtkm::Int64, 3>

Vec3i\_64 corresponds to a 3-dimensional vector of 64-bit integer values.

It is typedef for vtkm::Vec<vtkm::Int64, 3>.

using vtkm::Vec3ui\_64 = vtkm::Vec<vtkm::UInt64, 3>

Vec3ui\_64 corresponds to a 3-dimensional vector of 64-bit unsigned integer values.

It is typedef for vtkm::Vec<vtkm::UInt64, 3>.

using vtkm::Vec4i\_8 = vtkm::Vec<vtkm::Int8, 4>

Vec4i\_8 corresponds to a 4-dimensional vector of 8-bit integer values.

It is typedef for vtkm::Vec<vtkm::Int32, 4>.

using vtkm::Vec4ui\_8 = vtkm::Vec<vtkm::UInt8, 4>

Vec4ui\_8 corresponds to a 4-dimensional vector of 8-bit unsigned integer values.

It is typedef for vtkm::Vec<vtkm::UInt32, 4>.

using vtkm::Vec4i\_16 = vtkm::Vec<vtkm::Int16, 4>

Vec4i\_16 corresponds to a 4-dimensional vector of 16-bit integer values.

It is typedef for vtkm::Vec<vtkm::Int32, 4>.

using vtkm::Vec4ui\_16 = vtkm::Vec<vtkm::UInt16, 4>

Vec4ui\_16 corresponds to a 4-dimensional vector of 16-bit unsigned integer values.

It is typedef for vtkm::Vec<vtkm::UInt32, 4>.

using vtkm::Vec4i\_32 = vtkm::Vec<vtkm::Int32, 4>

Vec4i\_32 corresponds to a 4-dimensional vector of 32-bit integer values.

It is typedef for vtkm::Vec<vtkm::Int32, 4>.

using vtkm::Vec4ui\_32 = vtkm::Vec<vtkm::UInt32, 4>

Vec4ui\_32 corresponds to a 4-dimensional vector of 32-bit unsigned integer values.

It is typedef for vtkm::Vec<vtkm::UInt32, 4>.

```
using vtkm::Vec4i_64 = vtkm::Vec<vtkm::Int64, 4>
```

Vec4i\_64 corresponds to a 4-dimensional vector of 64-bit integer values.

It is typedef for `vtkm::Vec<vtkm::Int64, 4>`.

```
using vtkm::Vec4ui_64 = vtkm::Vec<vtkm::UInt64, 4>
```

Vec4ui\_64 corresponds to a 4-dimensional vector of 64-bit unsigned integer values.

It is typedef for `vtkm::Vec<vtkm::UInt64, 4>`.

These types really just scratch the surface of the Vec types available in VTK-m and the things that can be done with them. See [Chapter 20 \(Advanced Types\)](#) for more information on Vec types and what can be done with them.

## VTK-M VERSION

As the VTK-m code evolves, changes to the interface and behavior will inevitably happen. Consequently, code that links into VTK-m might need a specific version of VTK-m or changes its behavior based on what version of VTK-m it is using. To facilitate this, VTK-m software is managed with a versioning system and advertises its version in multiple ways. As with many software products, VTK-m has three version numbers: major, minor, and patch. The major version represents significant changes in the VTK-m implementation and interface. Changes in the major version include backward incompatible changes. The minor version represents added functionality. Generally, changes in the minor version do not introduce changes to the API. The patch version represents fixes provided after a release occurs. Patch versions represent minimal change and do not add features.

If you are writing a software package that is managed by CMake and load VTK-m with the `find_package()` command as described in [Section 2.4 \(Linking to VTK-m\)](#), then you can query the VTK-m version directly in the CMake configuration. When you load VTK-m with `find_package()`, CMake sets the variables `VTKm_VERSION_MAJOR`, `VTKm_VERSION_MINOR`, and `VTKm_VERSION_PATCH` to the major, minor, and patch versions, respectively. Additionally, `VTKm_VERSION` is set to the “major.minor” version number and `VTKm_VERSION_FULL` is set to the “major.minor.patch” version number. If the current version of VTK-m is actually a development version that is in between releases of VTK-m, then an abbreviated SHA of the git commit is also included as part of `VTKm_VERSION_FULL`.

---

### Did You Know?

If you have a specific version of VTK-m required for your software, you can also use the version option to the `find_package()` CMake command. The `find_package()` command takes an optional version argument that causes the command to fail if the wrong version of the package is found.

---

It is also possible to query the VTK-m version directly in your code through preprocessor macros. The `vtkm/Version.h` header file defines the following preprocessor macros to identify the VTK-m version.

#### **VTKM\_VERSION**

The version number of the loaded VTK-m package. This is in the form “major.minor”.

#### **VTKM\_VERSION\_FULL**

The extended version number of the VTK-m package including patch and in-between-release information. This is in the form “major.minor.patch[.gitsha1]” where “gitsha” is only included if the source code is in between releases.

#### **VTKM\_VERSION\_MAJOR**

The major VTK-m version number.

#### **VTKM\_VERSION\_MINOR**

The minor VTK-m version number.

#### **VTKM\_VERSION\_PATCH**

The patch VTK-m version number.

---

### Common Errors

Note that the CMake variables all begin with `VTkm_` (lowercase “m”) whereas the preprocessor macros begin with `VTKM_` (all uppercase). This follows the respective conventions of CMake variables and preprocessor macros.

---

Note that `vtkm/Version.h` does not include any other VTK-m header files. This gives your code a chance to load, query, and react to the VTK-m version before loading any VTK-m code proper.

## INITIALIZATION

When it comes to running VTK-m code, there are a few ways in which various facilities, such as logging device connections, and device configuration parameters, can be initialized. The preferred method of initializing these features is to run the `vtkm::cont::Initialize()` function. Although it is not strictly necessary to call `vtkm::cont::Initialize()`, it is recommended to set up state and check for available devices.

```
InitializeResult vtkm::cont::Initialize(int &argc, char *argv[], InitializeOptions opts =  
                                         InitializeOptions::None)
```

Initialize the VTKm library, parsing arguments when provided:

- Sets log level names when logging is configured.
- Sets the calling thread as the main thread for logging purposes.
- Sets the default log level to the argument provided to `--vtkm-log-level`.
- Forces usage of the device name passed to `--vtkm-device`.
- Prints usage when `-h` or `--vtkm-help` is passed.

The parameterless version only sets up log level names.

Additional options may be supplied via the *opts* argument, such as requiring the `--vtkm-device` option.

Results are available in the returned *InitializeResult*.

---

**Note:** This method may call `exit()` on parse error.

---

`vtkm::cont::Initialize()` can be called without any arguments, in which case VTK-m will be initialized with defaults. But it can also optionally take the `argc` and `argv` arguments to the main function to parse some options that control the state of VTK-m. VTK-m accepts arguments that, for example, configure the compute device to use or establish logging levels. Any arguments that are handled by VTK-m are removed from the `argc/argv` list so that your program can then respond to the remaining arguments.

`vtkm::cont::Initialize()` returns a `vtkm::cont::InitializeResult` structure. This structure contains information about the supported arguments and options selected during initialization.

struct **InitializeResult**

## Public Members

*DeviceAdapterId* **Device** = *DeviceAdapterTagUndefined*{ }

The device passed into `--vtkm-device` argument.

If no device was specified, then this value is set to *DeviceAdapterTagUndefined*. Note that if the user specifies “any” device, then this value can be set to *DeviceAdapterTagAny*, which is a pseudo-tag that allows any supported device.

`std::string` **Usage**

A usage statement for arguments parsed by VTK-m.

If the calling code wants to print a usage statement documenting the options that can be provided on the command line, then this string can be added to document the options supported by VTK-m.

*vtkm::cont::Initialize()* takes an optional third argument that specifies some options on the behavior of the argument parsing. The options are specified as a bit-wise “or” of fields specified in the *vtkm::cont::InitializeOptions* enum.

enum class *vtkm::cont::InitializeOptions*

*Values:*

enumerator **None**

Placeholder used when no options are enabled.

This is the value used when the third argument to *vtkm::cont::Initialize* is not provided.

enumerator **RequireDevice**

Issue an error if the device argument is not specified.

enumerator **DefaultAnyDevice**

If no device is specified, treat it as if the user gave `--vtkm-device=Any`.

This means that *DeviceAdapterTagUndefined* will never be returned in the result.

enumerator **AddHelp**

Add a help argument.

If `-h` or `--vtkm-help` is provided, prints a usage statement. Of course, the usage statement will only print out arguments processed by VTK-m, which is why help is not given by default. Alternatively, a string with usage help is returned from *vtkm::cont::Initialize* so that the calling program can provide VTK-m’s help in its own usage statement.

enumerator **ErrorOnBadOption**

If an unknown option is encountered, the program terminates with an error and a usage statement is printed.

If this option is not provided, any unknown options are returned in `argv`. If this option is used, it is a good idea to use `AddHelp` as well.

enumerator **ErrorOnBadArgument**

If an extra argument is encountered, the program terminates with an error and a usage statement is printed.

If this option is not provided, any unknown arguments are returned in `argv`.



enumerator **Strict**

If supplied, Initialize treats its own arguments as the only ones supported by the application and provides an error if not followed exactly.

This is a convenience option that is a combination of `ErrorOnBadOption`, `ErrorOnBadArgument`, and `AddHelp`.

Example 1: Calling `vtkm::cont::Initialize()`.

```

1  #include <vtkm/cont/Initialize.h>
2
3  int main(int argc, char** argv)
4  {
5      vtkm::cont::InitializeOptions options =
6          vtkm::cont::InitializeOptions::ErrorOnBadOption |
7          vtkm::cont::InitializeOptions::DefaultAnyDevice;
8      vtkm::cont::InitializeResult config = vtkm::cont::Initialize(argc, argv, options);
9
10     if (argc != 2)
11     {
12         std::cerr << "USAGE: " << argv[0] << " [options] filename" << std::endl;
13         std::cerr << "Available options are:" << std::endl;
14         std::cerr << config.Usage << std::endl;
15         return 1;
16     }
17     std::string filename = argv[1];
18
19     // Do something cool with VTK-m
20     // ...
21
22     return 0;
23 }
```



## DATA SETS

A *data set*, implemented with the `vtkm::cont::DataSet` class, contains and manages the geometric data structures that VTK-m operates on.

### class **DataSet**

Contains and manages the geometric data structures that VTK-m operates on.

A *DataSet* is the main data structure used by VTK-m to pass data in and out of filters, rendering, and other components. A data set comprises the following 3 data structures.

- *CellSet* A cell set describes topological connections. A cell set defines some number of points in space and how they connect to form cells, filled regions of space. A data set has exactly one cell set.
- *Field* A field describes numerical data associated with the topological elements in a cell set. The field is represented as an array, and each entry in the field array corresponds to a topological element (point, edge, face, or cell). Together the cell set topology and discrete data values in the field provide an interpolated function throughout the volume of space covered by the data set. A cell set can have any number of fields.
- *CoordinateSystem* A coordinate system is a special field that describes the physical location of the points in a data set. Although it is most common for a data set to contain a single coordinate system, VTK-m supports data sets with no coordinate system such as abstract data structures like graphs that might not have positions in a space. *DataSet* also supports multiple coordinate systems for data that have multiple representations for position. For example, geospatial data could simultaneously have coordinate systems defined by 3D position, latitude-longitude, and any number of 2D projections.

In addition to the base `vtkm::cont::DataSet`, VTK-m provides `vtkm::cont::PartitionedDataSet` to represent data partitioned into multiple domains. A `vtkm::cont::PartitionedDataSet` is implemented as a collection of `vtkm::cont::DataSet` objects. Partitioned data sets are described later in [Section 7.5 \(Partitioned Data Sets\)](#).

## 7.1 Building Data Sets

Before we go into detail on the cell sets, fields, and coordinate systems that make up a data set in VTK-m, let us first discuss how to build a data set. One simple way to build a data set is to load data from a file using the `vtkm::io` module. Reading files is discussed in detail in [Chapter 8 \(File I/O\)](#).

This section describes building data sets of different types using a set of classes named *DataSetBuilder\**, which provide a convenience layer on top of `vtkm::cont::DataSet` to make it easier to create data sets.

---

### Did You Know?

To simplify the introduction of `vtkm::cont::DataSet` objects, this section uses the simplest mechanisms. In many cases this involves loading data in a `std::vector` and passing that to VTK-m, which usually causes the data to be copied. This is not the most efficient method to load data into VTK-m. Although it is sufficient for small data or data that come from a “slow” source, such as a file, it might be a bottleneck for large data generated by another library. It is possible to adapt VTK-m’s `vtkm::cont::DataSet` to externally defined data. This is done by wrapping existing data into what is called `ArrayHandle`, but this is a more advanced topic that will not be addressed in this chapter. `ArrayHandle` objects are introduced in [Chapter 17 \(Basic Array Handles\)](#) and more adaptive techniques are described in later chapters.

---

## 7.1.1 Creating Uniform Grids

Uniform grids are meshes that have a regular array structure with points uniformly spaced parallel to the axes. Uniform grids are also sometimes called regular grids or images.

The `vtkm::cont::DataSetBuilderUniform` class can be used to easily create 2- or 3-dimensional uniform grids. `vtkm::cont::DataSetBuilderUniform` has several versions of a method named `vtkm::cont::DataSetBuilderUniform::Create()` that takes the number of points in each dimension, the origin, and the spacing. The origin is the location of the first point of the data (in the lower left corner), and the spacing is the distance between points in the x, y, and z directions.

class **DataSetBuilderUniform**

### Public Static Functions

```
template<typename T>
static inline vtkm::cont::DataSet Create(const vtkm::Id &dimension, const T &origin, const T &spacing,
                                         const std::string &coordNm = "coords")
```

Create a 1D uniform `DataSet`.

#### Parameters

- **dimension** – [in] The size of the grid. The dimensions are specified based on the number of points (as opposed to the number of cells).
- **origin** – [in] The origin of the data. This is the point coordinate with the minimum value in all dimensions.
- **spacing** – [in] The uniform distance between adjacent points.
- **coordNm** – [in] (optional) The name to register the coordinates as.

```
static vtkm::cont::DataSet Create(const vtkm::Id &dimension, const std::string &coordNm = "coords")
```

Create a 1D uniform `DataSet`.

The origin is set to 0 and the spacing is set to 1.

#### Parameters

- **dimension** – [in] The size of the grid. The dimensions are specified based on the number of points (as opposed to the number of cells).
- **coordNm** – [in] (optional) The name to register the coordinates as.

```
template<typename T>
static inline vtkm::cont::DataSet Create(const vtkm::Id2 &dimensions, const vtkm::Vec<T, 2> &origin, const
                                         vtkm::Vec<T, 2> &spacing, const std::string &coordNm = "coords")
```

Create a 2D uniform `DataSet`.

**Parameters**

- **dimensions** – [in] The size of the grid. The dimensions are specified based on the number of points (as opposed to the number of cells).
- **origin** – [in] The origin of the data. This is the point coordinate with the minimum value in all dimensions.
- **spacing** – [in] The uniform distance between adjacent points.
- **coordNm** – [in] (optional) The name to register the coordinates as.

static vtkm::cont::DataSet Create(const vtkm::Id2 &dimensions, const std::string &coordNm = "coords")

Create a 2D uniform DataSet.

The origin is set to (0,0) and the spacing is set to (1,1).

**Parameters**

- **dimensions** – [in] The size of the grid. The dimensions are specified based on the number of points (as opposed to the number of cells).
- **coordNm** – [in] (optional) The name to register the coordinates as.

template<typename T>

static inline vtkm::cont::DataSet Create(const vtkm::Id3 &dimensions, const vtkm::Vec<T, 3> &origin, const vtkm::Vec<T, 3> &spacing, const std::string &coordNm = "coords")

Create a 3D uniform DataSet.

**Parameters**

- **dimensions** – [in] The size of the grid. The dimensions are specified based on the number of points (as opposed to the number of cells).
- **origin** – [in] The origin of the data. This is the point coordinate with the minimum value in all dimensions.
- **spacing** – [in] The uniform distance between adjacent points.
- **coordNm** – [in] (optional) The name to register the coordinates as.

static vtkm::cont::DataSet Create(const vtkm::Id3 &dimensions, const std::string &coordNm = "coords")

Create a 3D uniform DataSet.

The origin is set to (0,0,0) and the spacing is set to (1,1,1).

**Parameters**

- **dimensions** – [in] The size of the grid. The dimensions are specified based on the number of points (as opposed to the number of cells).
- **coordNm** – [in] (optional) The name to register the coordinates as.

The following example creates a `vtkm::cont::DataSet` containing a uniform grid of  $101 \times 101 \times 26$  points.

Example 1: Creating a uniform grid. {.cxx}

```
1  vtkm::cont::DataSetBuilderUniform dataSetBuilder;
2
3  vtkm::cont::DataSet dataSet = dataSetBuilder.Create(vtkm::Id3(101, 101, 26));
```

If not specified, the origin will be at the coordinates (0,0,0) and the spacing will be 1 in each direction. Thus, in the previous example the width, height, and depth of the mesh in physical space will be 100, 100, and  $25$ , respectively, and the mesh will be centered at (50, 50, 12.5). Let us say we actually want a mesh of the same dimensions,

but we want the  $z$  direction to be stretched out so that the mesh will be the same size in each direction, and we want the mesh centered at the origin.

Example 2: Creating a uniform grid with custom origin and spacing.

```
1 vtkm::cont::DataSetBuilderUniform dataSetBuilder;
2
3 vtkm::cont::DataSet dataSet = dataSetBuilder.Create(vtkm::Id3(101, 101, 26),
4                                                     vtkm::Vec3f(-50.0, -50.0, -50.0),
5                                                     vtkm::Vec3f(1.0, 1.0, 4.0));
```

## 7.1.2 Creating Rectilinear Grids

A rectilinear grid is similar to a uniform grid except that a rectilinear grid can adjust the spacing between adjacent grid points. This allows the rectilinear grid to have tighter sampling in some areas of space, but the points are still constrained to be aligned with the axes and each other. The irregular spacing of a rectilinear grid is specified by providing a separate array each for the  $x$ ,  $y$ , and  $z$  coordinates.

The `vtkm::cont::DataSetBuilderRectilinear` class can be used to easily create 2- or 3-dimensional rectilinear grids. `vtkm::cont::DataSetBuilderRectilinear` has several versions of a method named `vtkm::cont::DataSetBuilderRectilinear::Create()` that takes these coordinate arrays and builds a `vtkm::cont::DataSet` out of them. The arrays can be supplied as either standard C arrays or as `std::vector` objects, in which case the data in the arrays are copied into the `vtkm::cont::DataSet`. These arrays can also be passed as `vtkm::cont::ArrayHandle` objects (introduced later in this book), in which case the data are shallow copied.

class `DataSetBuilderRectilinear`

### Public Static Functions

```
template<typename T>
static inline vtkm::cont::DataSet Create(const std::vector<T> &xvals, const std::string &coordNm =
                                         "coords")
```

Create a 1D rectilinear `DataSet`.

A rectilinear grid is specified with a scalar array for the point coordinates in the  $x$  direction. In this form, the coordinate array is specified with `std::vector`. The data is copied from the `std::vector`.

#### Parameters

- **xvals** – [in] An array of coordinates to use along the  $x$  dimension.
- **coordNm** – [in] (optional) The name to register the coordinates as.

```
template<typename T>
static inline vtkm::cont::DataSet Create(vtkm::Id nx, T *xvals, const std::string &coordNm = "coords")
```

Create a 1D rectilinear `DataSet`.

A rectilinear grid is specified with a scalar array for the point coordinates in the  $x$  direction. In this form, the coordinate array is specified with a standard C array. The data is copied from the array.

#### Parameters

- **nx** – [in] The size of the grid in the  $x$  direction (and length of the `xvals` array).
- **xvals** – [in] An array of coordinates to use along the  $x$  dimension.
- **coordNm** – [in] (optional) The name to register the coordinates as.

```
template<typename T>
static inline vtkm::cont::DataSet Create(const vtkm::cont::ArrayHandle<T> &xvals, const std::string
                                     &coordNm = "coords")
```

Create a 1D rectilinear *DataSet*.

A rectilinear grid is specified with a scalar array for the point coordinates in the x direction. In this form, the coordinate array is specified with *vtkm::cont::ArrayHandle*. The *ArrayHandle* is shared with the *DataSet*, so changing the *ArrayHandle* changes the *DataSet*.

#### Parameters

- **xvals** – [in] An array of coordinates to use along the x dimension.
- **coordNm** – [in] (optional) The name to register the coordinates as.

```
template<typename T>
static inline vtkm::cont::DataSet Create(const std::vector<T> &xvals, const std::vector<T> &yvals, const
                                     std::string &coordNm = "coords")
```

Create a 2D rectilinear *DataSet*.

A rectilinear grid is specified with separate arrays for the point coordinates in the x and y directions. In this form, the coordinate arrays are specified with *std::vector*. The data is copied from the *std::vectors*.

#### Parameters

- **xvals** – [in] An array of coordinates to use along the x dimension.
- **yvals** – [in] An array of coordinates to use along the x dimension.
- **coordNm** – [in] (optional) The name to register the coordinates as.

```
template<typename T>
static inline vtkm::cont::DataSet Create(vtkm::Id nx, vtkm::Id ny, T *xvals, T *yvals, const std::string
                                     &coordNm = "coords")
```

Create a 2D rectilinear *DataSet*.

A rectilinear grid is specified with separate arrays for the point coordinates in the x and y directions. In this form, the coordinate arrays are specified with standard C arrays. The data is copied from the arrays.

#### Parameters

- **nx** – [in] The size of the grid in the x direction (and length of the *xvals* array).
- **ny** – [in] The size of the grid in the x direction (and length of the *yvals* array).
- **xvals** – [in] An array of coordinates to use along the x dimension.
- **yvals** – [in] An array of coordinates to use along the x dimension.
- **coordNm** – [in] (optional) The name to register the coordinates as.

```
template<typename T>
static inline vtkm::cont::DataSet Create(const vtkm::cont::ArrayHandle<T> &xvals, const
                                     vtkm::cont::ArrayHandle<T> &yvals, const std::string &coordNm =
                                     "coords")
```

Create a 2D rectilinear *DataSet*.

A rectilinear grid is specified with separate arrays for the point coordinates in the x and y directions. In this form, the coordinate arrays are specified with *vtkm::cont::ArrayHandle*. The *ArrayHandles* are shared with the *DataSet*, so changing the *ArrayHandles* changes the *DataSet*.

#### Parameters

- **xvals** – [in] An array of coordinates to use along the x dimension.

- **yvals** – [in] An array of coordinates to use along the x dimension.
- **coordNm** – [in] (optional) The name to register the coordinates as.

```
template<typename T>
static inline vtkm::cont::DataSet Create(vtkm::Id nx, vtkm::Id ny, vtkm::Id nz, T *xvals, T *yvals, T *zvals,
                                         const std::string &coordNm = "coords")
```

Create a 3D rectilinear *DataSet*.

A rectilinear grid is specified with separate arrays for the point coordinates in the x, y, and z directions. In this form, the coordinate arrays are specified with standard C arrays. The data is copied from the arrays.

#### Parameters

- **nx** – [in] The size of the grid in the x direction (and length of the *xvals* array).
- **ny** – [in] The size of the grid in the x direction (and length of the *yvals* array).
- **nz** – [in] The size of the grid in the x direction (and length of the *zvals* array).
- **xvals** – [in] An array of coordinates to use along the x dimension.
- **yvals** – [in] An array of coordinates to use along the x dimension.
- **zvals** – [in] An array of coordinates to use along the x dimension.
- **coordNm** – [in] (optional) The name to register the coordinates as.

```
template<typename T>
static inline vtkm::cont::DataSet Create(const std::vector<T> &xvals, const std::vector<T> &yvals, const
                                         std::vector<T> &zvals, const std::string &coordNm = "coords")
```

Create a 3D rectilinear *DataSet*.

A rectilinear grid is specified with separate arrays for the point coordinates in the x, y, and z directions. In this form, the coordinate arrays are specified with `std::vector`. The data is copied from the `std::vectors`.

#### Parameters

- **xvals** – [in] An array of coordinates to use along the x dimension.
- **yvals** – [in] An array of coordinates to use along the x dimension.
- **zvals** – [in] An array of coordinates to use along the x dimension.
- **coordNm** – [in] (optional) The name to register the coordinates as.

```
template<typename T>
static inline vtkm::cont::DataSet Create(const vtkm::cont::ArrayHandle<T> &xvals, const
                                         vtkm::cont::ArrayHandle<T> &yvals, const
                                         vtkm::cont::ArrayHandle<T> &zvals, const std::string &coordNm =
                                         "coords")
```

Create a 3D rectilinear *DataSet*.

A rectilinear grid is specified with separate arrays for the point coordinates in the x, y, and z directions. In this form, the coordinate arrays are specified with `vtkm::cont::ArrayHandle`. The *ArrayHandles* are shared with the *DataSet*, so changing the *ArrayHandles* changes the *DataSet*.

#### Parameters

- **xvals** – [in] An array of coordinates to use along the x dimension.
- **yvals** – [in] An array of coordinates to use along the x dimension.
- **zvals** – [in] An array of coordinates to use along the x dimension.



- **coordNm** – [in] (optional) The name to register the coordinates as.

The following example creates a `vtkm::cont::DataSet` containing a rectilinear grid with  $201 \times 201 \times 101$  points with different irregular spacing along each axis.

Example 3: Creating a rectilinear grid.

```

1 // Make x coordinates range from -4 to 4 with tighter spacing near 0.
2 std::vector<vtkm::Float32> xCoordinates;
3 for (vtkm::Float32 x = -2.0f; x <= 2.0f; x += 0.02f)
4 {
5     xCoordinates.push_back(vtkm::CopySign(x * x, x));
6 }
7
8 // Make y coordinates range from 0 to 2 with tighter spacing near 2.
9 std::vector<vtkm::Float32> yCoordinates;
10 for (vtkm::Float32 y = 0.0f; y <= 4.0f; y += 0.02f)
11 {
12     yCoordinates.push_back(vtkm::Sqrt(y));
13 }
14
15 // Make z coordinates range from -1 to 1 with even spacing.
16 std::vector<vtkm::Float32> zCoordinates;
17 for (vtkm::Float32 z = -1.0f; z <= 1.0f; z += 0.02f)
18 {
19     zCoordinates.push_back(z);
20 }
21
22 vtkm::cont::DataSetBuilderRectilinear dataSetBuilder;
23
24 vtkm::cont::DataSet dataSet =
25     dataSetBuilder.Create(xCoordinates, yCoordinates, zCoordinates);

```

### 7.1.3 Creating Explicit Meshes

An explicit mesh is an arbitrary collection of cells with arbitrary connections. It can have multiple different types of cells. Explicit meshes are also known as unstructured grids. Explicit meshes can contain cells of different shapes. The shapes that VTK-m currently supports are listed in [Figure 1](#). Each shape is identified using either a numeric identifier, provided by VTK-m with identifiers of the form `vtkm::CELL_SHAPE_*` or special tag structures of the form `vtkm::CellSetTag*`. Cell shapes are discussed in detail in [Chapter 26 \(Working with Cells\)](#).

The cells of an explicit mesh are defined with the following 3 arrays, which are depicted graphically in [Figure 2](#).

#### Shapes

An array of ids identifying the shape of the cell. Each value is a `vtkm::UInt8` and should be set to one of the `vtkm::CELL_SHAPE_*` constants. The shapes and their identifiers are shown in [Figure 1](#). The size of this array is equal to the number of cells in the set.

#### Connectivity

An array that lists all the points that comprise each cell. Each entry in the array is a `vtkm::Id` giving the point id associated with a vertex of a cell. The points for each cell are given in a prescribed order for each shape, which is also shown in [Figure 1](#). The point indices are stored consecutively from the first cell to the last.

#### Offsets

An array of `vtkm::Id`'s pointing to the index in the connectivity array where the points for a particular cell

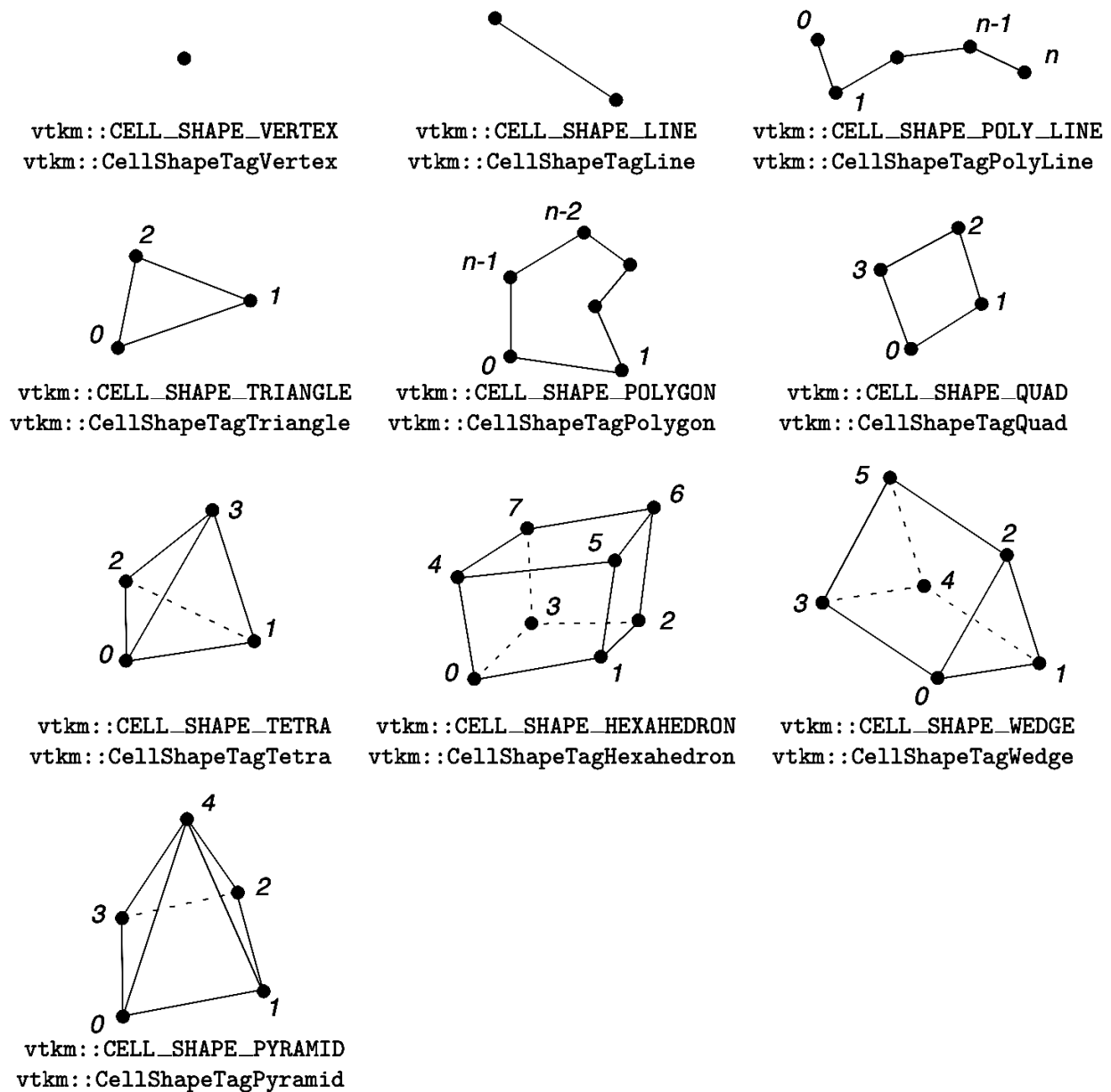


Figure 1: Basic Cell Shapes.

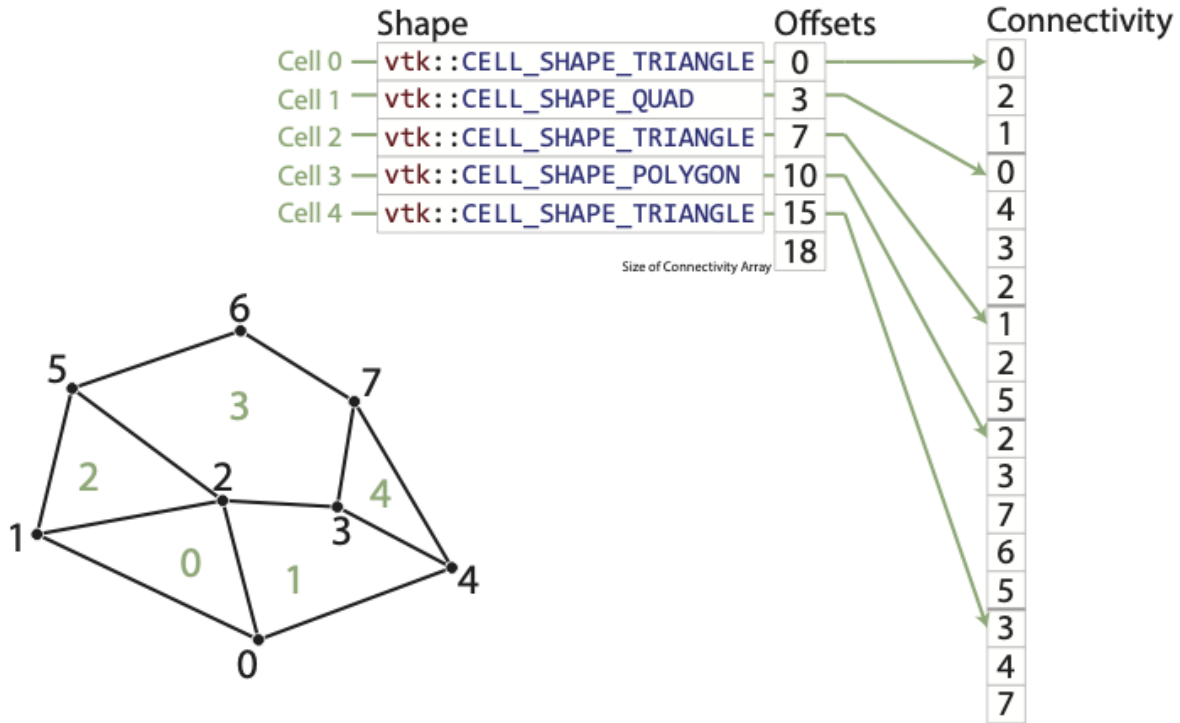


Figure 2: An example explicit mesh.

starts. The size of this array is equal to the number of cells in the set plus 1. The first entry is expected to be 0 (since the connectivity of the first cell is at the start of the connectivity array). The last entry, which does not correspond to any cell, should be the size of the connectivity array.

One important item that is missing from this list of arrays is a count of the number of indices associated with each cell. This is not explicitly represented in VTK-m's mesh structure because it can be implicitly derived from the offsets array by subtracting consecutive entries. However, it is usually the case when building an explicit mesh that you will have an array of these counts rather than the offsets. It is for this reason that VTK-m contains mechanisms to build an explicit data set with a "num indices" arrays rather than an offsets array.

The `vtkm::cont::DataSetBuilderExplicit` class can be used to create data sets with explicit meshes. `vtkm::cont::DataSetBuilderExplicit` has several versions of a method named `vtkm::cont::DataSetBuilderExplicit::Create()`. Generally, these methods take the shapes, number of indices, and connectivity arrays as well as an array of point coordinates.

class `DataSetBuilderExplicit`

## Public Static Functions

```
template<typename T>
static inline vtkm::cont::DataSet Create(const std::vector<T> &xVals, const std::vector<vtkm::UInt8>
                                         &shapes, const std::vector<vtkm::IdComponent> &numIndices,
                                         const std::vector<vtkm::Id> &connectivity, const std::string
                                         &coordsNm = "coords")
```

Create a 1D *DataSet* with arbitrary cell connectivity.

The cell connectivity is specified with arrays defining the shape and point connections of each cell. In this form, the cell connectivity and coordinates are specified as `std::vector` and the data will be copied to create the data object.

### Parameters

- **xVals** – [in] An array providing the x coordinate of each point.
- **shapes** – [in] An array of shapes for each cell. Each entry should be one of the `vtkm::CELL_SHAPE_*` values identifying the shape of the corresponding cell.
- **numIndices** – [in] An array containing for each cell the number of points incident on that cell.
- **connectivity** – [in] An array specifying for each cell the indices of points incident on each cell. Each cell has a short array of indices that reference points in the *coords* array. The length of each of these short arrays is specified by the *numIndices* array. These variable length arrays are tightly packed together in this *connectivity* array.
- **coordsNm** – [in] (optional) The name to register the coordinates as.

```
template<typename T>
static inline vtkm::cont::DataSet Create(const std::vector<T> &xVals, const std::vector<T> &yVals, const
                                         std::vector<vtkm::UInt8> &shapes, const
                                         std::vector<vtkm::IdComponent> &numIndices, const
                                         std::vector<vtkm::Id> &connectivity, const std::string &coordsNm =
                                         "coords")
```

Create a 2D *DataSet* with arbitrary cell connectivity.

The cell connectivity is specified with arrays defining the shape and point connections of each cell. In this form, the cell connectivity and coordinates are specified as `std::vector` and the data will be copied to create the data object.

### Parameters

- **xVals** – [in] An array providing the x coordinate of each point.
- **yVals** – [in] An array providing the x coordinate of each point.
- **shapes** – [in] An array of shapes for each cell. Each entry should be one of the `vtkm::CELL_SHAPE_*` values identifying the shape of the corresponding cell.
- **numIndices** – [in] An array containing for each cell the number of points incident on that cell.
- **connectivity** – [in] An array specifying for each cell the indices of points incident on each cell. Each cell has a short array of indices that reference points in the *coords* array. The length of each of these short arrays is specified by the *numIndices* array. These variable length arrays are tightly packed together in this *connectivity* array.
- **coordsNm** – [in] (optional) The name to register the coordinates as.

```
template<typename T>
```

```
static inline vtkm::cont::DataSet Create(const std::vector<T> &xVals, const std::vector<T> &yVals, const
std::vector<T> &zVals, const std::vector<vtkm::UInt8> &shapes,
const std::vector<vtkm::IdComponent> &numIndices, const
std::vector<vtkm::Id> &connectivity, const std::string &coordsNm =
"coords")
```

Create a 3D *DataSet* with arbitrary cell connectivity.

The cell connectivity is specified with arrays defining the shape and point connections of each cell. In this form, the cell connectivity and coordinates are specified as `std::vector` and the data will be copied to create the data object.

#### Parameters

- **xVals** – [in] An array providing the x coordinate of each point.
- **yVals** – [in] An array providing the x coordinate of each point.
- **zVals** – [in] An array providing the x coordinate of each point.
- **shapes** – [in] An array of shapes for each cell. Each entry should be one of the `vtkm::CELL_SHAPE_*` values identifying the shape of the corresponding cell.
- **numIndices** – [in] An array containing for each cell the number of points incident on that cell.
- **connectivity** – [in] An array specifying for each cell the indices of points incident on each cell. Each cell has a short array of indices that reference points in the *coords* array. The length of each of these short arrays is specified by the *numIndices* array. These variable length arrays are tightly packed together in this *connectivity* array.
- **coordsNm** – [in] (optional) The name to register the coordinates as.

```
template<typename T>
static inline vtkm::cont::DataSet Create(const std::vector<vtkm::Vec<T, 3>> &coords, const
std::vector<vtkm::UInt8> &shapes, const
std::vector<vtkm::IdComponent> &numIndices, const
std::vector<vtkm::Id> &connectivity, const std::string &coordsNm =
"coords")
```

Create a 3D *DataSet* with arbitrary cell connectivity.

The cell connectivity is specified with arrays defining the shape and point connections of each cell. In this form, the cell connectivity and coordinates are specified as `std::vector` and the data will be copied to create the data object.

#### Parameters

- **coords** – [in] An array of point coordinates.
- **shapes** – [in] An array of shapes for each cell. Each entry should be one of the `vtkm::CELL_SHAPE_*` values identifying the shape of the corresponding cell.
- **numIndices** – [in] An array containing for each cell the number of points incident on that cell.
- **connectivity** – [in] An array specifying for each cell the indices of points incident on each cell. Each cell has a short array of indices that reference points in the *coords* array. The length of each of these short arrays is specified by the *numIndices* array. These variable length arrays are tightly packed together in this *connectivity* array.
- **coordsNm** – [in] (optional) The name to register the coordinates as.

```
template<typename T>
```

```
static inline vtkm::cont::DataSet Create(const vtkm::cont::ArrayHandle<vtkm::Vec<T, 3>> &coords, const
                                       vtkm::cont::ArrayHandle<vtkm::UInt8> &shapes, const
                                       vtkm::cont::ArrayHandle<vtkm::IdComponent> &numIndices,
                                       const vtkm::cont::ArrayHandle<vtkm::Id> &connectivity, const
                                       std::string &coordsNm = "coords")
```

Create a 3D *DataSet* with arbitrary cell connectivity.

The cell connectivity is specified with arrays defining the shape and point connections of each cell. In this form, the cell connectivity and coordinates are specified as *ArrayHandle* and the memory will be shared with the created data object. That said, the *DataSet* construction will generate a new array for offsets.

#### Parameters

- **coords** – [in] An array of point coordinates.
- **shapes** – [in] An array of shapes for each cell. Each entry should be one of the `vtkm::CELL_SHAPE_*` values identifying the shape of the corresponding cell.
- **numIndices** – [in] An array containing for each cell the number of points incident on that cell.
- **connectivity** – [in] An array specifying for each cell the indices of points incident on each cell. Each cell has a short array of indices that reference points in the *coords* array. The length of each of these short arrays is specified by the *numIndices* array. These variable length arrays are tightly packed together in this *connectivity* array.
- **coordsNm** – [in] (optional) The name to register the coordinates as.

```
template<typename T, typename CellShapeTag>
static inline vtkm::cont::DataSet Create(const std::vector<vtkm::Vec<T, 3>> &coords, CellShapeTag tag,
                                       vtkm::IdComponent numberOfPointsPerCell, const
                                       std::vector<vtkm::Id> &connectivity, const std::string &coordsNm =
                                       "coords")
```

Create a 3D *DataSet* with arbitrary cell connectivity for a single cell type.

The cell connectivity is specified with an array defining the point connections of each cell. All the cells in the *DataSet* are of the same shape and contain the same number of incident points. In this form, the cell connectivity and coordinates are specified as `std::vector` and the data will be copied to create the data object.

#### Parameters

- **coords** – [in] An array of point coordinates.
- **tag** – [in] A tag object representing the shape of all the cells in the mesh. Cell shape tag objects have a name of the form `vtkm::CellShapeTag*` such as `vtkm::CellShapeTagTriangle` or `vtkm::CellShapeTagHexahedron`. To specify a cell shape determined at runtime, use `vtkm::CellShapeTagGeneric`.
- **numberOfPointsPerCell** – [in] The number of points that are incident to each cell.
- **connectivity** – [in] An array specifying for each cell the indices of points incident on each cell. Each cell has a short array of indices that reference points in the *coords* array. The length of each of these short arrays is specified by *numberOfPointsPerCell*. These short arrays are tightly packed together in this *connectivity* array.
- **coordsNm** – [in] (optional) The name to register the coordinates as.

```
template<typename T, typename CellShapeTag>
```

```
static inline vtkm::cont::DataSet Create(const vtkm::cont::ArrayHandle<vtkm::Vec<T, 3>> &coords,
                                       CellShapeTag tag, vtkm::IdComponent numberOfPointsPerCell,
                                       const vtkm::cont::ArrayHandle<vtkm::Id> &connectivity, const
                                       std::string &coordsNm = "coords")
```

Create a 3D *DataSet* with arbitrary cell connectivity for a single cell type.

The cell connectivity is specified with an array defining the point connections of each cell. All the cells in the *DataSet* are of the same shape and contain the same number of incident points. In this form, the cell connectivity and coordinates are specified as *ArrayHandle* and the memory will be shared with the created data object.

#### Parameters

- **coords** – [in] An array of point coordinates.
- **tag** – [in] A tag object representing the shape of all the cells in the mesh. Cell shape tag objects have a name of the form *vtkm::CellShapeTag\** such as *vtkm::CellShapeTagTriangle* or *vtkm::CellShapeTagHexahedron*. To specify a cell shape determined at runtime, use *vtkm::CellShapeTagGeneric*.
- **numberOfPointsPerCell** – [in] The number of points that are incident to each cell.
- **connectivity** – [in] An array specifying for each cell the indices of points incident on each cell. Each cell has a short array of indices that reference points in the *coords* array. The length of each of these short arrays is specified by *numberOfPointsPerCell*. These short arrays are tightly packed together in this *connectivity* array.
- **coordsNm** – [in] (optional) The name to register the coordinates as.

The following example creates a mesh like the one shown in Figure 2.

Example 4: Creating an explicit mesh with  
*vtkm::cont::DataSetBuilderExplicit*.

```
1 // Array of point coordinates.
2 std::vector<vtkm::Vec3f_32> pointCoordinates;
3 pointCoordinates.push_back(vtkm::Vec3f_32(1.1f, 0.0f, 0.0f));
4 pointCoordinates.push_back(vtkm::Vec3f_32(0.2f, 0.4f, 0.0f));
5 pointCoordinates.push_back(vtkm::Vec3f_32(0.9f, 0.6f, 0.0f));
6 pointCoordinates.push_back(vtkm::Vec3f_32(1.4f, 0.5f, 0.0f));
7 pointCoordinates.push_back(vtkm::Vec3f_32(1.8f, 0.3f, 0.0f));
8 pointCoordinates.push_back(vtkm::Vec3f_32(0.4f, 1.0f, 0.0f));
9 pointCoordinates.push_back(vtkm::Vec3f_32(1.0f, 1.2f, 0.0f));
10 pointCoordinates.push_back(vtkm::Vec3f_32(1.5f, 0.9f, 0.0f));
11
12 // Array of shapes.
13 std::vector<vtkm::UInt8> shapes;
14 shapes.push_back(vtkm::CELL_SHAPE_TRIANGLE);
15 shapes.push_back(vtkm::CELL_SHAPE_QUAD);
16 shapes.push_back(vtkm::CELL_SHAPE_TRIANGLE);
17 shapes.push_back(vtkm::CELL_SHAPE_POLYGON);
18 shapes.push_back(vtkm::CELL_SHAPE_TRIANGLE);
19
20 // Array of number of indices per cell.
21 std::vector<vtkm::IdComponent> numIndices;
22 numIndices.push_back(3);
23 numIndices.push_back(4);
```

(continues on next page)

(continued from previous page)

```

24  numIndices.push_back(3);
25  numIndices.push_back(5);
26  numIndices.push_back(3);
27
28  // Connectivity array.
29  std::vector<vtkm::Id> connectivity;
30  connectivity.push_back(0); // Cell 0
31  connectivity.push_back(2);
32  connectivity.push_back(1);
33  connectivity.push_back(0); // Cell 1
34  connectivity.push_back(4);
35  connectivity.push_back(3);
36  connectivity.push_back(2);
37  connectivity.push_back(1); // Cell 2
38  connectivity.push_back(2);
39  connectivity.push_back(5);
40  connectivity.push_back(2); // Cell 3
41  connectivity.push_back(3);
42  connectivity.push_back(7);
43  connectivity.push_back(6);
44  connectivity.push_back(5);
45  connectivity.push_back(3); // Cell 4
46  connectivity.push_back(4);
47  connectivity.push_back(7);
48
49  // Copy these arrays into a DataSet.
50  vtkm::cont::DataSetBuilderExplicit dataSetBuilder;
51
52  vtkm::cont::DataSet dataSet =
53    dataSetBuilder.Create(pointCoordinates, shapes, numIndices, connectivity);

```

Often it is awkward to build your own arrays and then pass them to `vtkm::cont::DataSetBuilderExplicit`. There also exists an alternate builder class named `vtkm::cont::DataSetBuilderExplicitIterative` that allows you to specify each cell and point one at a time rather than all at once. This is done by calling one of the versions of `vtkm::cont::DataSetBuilderExplicitIterative::AddPoint()` and one of the versions of `vtkm::cont::DataSetBuilderExplicitIterative::AddCell()` for each point and cell, respectively.

### class `DataSetBuilderExplicitIterative`

Helper class to build a `DataSet` by iteratively adding points and cells.

This class allows you to specify a `DataSet` by adding points and cells one at a time.

### Public Functions

void **Begin**(const std::string &coordName = "coords")

Begin defining points and cells of a `DataSet`.

The state of this object is initialized to be ready to use `AddPoint` and `AddCell` methods.

#### Parameters

**coordName** – [in] (optional) The name to register the coordinates as.



vtkm::Id **AddPoint**(const vtkm::Vec3f &pt)

Add a point to the *DataSet*.

**Parameters**

**pt** – [in] The coordinates of the point to add.

**Returns**

The index of the newly created point.

template<typename T>

inline vtkm::Id **AddPoint**(const vtkm::Vec<T, 3> &pt)

Add a point to the *DataSet*.

**Parameters**

**pt** – [in] The coordinates of the point to add.

**Returns**

The index of the newly created point.

vtkm::Id **AddPoint**(const vtkm::FloatDefault &x, const vtkm::FloatDefault &y, const vtkm::FloatDefault &z = 0)

Add a point to the *DataSet*.

**Parameters**

- **x** – [in] The x coordinate of the newly created point.
- **y** – [in] The y coordinate of the newly created point.
- **z** – [in] The z coordinate of the newly created point.

**Returns**

The index of the newly created point.

template<typename T>

inline vtkm::Id **AddPoint**(const T &x, const T &y, const T &z = 0)

Add a point to the *DataSet*.

**Parameters**

- **x** – [in] The x coordinate of the newly created point.
- **y** – [in] The y coordinate of the newly created point.
- **z** – [in] The z coordinate of the newly created point.

**Returns**

The index of the newly created point.

void **AddCell**(const vtkm::UInt8 &shape, const std::vector<vtkm::Id> &conn)

Add a cell to the *DataSet*.

**Parameters**

- **shape** – [in] Identifies the shape of the cell. Use one of the vtkm::CELL\_SHAPE\_\* values.
- **conn** – [in] List of indices to the incident points.

void **AddCell**(const vtkm::UInt8 &shape, const vtkm::Id \*conn, const vtkm::IdComponent &n)

Add a cell to the *DataSet*.

**Parameters**

- **shape** – [in] Identifies the shape of the cell. Use one of the vtkm::CELL\_SHAPE\_\* values.

- **conn** – [in] *List* of indices to the incident points.
- **n** – [in] The number of incident points (and the length of the **conn** array).

void **AddCell**(vtkm::UInt8 shape)

Start adding a cell to the *DataSet*.

The incident points are later added one at a time using **AddCellPoint**. The cell is completed the next time **AddCell** or **Create** is called.

#### Parameters

**shape** – [in] Identifies the shape of the cell. Use one of the

void **AddCellPoint**(vtkm::Id pointIndex)

Add an incident point to the current cell.

#### Parameters

**pointIndex** – [in] Index to the incident point.

vtkm::cont::DataSet **Create**()

Produce the *DataSet*.

The points and cells previously added are finalized and the resulting *DataSet* is returned.

The next example also builds the mesh shown in Figure 2 except this time using *vtkm::cont::DataSetBuilderExplicitIterative*.

Example 5: Creating an explicit mesh with  
*vtkm::cont::DataSetBuilderExplicitIterative*.

```

1  vtkm::cont::DataSetBuilderExplicitIterative dataSetBuilder;
2
3  dataSetBuilder.AddPoint(1.1, 0.0, 0.0);
4  dataSetBuilder.AddPoint(0.2, 0.4, 0.0);
5  dataSetBuilder.AddPoint(0.9, 0.6, 0.0);
6  dataSetBuilder.AddPoint(1.4, 0.5, 0.0);
7  dataSetBuilder.AddPoint(1.8, 0.3, 0.0);
8  dataSetBuilder.AddPoint(0.4, 1.0, 0.0);
9  dataSetBuilder.AddPoint(1.0, 1.2, 0.0);
10 dataSetBuilder.AddPoint(1.5, 0.9, 0.0);
11
12 dataSetBuilder.AddCell(vtkm::CELL_SHAPE_TRIANGLE);
13 dataSetBuilder.AddCellPoint(0);
14 dataSetBuilder.AddCellPoint(2);
15 dataSetBuilder.AddCellPoint(1);
16
17 dataSetBuilder.AddCell(vtkm::CELL_SHAPE_QUAD);
18 dataSetBuilder.AddCellPoint(0);
19 dataSetBuilder.AddCellPoint(4);
20 dataSetBuilder.AddCellPoint(3);
21 dataSetBuilder.AddCellPoint(2);
22
23 dataSetBuilder.AddCell(vtkm::CELL_SHAPE_TRIANGLE);
24 dataSetBuilder.AddCellPoint(1);
25 dataSetBuilder.AddCellPoint(2);
26 dataSetBuilder.AddCellPoint(5);
27

```

(continues on next page)

(continued from previous page)

```

28  dataSetBuilder.AddCell(vtkm::CELL_SHAPE_POLYGON);
29  dataSetBuilder.AddCellPoint(2);
30  dataSetBuilder.AddCellPoint(3);
31  dataSetBuilder.AddCellPoint(7);
32  dataSetBuilder.AddCellPoint(6);
33  dataSetBuilder.AddCellPoint(5);
34
35  dataSetBuilder.AddCell(vtkm::CELL_SHAPE_TRIANGLE);
36  dataSetBuilder.AddCellPoint(3);
37  dataSetBuilder.AddCellPoint(4);
38  dataSetBuilder.AddCellPoint(7);
39
40  vtkm::cont::DataSet dataSet = dataSetBuilder.Create();

```

## 7.1.4 Add Fields

In addition to creating the geometric structure of a data set, it is usually important to add fields to the data. Fields describe numerical data associated with the topological elements in a cell. They often represent a physical quantity (such as temperature, mass, or volume fraction) but can also represent other information (such as indices or classifications).

The easiest way to define fields in a data set is to use the `vtkm::cont::DataSet::AddPointField()` and `vtkm::cont::DataSet::AddCellField()` methods. Each of these methods take a requisite field name and the array with field data.

Both `vtkm::cont::DataSet::AddPointField()` and `vtkm::cont::DataSet::AddCellField()` are overloaded to accept arrays of data in different structures. Field arrays can be passed as standard C arrays or as `std::vector`'s, in which case the data are copied. Field arrays can also be passed in a `ArrayHandle` (introduced later in this book), in which case the data are not copied.

```
inline void vtkm::cont::DataSet::AddPointField(const std::string &fieldName, const
                                              vtkm::cont::UnknownArrayHandle &field)
```

Adds a point field of a given name to the `DataSet`.

Note that the indexing of fields is not the same as the order in which they are added, and that adding a field can arbitrarily reorder the integer indexing of all the fields. To retrieve a specific field, retrieve the field by name, not by integer index.

```
template<typename T>
inline void vtkm::cont::DataSet::AddPointField(const std::string &fieldName, const std::vector<T> &field)
```

Adds a point field of a given name to the `DataSet`.

Note that the indexing of fields is not the same as the order in which they are added, and that adding a field can arbitrarily reorder the integer indexing of all the fields. To retrieve a specific field, retrieve the field by name, not by integer index.

```
template<typename T>
inline void vtkm::cont::DataSet::AddPointField(const std::string &fieldName, const T *field, const vtkm::Id
                                              &n)
```

Adds a point field of a given name to the `DataSet`.

Note that the indexing of fields is not the same as the order in which they are added, and that adding a field can arbitrarily reorder the integer indexing of all the fields. To retrieve a specific field, retrieve the field by name, not by integer index.

```
inline void vtkm::cont::DataSet::AddCellField(const std::string &fieldName, const
                                             vtkm::cont::UnknownArrayHandle &field)
```

Adds a cell field of a given name to the *DataSet*.

Note that the indexing of fields is not the same as the order in which they are added, and that adding a field can arbitrarily reorder the integer indexing of all the fields. To retrieve a specific field, retrieve the field by name, not by integer index.

```
template<typename T>
inline void vtkm::cont::DataSet::AddCellField(const std::string &fieldName, const std::vector<T> &field)
```

Adds a cell field of a given name to the *DataSet*.

Note that the indexing of fields is not the same as the order in which they are added, and that adding a field can arbitrarily reorder the integer indexing of all the fields. To retrieve a specific field, retrieve the field by name, not by integer index.

```
template<typename T>
inline void vtkm::cont::DataSet::AddCellField(const std::string &fieldName, const T *field, const vtkm::Id
                                             &n)
```

Adds a cell field of a given name to the *DataSet*.

Note that the indexing of fields is not the same as the order in which they are added, and that adding a field can arbitrarily reorder the integer indexing of all the fields. To retrieve a specific field, retrieve the field by name, not by integer index.

The following (somewhat contrived) example defines fields for a uniform grid that identify which points and cells are on the boundary of the mesh.

Example 6: Adding fields to a *vtkm::cont::DataSet*.

```
1 // Make a simple structured data set.
2 const vtkm::Id3 pointDimensions(20, 20, 10);
3 const vtkm::Id3 cellDimensions = pointDimensions - vtkm::Id3(1, 1, 1);
4 vtkm::cont::DataSetBuilderUniform dataSetBuilder;
5 vtkm::cont::DataSet dataSet = dataSetBuilder.Create(pointDimensions);
6
7 // Create a field that identifies points on the boundary.
8 std::vector<vtkm::UInt8> boundaryPoints;
9 for (vtkm::Id zIndex = 0; zIndex < pointDimensions[2]; zIndex++)
10 {
11     for (vtkm::Id yIndex = 0; yIndex < pointDimensions[1]; yIndex++)
12     {
13         for (vtkm::Id xIndex = 0; xIndex < pointDimensions[0]; xIndex++)
14         {
15             if ((xIndex == 0) || (xIndex == pointDimensions[0] - 1) || (yIndex == 0) ||
16                 (yIndex == pointDimensions[1] - 1) || (zIndex == 0) ||
17                 (zIndex == pointDimensions[2] - 1))
18             {
19                 boundaryPoints.push_back(1);
20             }
21             else
22             {
23                 boundaryPoints.push_back(0);
24             }
25         }
26     }
27 }
```

(continues on next page)

(continued from previous page)

```

27 }
28
29 dataSet.AddPointField("boundary_points", boundaryPoints);
30
31 // Create a field that identifies cells on the boundary.
32 std::vector<vtkm::UInt8> boundaryCells;
33 for (vtkm::Id zIndex = 0; zIndex < cellDimensions[2]; zIndex++)
34 {
35     for (vtkm::Id yIndex = 0; yIndex < cellDimensions[1]; yIndex++)
36     {
37         for (vtkm::Id xIndex = 0; xIndex < cellDimensions[0]; xIndex++)
38         {
39             if ((xIndex == 0) || (xIndex == cellDimensions[0] - 1) || (yIndex == 0) ||
40                 (yIndex == cellDimensions[1] - 1) || (zIndex == 0) ||
41                 (zIndex == cellDimensions[2] - 1))
42             {
43                 boundaryCells.push_back(1);
44             }
45             else
46             {
47                 boundaryCells.push_back(0);
48             }
49         }
50     }
51 }
52
53 dataSet.AddCellField("boundary_cells", boundaryCells);

```

## 7.2 Cell Sets

A cell set determines the topological structure of the data in a data set.

### class **CellSet**

Defines the topological structure of the data in a *DataSet*.

Fundamentally, any cell set is a collection of cells, which typically (but not always) represent some region in space.

Subclassed by *vtkm::cont::CellSetExplicit< vtkm::cont::ArrayHandleConstant< vtkm::UInt8 >::StorageTag, vtkm::cont::StorageTagBasic, vtkm::cont::ArrayHandleCounting< vtkm::Id >::StorageTag >*, *vtkm::cont::CellSetExplicit< ShapesStorageTag, ConnectivityStorageTag, OffsetsStorageTag >*, *vtkm::cont::CellSetExtrude*, *vtkm::cont::CellSetPermutation< OriginalCellSetType\_, PermutationArrayHandleType\_ >*, *vtkm::cont::CellSetStructured< DIMENSION >*

## Public Functions

virtual vtkm::Id **GetNumberOfCells**() const = 0

Get the number of cells in the topology.

virtual vtkm::Id **GetNumberOfPoints**() const = 0

Get the number of points in the topology.

virtual vtkm::UInt8 **GetCellShape**(vtkm::Id id) const = 0

Get the shell shape of a particular cell.

virtual vtkm::IdComponent **GetNumberOfPointsInCell**(vtkm::Id id) const = 0

Get the number of points incident to a particular cell.

virtual void **GetCellPointIds**(vtkm::Id id, vtkm::Id \*ptids) const = 0

Get a list of points incident to a particular cell.

virtual std::shared\_ptr<CellSet> **NewInstance**() const = 0

Return a new *CellSet* that is the same derived class.

virtual void **DeepCopy**(const *CellSet* \*src) = 0

Copy the provided *CellSet* into this object.

The provided *CellSet* must be the same type as this one.

virtual void **PrintSummary**(std::ostream&) const = 0

Print a summary of this cell set.

virtual void **ReleaseResourcesExecution**() = 0

Remove the *CellSet* from any devices.

Any memory used on a device to store this object will be deleted. However, the data will still remain on the host.

3D cells are made up of *points*, *edges*, and *faces*. (2D cells have only points and edges, and 1D cells have only points.) [Figure 3](#) shows the relationship between a cell's shape and these topological elements. The arrangement of these points, edges, and faces is defined by the *shape* of the cell, which prescribes a specific ordering of each. The basic cell shapes provided by VTK-m are discussed in detail in [Chapter 26 \(Working with Cells\)](#).



Figure 3: The relationship between a cell shape and its topological elements (points, edges, and faces).

There are multiple ways to express the connections of a cell set, each with different benefits and restrictions. These different cell set types are managed by different cell set classes in VTK-m. All VTK-m cell set classes inherit from `vtkm::cont::CellSet`. The two basic types of cell sets are structured and explicit, and there are several variations of these types.

## 7.2.1 Structured Cell Sets

```
template<vtkm::IdComponent DIMENSION>
```

```
class CellSetStructured : public vtkm::cont::CellSet
```

Defines a 1-, 2-, or 3-dimensional structured grid of points.

The structured cells form lines, quadrilaterals, or hexahedra for 1-, 2-, or 3-dimensions, respectively, to connect the points. The topology is specified by simply providing the dimensions, which is the number of points in the i, j, and k directions of the grid of points.

### Public Functions

```
inline virtual vtkm::Id GetNumberOfCells() const override
```

Get the number of cells in the topology.

```
inline virtual vtkm::Id GetNumberOfPoints() const override
```

Get the number of points in the topology.

```
inline virtual void ReleaseResourcesExecution() override
```

Remove the *CellSet* from any devices.

Any memory used on a device to store this object will be deleted. However, the data will still remain on the host.

```
inline void SetPointDimensions(SchedulingRangeType dimensions)
```

Set the dimensions of the structured array of points.

```
inline SchedulingRangeType GetPointDimensions() const
```

Get the dimensions of the points.

```
inline virtual vtkm::IdComponent GetNumberOfPointsInCell(vtkm::Id = 0) const override
```

Get the number of points incident to a particular cell.

```
inline virtual vtkm::UInt8 GetCellShape(vtkm::Id = 0) const override
```

Get the shell shape of a particular cell.

```
inline virtual void GetCellPointIds(vtkm::Id id, vtkm::Id *ptids) const override
```

Get a list of points incident to a particular cell.

```
inline virtual std::shared_ptr<CellSet> NewInstance() const override
```

Return a new *CellSet* that is the same derived class.

```
inline virtual void DeepCopy(const CellSet *src) override
```

Copy the provided *CellSet* into this object.

The provided *CellSet* must be the same type as this one.

```
inline virtual void PrintSummary(std::ostream &out) const override
```

Print a summary of this cell set.

The number of points in a *vtkm::cont::CellSetStructured* is implicitly  $i \times j \times k$  and the number of cells is implicitly  $(i - 1) \times (j - 1) \times (k - 1)$  (for 3D grids). [Figure 4](#) demonstrates this arrangement.

The big advantage of using *vtkm::cont::CellSetStructured* to define a cell set is that it is very space efficient because the entire topology can be defined by the three integers specifying the dimensions. Also, algorithms can be optimized for *vtkm::cont::CellSetStructured*'s regular nature. However, *vtkm::cont::CellSetStructured*'s

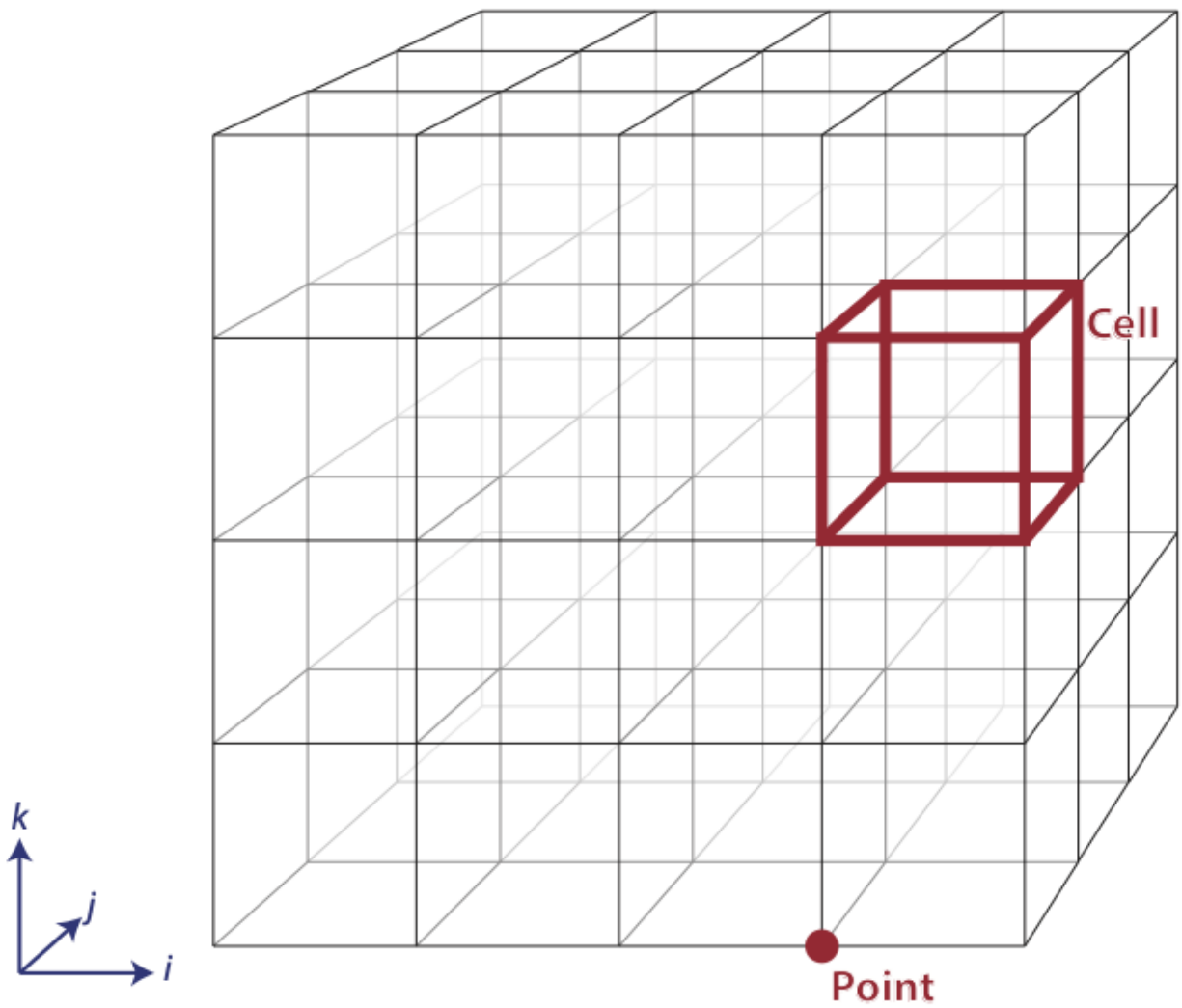


Figure 4: The arrangement of points and cells in a 3D structured grid.



strictly regular grid also limits its applicability. A structured cell set can only be a dense grid of lines, quadrilaterals, or hexahedra. It cannot represent irregular data well.

Many data models in other software packages, such as the one for VTK, make a distinction between uniform, rectilinear, and curvilinear grids. VTK-m's cell sets do not. All three of these grid types are represented by `vtkm::cont::CellSetStructured`. This is because in a VTK-m data set the cell set and the coordinate system are defined independently and used interchangeably. A structured cell set with uniform point coordinates makes a uniform grid. A structured cell set with point coordinates defined irregularly along coordinate axes makes a rectilinear grid. And a structured cell set with arbitrary point coordinates makes a curvilinear grid. The point coordinates are defined by the data set's coordinate system, which is discussed in [Section 7.4 \(Coordinate Systems\)](#).

## 7.2.2 Explicit Cell Sets

```
template<typename ShapesStorageTag = ::vtkm::cont::StorageTagBasic, typename ConnectivityStorageTag =  
::vtkm::cont::StorageTagBasic, typename OffsetsStorageTag = ::vtkm::cont::StorageTagBasic>  
class CellSetExplicit : public vtkm::cont::CellSet
```

Defines an irregular collection of cells.

The cells can be of different types and connected in arbitrary ways. This is done by explicitly providing for each cell a sequence of points that defines the cell.

### Public Functions

```
virtual vtkm::Id GetNumberOfCells() const override
```

Get the number of cells in the topology.

```
virtual vtkm::Id GetNumberOfPoints() const override
```

Get the number of points in the topology.

```
virtual void PrintSummary(std::ostream &out) const override
```

Print a summary of this cell set.

```
virtual void ReleaseResourcesExecution() override
```

Remove the `CellSet` from any devices.

Any memory used on a device to store this object will be deleted. However, the data will still remain on the host.

```
virtual std::shared_ptr<CellSet> NewInstance() const override
```

Return a new `CellSet` that is the same derived class.

```
virtual void DeepCopy(const CellSet *src) override
```

Copy the provided `CellSet` into this object.

The provided `CellSet` must be the same type as this one.

```
virtual vtkm::IdComponent GetNumberOfPointsInCell(vtkm::Id cellid) const override
```

Get the number of points incident to a particular cell.

```
virtual void GetCellPointIds(vtkm::Id id, vtkm::Id *ptids) const override
```

Get a list of points incident to a particular cell.

```
virtual vtkm::UInt8 GetCellShape(vtkm::Id cellid) const override
```

Get the shell shape of a particular cell.

void **PrepareToAddCells**(vtkm::Id numCells, vtkm::Id connectivityMaxLen)

Start adding cells one at a time.

After this method is called, `AddCell` is called repeatedly to add each cell. Once all cells are added, call `CompleteAddingCells`.

template<typename IdVecType>

void **AddCell**(vtkm::UInt8 cellType, vtkm::IdComponent numVertices, const IdVecType &ids)

Add a cell.

This can only be called after `AddCell`.

void **CompleteAddingCells**(vtkm::Id numPoints)

Finish adding cells one at a time.

void **Fill**(vtkm::Id numPoints, const vtkm::cont::ArrayHandle<vtkm::UInt8, ShapesStorageTag> &cellTypes, const vtkm::cont::ArrayHandle<vtkm::Id, ConnectivityStorageTag> &connectivity, const vtkm::cont::ArrayHandle<vtkm::Id, OffsetsStorageTag> &offsets)

Set all the cells of the mesh.

This method can be used to fill the memory from another system without copying data.

The types of cell sets are listed in [Figure 5](#).

An explicit cell set is defined with a minimum of three arrays. The first array identifies the shape of each cell. (Identifiers for cell shapes are shown in [Figure 5](#).) The second array has a sequence of point indices that make up each cell. The third array identifies an offset into the second array where the point indices for each cell is found plus an extra entry at the end set to the size of the second array. [Figure 6](#) shows a simple example of an explicit cell set.

An explicit cell set can also identify the number of indices defined for each cell by subtracting consecutive entries in the offsets array. It is often the case when creating a `vtkm::cont::CellSetExplicit` that you have an array containing the number of indices rather than the offsets. Such an array can be converted to an offsets array that can be used with `vtkm::cont::CellSetExplicit` by using the `vtkm::cont::ConvertNumComponentsToOffsets()` convenience function.

```
void vtkm::cont::ConvertNumComponentsToOffsets(const vtkm::cont::UnknownArrayHandle
                                              &numComponentsArray,
                                              vtkm::cont::ArrayHandle<vtkm::Id> &offsetsArray,
                                              vtkm::Id &componentsArraySize,
                                              vtkm::cont::DeviceAdapterId device =
                                              vtkm::cont::DeviceAdapterTagAny{ })
```

`ConvertNumComponentsToOffsets` takes an array of *Vec* sizes (i.e.

the number of components in each *Vec*) and returns an array of offsets to a packed array of such *Vectors*. The resulting array can be used with `ArrayHandleGroupVecVariable`.

Note that this function is pre-compiled for some set of `ArrayHandle` types. If you get a warning about an inefficient conversion (or the operation fails outright), you might need to use `vtkm::cont::internal::ConvertNumComponentsToOffsetsTemplate`.

#### Parameters

- **numComponentsArray** – [in] the input array that specifies the number of components in each group *Vec*.
- **offsetsArray** – [out] (optional) the output `ArrayHandle`, which must have a value type of `vtkm::Id`. If the output `ArrayHandle` is not given, it is returned.

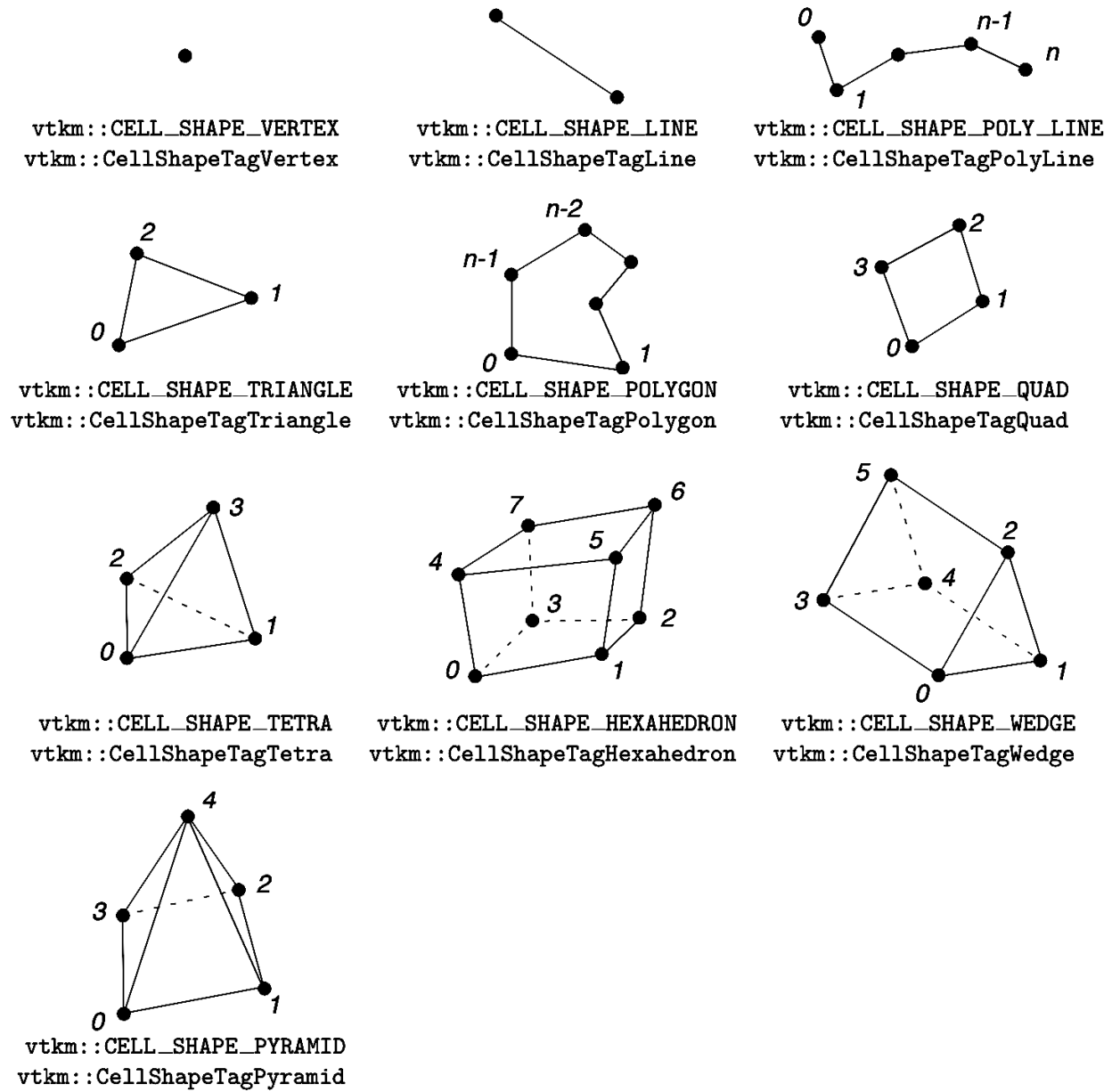


Figure 5: Basic Cell Shapes in a `vtkm::cont::CellSetExplicit`.

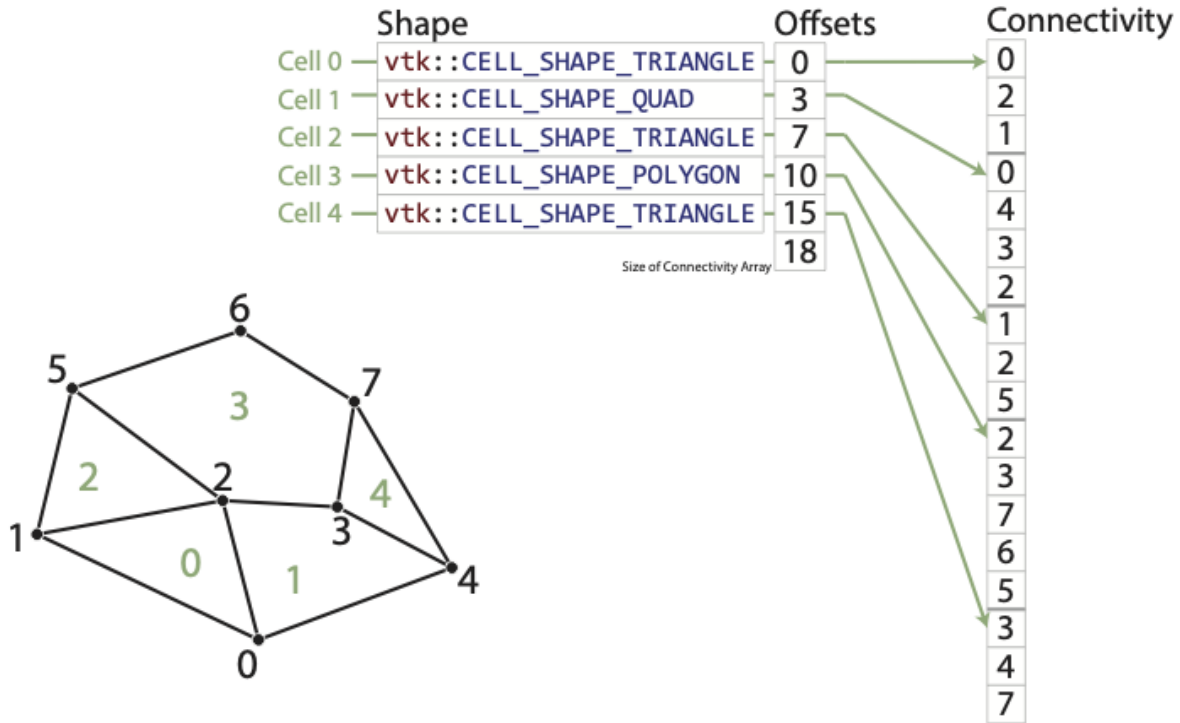


Figure 6: Example of cells in a `vtkm::cont::CellSetExplicit` and the arrays that define them.

- **componentsArraySize** – [in] (optional) a reference to a `vtkm::Id` and is filled with the expected size of the component values array.
- **device** – [in] (optional) specifies the device on which to run the conversion.

`vtkm::cont::CellSetExplicit` is a powerful representation for a cell set because it can represent an arbitrary collection of cells. However, because all connections must be explicitly defined, `vtkm::cont::CellSetExplicit` requires a significant amount of memory to represent the topology.

An important specialization of an explicit cell set is `vtkm::cont::CellSetSingleType`.

```
template<typename ConnectivityStorageTag = ::vtkm::cont::StorageTagBasic>
```

```
class CellSetSingleType : public
```

```
vtkm::cont::CellSetExplicit<vtkm::cont::ArrayHandleConstant<vtkm::UInt8>::StorageTag,
```

```
::vtkm::cont::StorageTagBasic, vtkm::cont::ArrayHandleCounting<vtkm::Id>::StorageTag>
```

An explicit cell set with all cells of the same shape.

`CellSetSingleType` is an explicit cell set constrained to contain cells that all have the same shape and all have the same number of points. So, for example if you are creating a surface that you know will contain only triangles, `CellSetSingleType` is a good representation for these data.

Using `CellSetSingleType` saves memory because the array of cell shapes and the array of point counts no longer need to be stored. `CellSetSingleType` also allows VTK-m to skip some processing and other storage required for general explicit cell sets.

## Public Functions

inline void **PrepareToAddCells**(vtkm::Id numCells, vtkm::Id connectivityMaxLen)

Start adding cells one at a time.

After this method is called, `AddCell` is called repeatedly to add each cell. Once all cells are added, call `CompleteAddingCells`.

template<typename **IdVecType**>

inline void **AddCell**(vtkm::UInt8 shapeId, vtkm::IdComponent numVertices, const *IdVecType* &ids)

Add a cell.

This can only be called after `AddCell`.

inline void **CompleteAddingCells**(vtkm::Id numPoints)

Finish adding cells one at a time.

inline void **Fill**(vtkm::Id numPoints, vtkm::UInt8 shapeId, vtkm::IdComponent numberOfPointsPerCell, const vtkm::cont::ArrayHandle<vtkm::Id, *ConnectivityStorageTag*> &connectivity)

Set all the cells of the mesh.

This method can be used to fill the memory from another system without copying data.

inline virtual vtkm::UInt8 **GetCellShape**(vtkm::Id) const override

Get the shell shape of a particular cell.

inline virtual std::shared\_ptr<*CellSet*> **NewInstance**() const override

Return a new *CellSet* that is the same derived class.

inline virtual void **DeepCopy**(const *CellSet* \*src) override

Copy the provided *CellSet* into this object.

The provided *CellSet* must be the same type as this one.

inline virtual void **PrintSummary**(std::ostream &out) const override

Print a summary of this cell set.

### 7.2.3 Cell Set Permutations

To rearrange, and possibly subsample, cells in a *CellSet*, use *vtkm::cont::CellSetPermutation* to define a new set without copying.

template<typename **OriginalCellSetType**\_, typename **PermutationArrayHandleType**\_ =

vtkm::cont::ArrayHandle<vtkm::Id, ::vtkm::cont::StorageTagBasic>>

class **CellSetPermutation** : public vtkm::cont::CellSet

Rearranges the cells of one cell set to create another cell set.

This restructuring of cells is not done by copying data to a new structure. Rather, *CellSetPermutation* establishes a look-up from one cell structure to another. Cells are permuted on the fly while algorithms are run.

A *CellSetPermutation* is established by providing a mapping array that for every cell index provides the equivalent cell index in the cell set being permuted. *CellSetPermutation* is most often used to mask out cells in a data set so that algorithms will skip over those cells when running.

## Public Functions

inline **CellSetPermutation**(const PermutationArrayHandleType &validCellIds, const OriginalCellSetType &cellset)

Create a *CellSetPermutation*.

### Parameters

- **validCellIds** – [in] An array that defines the permutation. If index  $i$  is value  $j$ , then the  $i$ th cell of this cell set will be the same as the  $j$ th cell in the original *cellset*.
- **cellset** – [in] The original cell set that this one is permuting.

inline const OriginalCellSetType &**GetFullCellSet**() const

Returns the original *CellSet* that this one is permuting.

inline const PermutationArrayHandleType &**GetValidCellIds**() const

Returns the array used to permute the cell indices.

inline virtual vtkm::Id **GetNumberOfCells**() const override

Get the number of cells in the topology.

inline virtual vtkm::Id **GetNumberOfPoints**() const override

Get the number of points in the topology.

inline virtual void **ReleaseResourcesExecution**() override

Remove the *CellSet* from any devices.

Any memory used on a device to store this object will be deleted. However, the data will still remain on the host.

inline virtual vtkm::IdComponent **GetNumberOfPointsInCell**(vtkm::Id cellIndex) const override

Get the number of points incident to a particular cell.

inline virtual vtkm::UInt8 **GetCellShape**(vtkm::Id id) const override

Get the shell shape of a particular cell.

inline virtual void **GetCellPointIds**(vtkm::Id id, vtkm::Id \*ptids) const override

Get a list of points incident to a particular cell.

inline virtual std::shared\_ptr<*CellSet*> **NewInstance**() const override

Return a new *CellSet* that is the same derived class.

inline virtual void **DeepCopy**(const *CellSet* \*src) override

Copy the provided *CellSet* into this object.

The provided *CellSet* must be the same type as this one.

inline void **Fill**(const PermutationArrayHandleType &validCellIds, const OriginalCellSetType &cellset)

Set the topology.

### Parameters

- **validCellIds** – [in] An array that defines the permutation. If index  $i$  is value  $j$ , then the  $i$ th cell of this cell set will be the same as the  $j$ th cell in the original *cellset*.
- **cellset** – [in] The original cell set that this one is permuting.

```
inline virtual void PrintSummary(std::ostream &out) const override
    Print a summary of this cell set.
```

---

### Did You Know?

Although `vtkm::cont::CellSetPermutation` can mask cells, it cannot mask points. All points from the original cell set are available in the permuted cell set regardless of whether they are used.

---

The following example uses `vtkm::cont::CellSetPermutation` with a counting array to expose every tenth cell. This provides a simple way to subsample a data set.

Example 7: Subsampling a data set with  
`vtkm::cont::CellSetPermutation`.

```
1 // Create a simple data set.
2 vtkm::cont::DataSetBuilderUniform dataSetBuilder;
3 vtkm::cont::DataSet originalDataSet = dataSetBuilder.Create(vtkm::Id3(33, 33, 26));
4 vtkm::cont::CellSetStructured<3> originalCellSet;
5 originalDataSet.GetCellSet().AsCellSet(originalCellSet);
6
7 // Create a permutation array for the cells. Each value in the array refers
8 // to a cell in the original cell set. This particular array selects every
9 // 10th cell.
10 vtkm::cont::ArrayHandleCounting<vtkm::Id> permutationArray(0, 10, 2560);
11
12 // Create a permutation of that cell set containing only every 10th cell.
13 vtkm::cont::CellSetPermutation<vtkm::cont::CellSetStructured<3>,
14                               vtkm::cont::ArrayHandleCounting<vtkm::Id>>
15   permutedCellSet(permutationArray, originalCellSet);
```

## 7.2.4 Cell Set Extrude

class **CellSetExtrude** : public `vtkm::cont::CellSet`

Defines a 3-dimensional extruded mesh representation.

`CellSetExtrude` takes takes a mesh defined in the XZ-plane and extrudes it along the Y-axis. This plane is repeated in a series of steps and forms wedge cells between them.

The extrusion can be linear or rotational (e.g., to form a torus).

### Public Functions

virtual `vtkm::Id` **GetNumberOfCells**() const override

Get the number of cells in the topology.

virtual `vtkm::Id` **GetNumberOfPoints**() const override

Get the number of points in the topology.

virtual `vtkm::UInt8` **GetCellShape**(`vtkm::Id` id) const override

Get the shell shape of a particular cell.

virtual vtkm::IdComponent **GetNumberOfPointsInCell**(vtkm::Id id) const override

Get the number of points incident to a particular cell.

virtual void **GetCellPointIds**(vtkm::Id id, vtkm::Id \*ptids) const override

Get a list of points incident to a particular cell.

virtual std::shared\_ptr<CellSet> **NewInstance**() const override

Return a new *CellSet* that is the same derived class.

virtual void **DeepCopy**(const CellSet \*src) override

Copy the provided *CellSet* into this object.

The provided *CellSet* must be the same type as this one.

virtual void **PrintSummary**(std::ostream &out) const override

Print a summary of this cell set.

virtual void **ReleaseResourcesExecution**() override

Remove the *CellSet* from any devices.

Any memory used on a device to store this object will be deleted. However, the data will still remain on the host.

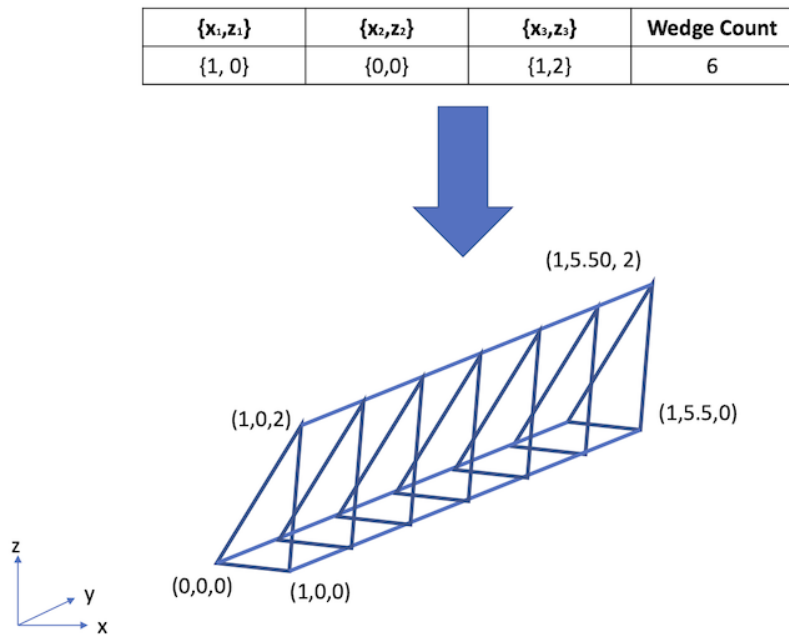


Figure 7: An example of an extruded wedge from XZ-plane coordinates. Six wedges are extracted from three XZ-plane points.

The extruded mesh is advantageous because it is represented on-the-fly as required, so no additional memory is required. In contrast other forms of cell sets, such as `vtkm::cont::CellSetExplicit`, need to be explicitly constructed by replicating the vertices and cells. Figure 7 shows an example of six wedges extruded from three 2-dimensional coordinates.



## 7.2.5 Unknown Cell Sets

Each of the aforementioned cell set types are represented by a different class. A `vtkm::cont::DataSet` object must hold one of these cell set objects that represent the cell structure. The actual object used is not determined until run time.

The `vtkm::cont::DataSet` object manages the cell set object with `vtkm::cont::UnknownCellSet`. When you call `vtkm::cont::DataSet::GetCellSet()`, it returns a `vtkm::cont::UnknownCellSet`.

The `vtkm::cont::UnknownCellSet` object provides mechanisms to query the cell set, identify its type, and cast it to one of the concrete `CellSet` types. See Chapter [ref{chap:UnknownCellSet}](#) for details on working with `vtkm::cont::UnknownCellSet`.

## 7.3 Fields

A field on a data set provides a value on every point in space on the mesh. Fields are often used to describe physical properties such as pressure, temperature, mass, velocity, and much more. Fields are represented in a VTK-m data set as an array where each value is associated with a particular element type of a mesh (such as points or cells). This association of field values to mesh elements and the structure of the cell set determines how the field is interpolated throughout the space of the mesh.

Fields are managed by the `vtkm::cont::Field` class.

class **Field**

A *Field* encapsulates an array on some piece of the mesh, such as the points, a cell set, a point logical dimension, or the whole mesh.

Subclassed by `vtkm::cont::CoordinateSystem`

Fields are identified by a simple name string.

```
inline const std::string &vtkm::cont::Field::GetName() const
```

The `vtkm::cont::Field` object internally holds a reference to an array in a type-agnostic way. Filters and other VTK-m units will determine the type of the array and pull it out of the `vtkm::cont::Field`.

```
const vtkm::cont::UnknownArrayHandle &vtkm::cont::Field::GetData() const
```

The field data is associated with a particular type of element of a mesh such as points, cells, or the whole mesh.

```
inline Association vtkm::cont::Field::GetAssociation() const
```

Associations are identified by the `vtkm::cont::Field::Association` enumeration.

```
enum class vtkm::cont::Field::Association
```

Identifies what elements of a data set a field is associated with.

The `Association` enum is used by `vtkm::cont::Field` to specify on what topological elements each item in the field is associated with.

*Values:*

enumerator **Any**

Any field regardless of the association.

This is used when choosing a `vtkm::cont::Field` that could be of any association. It is often used as the default if no association is given.

**enumerator `WholeDataSet`**

A “global” field that applies to the entirety of a `vtkm::cont::DataSet`.

Fields of this association often contain summary or annotation information. An example of a whole data set field could be the region that the mesh covers.

**enumerator `Points`**

A field that applies to points.

There is a separate field value attached to each point. Point fields usually represent samples of continuous data that can be reinterpolated through cells. Physical properties such as temperature, pressure, density, velocity, etc. are usually best represented in point fields. Data that deals with the points of the topology, such as displacement vectors, are also appropriate for point data.

**enumerator `Cells`**

A field that applies to cells.

There is a separate field value attached to each cell in a cell set. Cell fields usually represent values from an integration over the finite cells of the mesh. Integrated values like mass or volume are best represented in cell fields. Statistics about each cell like strain or cell quality are also appropriate for cell data.

**enumerator `Partitions`**

A field that applies to partitions.

This type of field is attached to a `vtkm::cont::PartitionedDataSet`. There is a separate field value attached to each partition. Identification or information about the arrangement of partitions such as hierarchy levels are usually best represented in partition fields.

**enumerator `Global`**

A field that applies to all partitions.

This type of field is attached to a `vtkm::cont::PartitionedDataSet`. It contains values that are “global” across all partitions and data therein.

The `vtkm::cont::Field` class also has several convenience methods for querying the association.

```
inline bool vtkm::cont::Field::IsPointField() const
```

```
inline bool vtkm::cont::Field::IsCellField() const
```

```
inline bool vtkm::cont::Field::IsWholeDataSetField() const
```

```
inline bool vtkm::cont::Field::IsPartitionsField() const
```

```
inline bool vtkm::cont::Field::IsGlobalField() const
```

`vtkm::cont::Field` has a convenience method named `vtkm::cont::Field::GetRange()` that finds the range of values stored in the field array.

```
const vtkm::cont::ArrayHandle<vtkm::Range> &vtkm::cont::Field::GetRange() const
```

Returns the range of each component in the field array.

The ranges of each component are returned in an `ArrayHandle` containing `vtkm::Range` values. So, for example, calling `GetRange` on a scalar field will return an `ArrayHandle` with exactly 1 entry in it. Calling `GetRange` on a field of 3D vectors will return an `ArrayHandle` with exactly 3 entries corresponding to each of the components in the range.

Details on how to get data from a `vtkm::cont::ArrayHandle` them is given in Chapter [ref{chap:AccessingAllocatingArrays}](#).

## 7.4 Coordinate Systems

A coordinate system determines the location of a mesh's elements in space. The spatial location is described by providing a 3D vector at each point that gives the coordinates there. The point coordinates can then be interpolated throughout the mesh.

class **CoordinateSystem** : public `vtkm::cont::Field`

Manages a coordinate system for a `DataSet`.

A coordinate system is really a field with a special meaning, so `CoordinateSystem` class inherits from the `Field` class. `CoordinateSystem` constrains the field to be associated with points and typically has 3D floating point vectors for values.

In addition to all the methods provided by the `vtkm::cont::Field` superclass, the `vtkm::cont::CoordinateSystem` also provides a `vtkm::cont::CoordinateSystem::GetBounds()` convenience method that returns a `vtkm::Bounds` object giving the spatial bounds of the coordinate system.

```
inline vtkm::Bounds vtkm::cont::CoordinateSystem::GetBounds() const
```

It is typical for a `vtkm::cont::DataSet` to have one coordinate system defined, but it is possible to define multiple coordinate systems. This is helpful when there are multiple ways to express coordinates. For example, positions in geographic may be expressed as Cartesian coordinates or as latitude-longitude coordinates. Both are valid and useful in different ways.

It is also valid to have a `vtkm::cont::DataSet` with no coordinate system. This is useful when the structure is not rooted in physical space. For example, if the cell set is representing a graph structure, there might not be any physical space that has meaning for the graph.

## 7.5 Partitioned Data Sets

class **PartitionedDataSet**

Comprises a set of `vtkm::cont::DataSet` objects.

### Iterators

`PartitionedDataSet` provides an iterator interface that allows you to iterate over the contained partitions using the `for (auto ds : pds)` syntax.

## Public Functions

**PartitionedDataSet**(const vtkm::cont::DataSet &ds)

Create a new *PartitionedDataSet* containing a single *DataSet* *ds*.

explicit **PartitionedDataSet**(const std::vector<vtkm::cont::DataSet> &partitions)

Create a new *PartitionedDataSet* with a *DataSet* vector *partitions*.

explicit **PartitionedDataSet**(vtkm::Id size)

Create a new *PartitionedDataSet* with the capacity set to be *size*.

vtkm::cont::Field **GetFieldFromPartition**(const std::string &field\_name, int partition\_index) const

Get the field *field\_name* from partition *partition\_index*.

vtkm::Id **GetNumberOfPartitions**() const

Get number of *DataSet* objects stored in this *PartitionedDataSet*.

vtkm::Id **GetGlobalNumberOfPartitions**() const

Get number of partitions across all MPI ranks.

<b>Warning:</b> This method requires global communication (MPI_Allreduce) if MPI is enabled.
--

const vtkm::cont::DataSet &**GetPartition**(vtkm::Id partId) const

Get the *DataSet* *partId*.

const std::vector<vtkm::cont::DataSet> &**GetPartitions**() const

Get an STL vector of all *DataSet* objects stored in *PartitionedDataSet*.

void **AppendPartition**(const vtkm::cont::DataSet &ds)

Add *DataSet* *ds* to the end of the list of partitions.

void **InsertPartition**(vtkm::Id index, const vtkm::cont::DataSet &ds)

Add *DataSet* *ds* to position *index* of the contained *DataSet* vector.

All partitions at or after this location are pushed back.

void **ReplacePartition**(vtkm::Id index, const vtkm::cont::DataSet &ds)

Replace the *index* positioned element of the contained *DataSet* vector with *ds*.

void **AppendPartitions**(const std::vector<vtkm::cont::DataSet> &partitions)

Append the *DataSet* vector *partitions* to the end of list of partitions.

This list can be provided as a `std::vector`, or it can be an initializer list (declared in { } curly braces).

inline vtkm::IdComponent **GetNumberOfFields**() const

Methods to Add and Get fields on a *PartitionedDataSet*.

inline void **AddField**(const Field &field)

Adds a field that is applied to the meta-partition structure.

The *field* must have a partition that applies across all partitions.

template<typename T, typename Storage>

inline void **AddGlobalField**(const std::string &fieldName, const vtkm::cont::ArrayHandle<T, Storage> &field)

Add a field with a global association.

template<typename T, typename Storage>

```
inline void AddPartitionsField(const std::string &fieldName, const vtkm::cont::ArrayHandle<T, Storage>
                                &field)
```

Add a field where each entry is associated with a whole partition.

```
inline vtkm::cont::Field &GetField(const std::string &name, vtkm::cont::Field::Association assoc =
                                   vtkm::cont::Field::Association::Any)
```

Get a field associated with the partitioned data structure.

The field is selected by name and, optionally, the association.

```
inline const vtkm::cont::Field &GetGlobalField(const std::string &name) const
```

Get a global field.

```
inline const vtkm::cont::Field &GetPartitionsField(const std::string &name) const
```

Get a field associated with the partitions.

```
inline bool HasField(const std::string &name, vtkm::cont::Field::Association assoc =
                     vtkm::cont::Field::Association::Any) const
```

Query whether the partitioned data set has the named field.

```
inline bool HasGlobalField(const std::string &name) const
```

Query whether the partitioned data set has the named global field.

```
inline bool HasPartitionsField(const std::string &name) const
```

Query whether the partitioned data set has the named partition field.

```
void CopyPartitions(const vtkm::cont::PartitionedDataSet &source)
```

Copies the partitions from the source. The fields on the *PartitionedDataSet* are not copied.

The following example creates a *vtkm::cont::PartitionedDataSet* containing two uniform grid data sets.

Example 8: Creating a *vtkm::cont::PartitionedDataSet*.

```
1 // Create two uniform data sets
2 vtkm::cont::DataSetBuilderUniform dataSetBuilder;
3
4 vtkm::cont::DataSet dataSet1 = dataSetBuilder.Create(vtkm::Id3(10, 10, 10));
5 vtkm::cont::DataSet dataSet2 = dataSetBuilder.Create(vtkm::Id3(30, 30, 30));
6
7 // Add the datasets to a multi block
8 vtkm::cont::PartitionedDataSet partitionedData;
9 partitionedData.AppendPartitions({ dataSet1, dataSet2 });
```

It is always possible to retrieve the independent blocks in a *vtkm::cont::PartitionedDataSet*, from which you can iterate and get information about the data. However, VTK-m provides several helper functions to collect metadata information about the collection as a whole.

```
vtkm::Bounds vtkm::cont::BoundsCompute(const vtkm::cont::DataSet &dataset, vtkm::Id
                                       coordinate_system_index = 0)
```

Functions to compute bounds for a single dataset or partition dataset.

These are utility functions that compute bounds for a single dataset or partitioned dataset. When VTK-m is operating in an distributed environment, these are bounds on the local process. To get global bounds across all ranks, use *vtkm::cont::BoundsGlobalCompute* instead.

Note that if the provided *CoordinateSystem* does not exists, empty bounds are returned. Likewise, for *PartitionedDataSet*, partitions without the chosen *CoordinateSystem* are skipped.

```
vtkm::Bounds vtkm::cont::BoundsCompute(const vtkm::cont::PartitionedDataSet &pds, vtkm::Id
                                         coordinate_system_index = 0)
```

```
vtkm::Bounds vtkm::cont::BoundsCompute(const vtkm::cont::DataSet &dataset, const std::string
                                         &coordinate_system_name)
```

```
vtkm::Bounds vtkm::cont::BoundsCompute(const vtkm::cont::PartitionedDataSet &pds, const std::string
                                         &coordinate_system_name)
```

```
vtkm::Bounds vtkm::cont::BoundsGlobalCompute(const vtkm::cont::DataSet &dataset, vtkm::Id
                                              coordinate_system_index = 0)
```

Functions to compute bounds for a single dataset or partitioned dataset globally.

These are utility functions that compute bounds for a single dataset or partitioned dataset globally i.e. across all ranks when operating in a distributed environment. When VTK-m not operating in an distributed environment, these behave same as `vtkm::cont::BoundsCompute`.

Note that if the provided *CoordinateSystem* does not exists, empty bounds are returned. Likewise, for *PartitionedDataSet*, partitions without the chosen *CoordinateSystem* are skipped.

```
vtkm::Bounds vtkm::cont::BoundsGlobalCompute(const vtkm::cont::PartitionedDataSet &pds, vtkm::Id
                                              coordinate_system_index = 0)
```

```
vtkm::Bounds vtkm::cont::BoundsGlobalCompute(const vtkm::cont::DataSet &dataset, const std::string
                                              &coordinate_system_name)
```

```
vtkm::Bounds vtkm::cont::BoundsGlobalCompute(const vtkm::cont::PartitionedDataSet &pds, const std::string
                                              &coordinate_system_name)
```

```
vtkm::cont::ArrayHandle<vtkm::Range> vtkm::cont::FieldRangeCompute(const vtkm::cont::DataSet &dataset,
                                                                    const std::string &name,
                                                                    vtkm::cont::Field::Association
                                                                    assoc =
                                                                    vtkm::cont::Field::Association::Any)
```

Compute ranges for fields in a *DataSet* or *PartitionedDataSet*.

These methods to compute ranges for fields in a single dataset or a partitioned dataset. When using VTK-m in a hybrid-parallel environment with distributed processing, this class uses ranges for locally available data alone. Use `FieldRangeGlobalCompute` to compute ranges globally across all ranks even in distributed mode. Returns the range for a field from a dataset. If the field is not present, an empty *ArrayHandle* will be returned.

```
vtkm::cont::ArrayHandle<vtkm::Range> vtkm::cont::FieldRangeCompute(const
                                                                    vtkm::cont::PartitionedDataSet
                                                                    &pds, const std::string &name,
                                                                    vtkm::cont::Field::Association
                                                                    assoc =
                                                                    vtkm::cont::Field::Association::Any)
```

Returns the range for a field from a *PartitionedDataSet*.

If the field is not present on any of the partitions, an empty *ArrayHandle* will be returned. If the field is present on some partitions, but not all, those partitions without the field are skipped.

The returned array handle will have as many values as the maximum number of components for the selected field across all partitions.

```

vtkm::cont::ArrayHandle<vtkm::Range> vtkm::cont::FieldRangeGlobalCompute(const vtkm::cont::DataSet
                                                                    &dataset, const std::string
                                                                    &name,
                                                                    vtkm::cont::Field::Association
                                                                    assoc =
                                                                    vtkm::cont::Field::Association::Any)

```

utility functions to compute global ranges for dataset fields.

These functions compute global ranges for fields in a single *DataSet* or a *PartitionedDataSet*. In non-distributed environments, this is exactly same as *FieldRangeCompute*. In distributed environments, however, the range is computed locally on each rank and then a reduce-all collective is performed to reduce the ranges on all ranks. Returns the range for a field from a dataset. If the field is not present, an empty *ArrayHandle* will be returned.

```

vtkm::cont::ArrayHandle<vtkm::Range> vtkm::cont::FieldRangeGlobalCompute(const
                                                                    vtkm::cont::PartitionedDataSet
                                                                    &pds, const std::string
                                                                    &name,
                                                                    vtkm::cont::Field::Association
                                                                    assoc =
                                                                    vtkm::cont::Field::Association::Any)

```

Returns the range for a field from a *PartitionedDataSet*.

If the field is not present on any of the partitions, an empty *ArrayHandle* will be returned. If the field is present on some partitions, but not all, those partitions without the field are skipped.

The returned array handle will have as many values as the maximum number of components for the selected field across all partitions.

The following example illustrates a spatial bounds query and a field range query on a *vtkm::cont::PartitionedDataSet*.

Example 9: Queries on a *vtkm::cont::PartitionedDataSet*.

```

1 // Get the bounds of a multi-block data set
2 vtkm::Bounds bounds = vtkm::cont::BoundsCompute(partitionedData);
3
4 // Get the overall min/max of a field named "cellvar"
5 vtkm::cont::ArrayHandle<vtkm::Range> cellvarRanges =
6   vtkm::cont::FieldRangeCompute(partitionedData, "cellvar");
7
8 // Assuming the "cellvar" field has scalar values, then cellvarRanges has one entry
9 vtkm::Range cellvarRange = cellvarRanges.ReadPortal().Get(0);

```

### Did You Know?

The aforementioned functions for querying a *vtkm::cont::PartitionedDataSet* object also work on *vtkm::cont::DataSet* objects. This is particularly useful with the *vtkm::cont::BoundsGlobalCompute()* and *vtkm::cont::FieldRangeGlobalCompute()* functions to manage distributed parallel objects.

Filters can be executed on *vtkm::cont::PartitionedDataSet* objects in a similar way they are executed on *vtkm::cont::DataSet* objects. In both cases, the *vtkm::cont::Filter::Execute()* method is called on the filter giving data object as an argument.

Example 10: Applying a filter to multi block data.

```
1 vtkm::filter::field_conversion::CellAverage cellAverage;  
2 cellAverage.SetActiveField("pointvar", vtkm::cont::Field::Association::Points);  
3  
4 vtkm::cont::PartitionedDataSet results = cellAverage.Execute(partitionedData);
```



Before VTK-m can be used to process data, data need to be loaded into the system. VTK-m comes with a basic file I/O package to get started developing very quickly. All the file I/O classes are declared under the `vtkm::io` namespace.

---

### Did You Know?

Files are just one of many ways to get data in and out of VTK-m. In later chapters we explore ways to define VTK-m data structures of increasing power and complexity. In particular, [Section 7.1 \(Building Data Sets\)](#) describes how to build VTK-m data set objects and [Section ref{sec:ArrayHandle:Adapting}](#) documents how to adapt data structures defined in other libraries to be used directly in VTK-m.

---

## 8.1 Readers

All reader classes provided by VTK-m are located in the `vtkm::io` namespace. The general interface for each reader class is to accept a filename in the constructor and to provide a `ReadDataSet` method to load the data from disk.

The data in the file are returned in a `vtkm::cont::DataSet` object as described in [Chapter 7 \(Data Sets\)](#), but it is sufficient to know that a `DataSet` can be passed among readers, writers, filters, and rendering units.

### 8.1.1 Legacy VTK File Reader

Legacy VTK files are a simple open format for storing visualization data. These files typically have a `.vtk` extension. Legacy VTK files are popular because they are simple to create and read and are consequently supported by a large number of tools. The format of legacy VTK files is well documented in *The VTK User's Guide* [as well as online](<https://examples.vtk.org/site/VTKFileFormats/>). Legacy VTK files can also be read and written with tools like ParaView and VisIt.

Legacy VTK files can be read using the `vtkm::io::VTKDataSetReader` class.

```
class VTKDataSetReader : public vtkm::io::VTKDataSetReaderBase
```

```
    Reads a legacy VTK file.
```

```
    By convention, legacy VTK files have a .vtk extension. This class should be constructed with a filename, and the data read with ReadDataSet.
```

## Public Functions

**VTKDataSetReader**(const std::string &fileName)

Construct a reader to load data from the given file.

Example 1: Reading a legacy VTK file.

```
1 #include <vtkm/io/VTKDataSetReader.h>
2
3 vtkm::cont::DataSet OpenDataFromVTKFile()
4 {
5     vtkm::io::VTKDataSetReader reader("data.vtk");
6
7     return reader.ReadDataSet();
8 }
```

### 8.1.2 Image Readers

VTK-m provides classes to read images from some standard image formats. These readers will store the data in a `vtkm::cont::DataSet` object with the colors stored as a named point field. The colors are read as 4-component RGBA vectors for each pixel. Each component in the pixel color is stored as a 32-bit float between 0 and 1.

Portable Network Graphics (PNG) files can be read using the `vtkm::io::ImageReaderPNG` class.

class **ImageReaderPNG** : public `vtkm::io::ImageReaderBase`

Reads images using the PNG format.

`ImageReaderPNG` is constructed with the name of the file to read. The data from the file is read by calling the `ReadDataSet` method.

`ImageReaderPNG` will automatically upsample/downsample read image data to a 16 bit RGB no matter how the image is compressed. It is up to the user to decide the pixel format for input PNGs

By default, the colors are stored in a field named “colors”, but the name of the field can optionally be changed using the `SetPointFieldName` method.

Example 2: Reading an image from a PNG file.

```
1 #include <vtkm/io/ImageReaderPNG.h>
2
3 vtkm::cont::DataSet OpenDataFromPNG()
4 {
5     vtkm::io::ImageReaderPNG imageReader("data.png");
6     imageReader.SetPointFieldName("pixel_colors");
7     return imageReader.ReadDataSet();
8 }
```

Portable anymap (PNM) files can be read using the `vtkm::io::ImageReaderPNM` class.

class **ImageReaderPNM** : public `vtkm::io::ImageReaderBase`

Reads images using the PNM format.

`ImageReaderPNM` is constructed with the name of the file to read. The data from the file is read by calling the `ReadDataSet` method.

Currently, *ImageReaderPNM* only supports files using the portable pixmap (PPM) format (with magic number `P6'). These files are most commonly stored with a *.ppm* extension although the *.pnm* extension is also valid. More details on the PNM format can be found here at <http://netpbm.sourceforge.net/doc/ppm.html>

By default, the colors are stored in a field named “colors”, but the name of the field can optionally be changed using the *SetPointFieldName* method.

Like for PNG files, a *vtkm::io::ImageReaderPNM* is constructed with the name of the file to read from.

Example 3: Reading an image from a PNM file.

```

1  #include <vtkm/io/ImageReaderPNM.h>
2
3  vtkm::cont::DataSet OpenDataFromPNM()
4  {
5      vtkm::io::ImageReaderPNM imageReader("data.ppm");
6      imageReader.SetPointFieldName("pixels");
7      return imageReader.ReadDataSet();
8  }
```

## 8.2 Writers

All writer classes provided by VTK-m are located in the *vtkm::io* namespace. The general interface for each writer class is to accept a filename in the constructor and to provide a *WriteDataSet* method to save data to the disk. The *WriteDataSet* method takes a *vtkm::cont::DataSet* object as an argument, which contains the data to write to the file.

### 8.2.1 Legacy VTK File Writer

Legacy VTK files can be written using the *vtkm::io::VTKDataSetWriter* class.

class **VTKDataSetWriter**

Reads a legacy VTK file.

By convention, legacy VTK files have a *.vtk* extension. This class should be constructed with a filename, and the data read with *ReadDataSet*.

#### Public Functions

**VTKDataSetWriter**(const std::string &fileName)

Construct a writer to save data to the given file.

void **WriteDataSet**(const vtkm::cont::DataSet &dataSet) const

Write data from the given DataSet object to the file specified in the constructor.

vtkm::io::FileType **GetFileType**() const

Get whether the file will be written in ASCII or binary format.

void **SetFileType**(vtkm::io::FileType type)

Set whether the file will be written in ASCII or binary format.

```
inline void SetFileTypeToAscii()
```

Set whether the file will be written in ASCII or binary format.

```
inline void SetFileTypeToBinary()
```

Set whether the file will be written in ASCII or binary format.

```
enum class vtkm::io::FileType
```

*Values:*

```
enumerator ASCII
```

```
enumerator BINARY
```

Example 4: Writing a legacy VTK file.

```
1 #include <vtkm/io/VTKDataSetWriter.h>
2
3 void SaveDataAsVTKFile(vtkm::cont::DataSet data)
4 {
5     vtkm::io::VTKDataSetWriter writer("data.vtk");
6
7     writer.WriteDataSet(data);
8 }
```

## 8.2.2 Image Writers

VTK-m provides classes to some standard image formats. These writers store data in a `vtkm::cont::DataSet`. The data must be a 2D structure with the colors stored in a point field. (See [Chapter 7 \(Data Sets\)](#) for details on `vtkm::cont::DataSet` objects.)

Portable Network Graphics (PNG) files can be written using the `vtkm::io::ImageWriterPNG` class.

```
class ImageWriterPNG : public vtkm::io::ImageWriterBase
```

Writes images using the PNG format.

`ImageWriterPNG` is constructed with the name of the file to write. The data is written to the file by calling the `WriteDataSet` method.

When writing files, `ImageReaderPNG` automatically compresses data to optimal sizes relative to the actual bit complexity of the provided image.

By default, PNG files are written as RGBA colors using 8-bits for each component. You can change the format written using the `vtkm::io::ImageWriterPNG::SetPixelDepth()` method. This takes an item in the `vtkm::io::ImageWriterPNG::PixelDepth` enumeration.

```
enum class vtkm::io::ImageWriterBase::PixelDepth
```

*Values:*

```
enumerator PIXEL_8
```

```
enumerator PIXEL_16
```

Example 5: Writing an image to a PNG file.

```

1  #include <vtkm/io/ImageWriterPNG.h>
2
3  void WriteToPNG(const vtkm::cont::DataSet& dataSet)
4  {
5      vtkm::io::ImageWriterPNG imageWriter("data.png");
6      imageWriter.SetPixelDepth(vtkm::io::ImageWriterPNG::PixelDepth::PIXEL_16);
7      imageWriter.WriteDataSet(dataSet);
8  }

```

Portable anymap (PNM) files can be written using the `vtkm::io::ImageWriterPNM` class.

class **ImageWriterPNM** : public `vtkm::io::ImageWriterBase`

Writes images using the PNM format.

`ImageWriterPNM` is constructed with the name of the file to write. The data is written to the file by calling the `WriteDataSet` method.

`ImageWriterPNM` writes images in PNM format (for magic number P6). These files are most commonly stored with a `.ppm` extension although the `.pnm` extension is also valid. More details on the PNM format can be found at <http://netpbm.sourceforge.net/doc/ppm.html>

## Public Functions

virtual void **Write**(`vtkm::Id` width, `vtkm::Id` height, const `ColorArrayType &pixels`) override

Attempts to write the `ImageDataSet` to a PNM file.

The `MaxColorValue` set in the file with either be selected from the stored `MaxColorValue` member variable, or from the templated type if `MaxColorValue` hasn't been set from a read file.

Example 6: Writing an image to a PNM file.

```

1  #include <vtkm/io/ImageWriterPNM.h>
2
3  void WriteToPNM(const vtkm::cont::DataSet& dataSet)
4  {
5      vtkm::io::ImageWriterPNM imageWriter("data.ppm");
6      imageWriter.SetPixelDepth(vtkm::io::ImageWriterPNM::PixelDepth::PIXEL_16);
7      imageWriter.WriteDataSet(dataSet);
8  }

```



## RUNNING FILTERS

Filters are functional units that take data as input and write new data as output. Filters operate on `vtkm::cont::DataSet` objects, which are described in [Chapter 7 \(Data Sets\)](#).

---

### Did You Know?

The structure of filters in VTK-m is significantly simpler than their counterparts in VTK. VTK filters are arranged in a dataflow network (a.k.a. a visualization pipeline) and execution management is handled automatically. In contrast, VTK-m filters are simple imperative units, which are simply called with input data and return output data.

---

VTK-m comes with several filters ready for use. This chapter gives an overview of how to run the filters. [Chapter 10 \(Provided Filters\)](#) describes the common filters provided by VTK-m. Later, [Part III \(Developing Algorithms\)](#) describes the necessary steps in creating new filters in VTK-m.

## 9.1 Basic Filter Operation

Different filters will be used in different ways, but the basic operation of all filters is to instantiate the filter class, set the state parameters on the filter object, and then call the filter's `vtkm::filter::Filter::Execute()` method. It takes a `vtkm::cont::DataSet` and returns a new `vtkm::cont::DataSet`, which contains the modified data.

```
vtkm::cont::DataSet vtkm::filter::Filter::Execute(const vtkm::cont::DataSet &input)
```

Executes the filter on the input and produces a result dataset.

On success, this the dataset produced. On error, `vtkm::cont::ErrorExecution` will be thrown.

The `vtkm::filter::Filter::Execute()` method can alternately take a `vtkm::cont::PartitionedDataSet` object, which is a composite of `vtkm::cont::DataSet` objects. In this case `vtkm::filter::Filter::Execute()` will return another `vtkm::cont::PartitionedDataSet` object.

```
vtkm::cont::PartitionedDataSet vtkm::filter::Filter::Execute(const vtkm::cont::PartitionedDataSet  
                                                             &input)
```

Executes the filter on the input PartitionedDataSet and produces a result PartitionedDataSet.

On success, this the dataset produced. On error, `vtkm::cont::ErrorExecution` will be thrown.

The following example provides a simple demonstration of using a filter. It specifically uses the point elevation filter to estimate the air pressure at each point based on its elevation.

Example 1: Using `vtkm::filter::field_transform::PointElevation` to estimate air pressure.

```

1 VTKM_CONT
2 vtkm::cont::DataSet ComputeAirPressure(vtkm::cont::DataSet dataSet)
3 {
4     vtkm::filter::field_transform::PointElevation elevationFilter;
5
6     // Use the elevation filter to estimate atmospheric pressure based on the
7     // height of the point coordinates. Atmospheric pressure is 101325 Pa at
8     // sea level and drops about 12 Pa per meter.
9     elevationFilter.SetLowPoint(0.0, 0.0, 0.0);
10    elevationFilter.SetHighPoint(0.0, 0.0, 2000.0);
11    elevationFilter.SetRange(101325.0, 77325.0);
12
13    elevationFilter.SetUseCoordinateSystemAsField(true);
14
15    elevationFilter.SetOutputFieldName("pressure");
16
17    vtkm::cont::DataSet result = elevationFilter.Execute(dataSet);
18
19    return result;
20 }
```

We see that this example follows the previously described procedure of constructing the filter (*line 4*), setting the state parameters (*lines 9 – 15*), and finally executing the filter on a `vtkm::cont::DataSet` (*line 17*).

Every `vtkm::cont::DataSet` object contains a list of *fields*, which describe some numerical value associated with different parts of the data set in space. Fields often represent physical properties such as temperature, pressure, or velocity. Fields are identified with string names. There are also special fields called coordinate systems that describe the location of points in space. Fields are mentioned here because they are often used as input data to the filter's operation and filters often generate new fields in the output. This is the case in [Example 1](#). In *line 13* the coordinate system is set as the input field and in *line 15* the name to use for the generated output field is selected.

## 9.2 Advanced Field Management

Most filters work with fields as inputs and outputs to their algorithms. Although in the previous discussions of the filters we have seen examples of specifying fields, these examples have been kept brief in the interest of clarity. In this section we revisit how filters manage fields and provide more detailed documentation of the controls.

Note that not all of the discussion in this section applies to all the filters provided by VTK-m. For example, not all filters have a specified input field. But where possible, the interface to the filter objects is kept consistent.



## 9.2.1 Input Fields

Filters that take one or more fields as input have a common set of methods to set the “active” fields to operate on. They might also have custom methods to ease setting the appropriate fields, but these are the base methods.

```
inline void vtkm::filter::Filter::SetActiveField(const std::string &name, vtkm::cont::Field::Association
                                              association = vtkm::cont::Field::Association::Any)
```

Specifies a field to operate on.

The number of input fields (or whether the filter operates on input fields at all) is specific to each particular filter.

```
inline void vtkm::filter::Filter::SetActiveField(vtkm::IdComponent index, const std::string &name,
                                              vtkm::cont::Field::Association association =
                                              vtkm::cont::Field::Association::Any)
```

Specifies a field to operate on.

The number of input fields (or whether the filter operates on input fields at all) is specific to each particular filter.

```
inline const std::string &vtkm::filter::Filter::GetActiveFieldName(vtkm::IdComponent index = 0) const
```

Specifies a field to operate on.

The number of input fields (or whether the filter operates on input fields at all) is specific to each particular filter.

```
inline vtkm::cont::Field::Association vtkm::filter::Filter::GetActiveFieldAssociation(vtkm::IdComponent
                                                                                      index = 0) const
```

Specifies a field to operate on.

The number of input fields (or whether the filter operates on input fields at all) is specific to each particular filter.

```
inline void vtkm::filter::Filter::SetActiveCoordinateSystem(vtkm::Id coord_idx)
```

Specifies the coordinate system index to make active to use when processing the input `vtkm::cont::DataSet`.

This is used primarily by the Filter to select the coordinate system to use as a field when `UseCoordinateSystemAsField` is true.

```
inline void vtkm::filter::Filter::SetActiveCoordinateSystem(vtkm::IdComponent index, vtkm::Id
                                                           coord_idx)
```

Specifies the coordinate system index to make active to use when processing the input `vtkm::cont::DataSet`.

This is used primarily by the Filter to select the coordinate system to use as a field when `UseCoordinateSystemAsField` is true.

```
inline vtkm::Id vtkm::filter::Filter::GetActiveCoordinateSystemIndex(vtkm::IdComponent index = 0)
                                                                    const
```

Specifies the coordinate system index to make active to use when processing the input `vtkm::cont::DataSet`.

This is used primarily by the Filter to select the coordinate system to use as a field when `UseCoordinateSystemAsField` is true.

```
inline void vtkm::filter::Filter::SetUseCoordinateSystemAsField(bool val)
```

Specifies whether to use point coordinates as the input field.

When true, the values for the active field are ignored and the active coordinate system is used instead.

```
inline void vtkm::filter::Filter::SetUseCoordinateSystemAsField(vtkm::IdComponent index, bool val)
```

Specifies whether to use point coordinates as the input field.

When true, the values for the active field are ignored and the active coordinate system is used instead.

`inline bool vtkm::filter::Filter::GetUseCoordinateSystemAsField(vtkm::IdComponent index = 0) const`  
Specifies whether to use point coordinates as the input field.

When true, the values for the active field are ignored and the active coordinate system is used instead.

`inline vtkm::IdComponent vtkm::filter::Filter::GetNumberOfActiveFields() const`  
Return the number of active fields currently set.

The general interface to `Filter` allows a user to set an arbitrary number of active fields (indexed 0 and on). This method returns the number of active fields that are set. Note that the filter implementation is free to ignore any active fields it does not support. Also note that an active field can be set to be either a named field or a coordinate system.

The `vtkm::filter::Filter::SetActiveField()` method takes an optional argument that specifies which topological elements the field is associated with (such as points or cells). The `vtkm::cont::Field::Association` enumeration is used to select the field association.

Example 2: Setting a field's active filter with an association.

```
1 filter.SetActiveField("pointvar", vtkm::cont::Field::Association::Points);
```

---

### Common Errors

It is possible to have two fields with the same name that are only differentiable by the association. That is, you could have a point field and a cell field with different data but the same name. Thus, it is best practice to specify the field association when possible. Likewise, it is poor practice to have two fields with the same name, particularly if the data are not equivalent in some way. It is often the case that fields are selected without an association.

It is also possible to set the active scalar field as a coordinate system of the data. A coordinate system essentially provides the spatial location of the points of the data and they have a special place in the `vtkm::cont::DataSet` structure. (See [Section 7.4 \(Coordinate Systems\)](#) for details on coordinate systems.) You can use a coordinate system as the active scalars by calling the `vtkm::filter::Filter::SetUseCoordinateSystemAsField()` method with a true flag. Since a `vtkm::cont::DataSet` can have multiple coordinate systems, you can select the desired coordinate system with `vtkm::filter::Filter::SetActiveCoordinateSystem()`. (By default, the first coordinate system, index 0, will be used.)

Example 3: Setting the active coordinate system.

```
1 filter.SetUseCoordinateSystemAsField(true);  
2 filter.SetActiveCoordinateSystem(1);
```

## 9.2.2 Passing Fields from Input to Output

After a filter successfully executes and returns a new data set, fields are mapped from input to output. Depending on what operation the filter does, this could be a simple shallow copy of an array, or it could be a computed operation. By default, the filter will automatically pass all fields from input to output (performing whatever transformations are necessary). You can control which fields are passed (and equivalently which are not) with the `vtkm::filter::Filter::SetFieldsToPass()` methods.

`void vtkm::filter::Filter::SetFieldsToPass(vtkm::filter::FieldSelection &&fieldsToPass)`

Specify which fields get passed from input to output.

After a filter successfully executes and returns a new data set, fields are mapped from input to output. Depending on what operation the filter does, this could be a simple shallow copy of an array, or it could be a computed operation. You can control which fields are passed (and equivalently which are not) with this parameter.

By default, all fields are passed during execution.

```
inline const vtkm::filter::FieldSelection &vtkm::filter::Filter::GetFieldsToPass() const
```

Specify which fields get passed from input to output.

After a filter successfully executes and returns a new data set, fields are mapped from input to output. Depending on what operation the filter does, this could be a simple shallow copy of an array, or it could be a computed operation. You can control which fields are passed (and equivalently which are not) with this parameter.

By default, all fields are passed during execution.

```
inline vtkm::filter::FieldSelection &vtkm::filter::Filter::GetFieldsToPass()
```

Specify which fields get passed from input to output.

After a filter successfully executes and returns a new data set, fields are mapped from input to output. Depending on what operation the filter does, this could be a simple shallow copy of an array, or it could be a computed operation. You can control which fields are passed (and equivalently which are not) with this parameter.

By default, all fields are passed during execution.

There are multiple ways to use `vtkm::filter::Filter::SetFieldsToPass()` to control what fields are passed. If you want to turn off all fields so that none are passed, call `vtkm::filter::Filter::SetFieldsToPass()` with `vtkm::filter::FieldSelection::Mode::None`.

Example 4: Turning off the passing of all fields when executing a filter.

```
filter.SetFieldsToPass(vtkm::filter::FieldSelection::Mode::None);
```

If you want to pass one specific field, you can pass that field's name to `vtkm::filter::Filter::SetFieldsToPass()`.

```
inline void vtkm::filter::Filter::SetFieldsToPass(const std::string &fieldname,
                                                  vtkm::filter::FieldSelection::Mode mode)
```

Specify which fields get passed from input to output.

After a filter successfully executes and returns a new data set, fields are mapped from input to output. Depending on what operation the filter does, this could be a simple shallow copy of an array, or it could be a computed operation. You can control which fields are passed (and equivalently which are not) with this parameter.

By default, all fields are passed during execution.

```
void vtkm::filter::Filter::SetFieldsToPass(const std::string &fieldname, vtkm::cont::Field::Association
                                          association, vtkm::filter::FieldSelection::Mode mode =
                                          vtkm::filter::FieldSelection::Mode::Select)
```

Specify which fields get passed from input to output.

After a filter successfully executes and returns a new data set, fields are mapped from input to output. Depending on what operation the filter does, this could be a simple shallow copy of an array, or it could be a computed operation. You can control which fields are passed (and equivalently which are not) with this parameter.

By default, all fields are passed during execution.

Example 5: Setting one field to pass by name.

```
filter.SetFieldsToPass("pointvar");
```

Or you can provide a list of fields to pass by giving `vtkm::filter::Filter::SetFieldsToPass()` an initializer list of names.

```
void vtkm::filter::Filter::SetFieldsToPass(std::initializer_list<std::string> fields,
                                           vtkm::filter::FieldSelection::Mode mode =
                                           vtkm::filter::FieldSelection::Mode::Select)
```

Specify which fields get passed from input to output.

After a filter successfully executes and returns a new data set, fields are mapped from input to output. Depending on what operation the filter does, this could be a simple shallow copy of an array, or it could be a computed operation. You can control which fields are passed (and equivalently which are not) with this parameter.

By default, all fields are passed during execution.

Example 6: Using a list of fields for a filter to pass.

```
1 filter.SetFieldsToPass({ "pointvar", "cellvar" });
```

If you want to instead select a list of fields to *not* pass, you can add `vtkm::filter::FieldSelection::Mode::Exclude` as an argument to `vtkm::filter::Filter::SetFieldsToPass()`.

Example 7: Excluding a list of fields for a filter to pass.

```
1 filter.SetFieldsToPass({ "pointvar", "cellvar" },
2                       vtkm::filter::FieldSelection::Mode::Exclude);
```

Ultimately, `vtkm::filter::Filter::SetFieldsToPass()` takes a `vtkm::filter::FieldSelection` object. You can create one directly to select (or exclude) specific fields and their associations.

#### class **FieldSelection**

A `FieldSelection` stores information about fields to map for input dataset to output when a filter is executed.

A `FieldSelection` object is passed to `vtkm::filter::Filter::Execute` to execute the filter and map selected fields. It is possible to easily construct `FieldSelection` that selects all or none of the input fields.

#### Unnamed Group

```
inline void AddField(const vtkm::cont::Field &inputField)
```

Add fields to select or exclude. If no mode is specified, then the mode will follow that of `GetMode()`.

```
inline void AddField(const vtkm::cont::Field &inputField, Mode mode)
```

Add fields to select or exclude. If no mode is specified, then the mode will follow that of `GetMode()`.

```
inline void AddField(const std::string &fieldName, vtkm::cont::Field::Association association =
                  vtkm::cont::Field::Association::Any)
```

Add fields to select or exclude. If no mode is specified, then the mode will follow that of `GetMode()`.

```
inline void AddField(const std::string &fieldName, Mode mode)
```

Add fields to select or exclude. If no mode is specified, then the mode will follow that of `GetMode()`.

```
void AddField(const std::string &fieldName, vtkm::cont::Field::Association association, Mode mode)
```

Add fields to select or exclude. If no mode is specified, then the mode will follow that of `GetMode()`.

## Unnamed Group

inline Mode **GetFieldMode**(const vtkm::cont::Field &inputField) const

Returns the mode for a particular field. If the field has been added with **AddField** (or another means), then this will return **Select** or **Exclude**. If the field has not been added, **None** will be returned.

Mode **GetFieldMode**(const std::string &fieldName, vtkm::cont::Field::Association association = vtkm::cont::Field::Association::Any) const

Returns the mode for a particular field. If the field has been added with **AddField** (or another means), then this will return **Select** or **Exclude**. If the field has not been added, **None** will be returned.

## Public Functions

**FieldSelection**(const std::string &field, Mode mode = Mode::Select)

Use this constructor to create a field selection given a single field name.

```
FieldSelection("field_name");
```

**FieldSelection**(const char \*field, Mode mode = Mode::Select)

Use this constructor to create a field selection given a single field name.

```
FieldSelection("field_name");
```

**FieldSelection**(const std::string &field, vtkm::cont::Field::Association association, Mode mode = Mode::Select)

Use this constructor to create a field selection given a single name and association.

```
FieldSelection("field_name", vtkm::cont::Field::Association::Points)
```

```
{cpp}
```

**FieldSelection**(std::initializer\_list<std::string> fields, Mode mode = Mode::Select)

Use this constructor to create a field selection given the field names.

```
FieldSelection({"field_one", "field_two"});
```

**FieldSelection**(std::initializer\_list<std::pair<std::string, vtkm::cont::Field::Association>> fields, Mode mode = Mode::Select)

Use this constructor create a field selection given the field names and associations e.g.

```
using pair_type = std::pair<std::string, vtkm::cont::Field::Association>;
FieldSelection({
    pair_type{"field_one", vtkm::cont::Field::Association::Points},
    pair_type{"field_two", vtkm::cont::Field::Association::Cells} });
```

**FieldSelection**(std::initializer\_list<vtkm::Pair<std::string, vtkm::cont::Field::Association>> fields, Mode mode = Mode::Select)

Use this constructor create a field selection given the field names and associations e.g.

```
using pair_type = vtkm::Pair<std::string, vtkm::cont::Field::Association>;
FieldSelection({
    pair_type{"field_one", vtkm::cont::Field::Association::Points},
    pair_type{"field_two", vtkm::cont::Field::Association::Cells} });
```

inline bool **IsFieldSelected**(const vtkm::cont::Field &inputField) const

Returns true if the input field should be mapped to the output dataset.

inline bool **HasField**(const vtkm::cont::Field &inputField) const

Returns true if the input field has been added to this selection.

Note that depending on the mode of this selection, the result of `HasField` is not necessarily the same as `IsFieldSelected`. (If the mode is `MODE_SELECT`, then the result of the two will be the same.)

void **ClearFields**()

Clear all fields added using `AddField`.

Mode **GetMode**() const

Gets the mode of the field selection.

If `Select` mode is on, then only fields that have a `Select` mode are considered as selected. (All others are considered unselected.) Calling `AddField` in this mode will mark it as `Select`. If `Exclude` mode is on, then all fields are considered selected except those fields with an `Exclude` mode. Calling `AddField` in this mode will mark it as `Exclude`.

void **SetMode**(Mode val)

Sets the mode of the field selection.

If `Select` mode is on, then only fields that have a `Select` mode are considered as selected. (All others are considered unselected.) Calling `AddField` in this mode will mark it as `Select`. If `Exclude` mode is on, then all fields are considered selected except those fields with an `Exclude` mode. Calling `AddField` in this mode will mark it as `Exclude`.

If the mode is set to `None`, then the field modes are cleared and the overall mode is set to `Select` (meaning none of the fields are initially selected). If the mode is set to `All`, then the field modes are cleared and the overall mode is set to `Exclude` (meaning all of the fields are initially selected).

Example 8: Using `vtkm::filter::FieldSelection` to select cells to pass.

```
1 vtkm::filter::FieldSelection fieldSelection;
2 fieldSelection.AddField("scalars");
3 fieldSelection.AddField("cellvar", vtkm::cont::Field::Association::Cells);
4
5 filter.SetFieldsToPass(fieldSelection);
```

It is also possible to specify field attributions directly to `vtkm::filter::Filter::SetFieldsToPass()`. If you only have one field, you can just specify both the name and attribution. If you have multiple fields, you can provide an initializer list of `std::pair` or `vtkm::Pair` containing a `std::string` and a `vtkm::cont::Field::Association`. In either case, you can add an optional last argument of `vtkm::filter::FieldSelection::Mode::Exclude` to exclude the specified filters instead of selecting them.

```
void vtkm::filter::Filter::SetFieldsToPass(std::initializer_list<std::pair<std::string,
    vtkm::cont::Field::Association>> fields,
    vtkm::filter::FieldSelection::Mode mode =
    vtkm::filter::FieldSelection::Mode::Select)
```

Specify which fields get passed from input to output.

After a filter successfully executes and returns a new data set, fields are mapped from input to output. Depending on what operation the filter does, this could be a simple shallow copy of an array, or it could be a computed operation. You can control which fields are passed (and equivalently which are not) with this parameter.

By default, all fields are passed during execution.

Example 9: Selecting one field and its association for a filter to pass.

```
1 filter.SetFieldsToPass("pointvar", vtkm::cont::Field::Association::Points);
```

Example 10: Selecting a list of fields and their associations for a filter to pass.

```
1 filter.SetFieldsToPass(  
2     { vtkm::make_Pair("pointvar", vtkm::cont::Field::Association::Points),  
3       vtkm::make_Pair("cellvar", vtkm::cont::Field::Association::Cells),  
4       vtkm::make_Pair("scalars", vtkm::cont::Field::Association::Any) });
```

Note that coordinate systems in a `vtkm::cont::DataSet` are simply links to point fields, and by default filters will pass coordinate systems regardless of the field selection flags. To prevent a filter from passing a coordinate system if its associated field is not selected, use the `vtkm::filter::Filter::SetPassCoordinateSystems()` method.

`inline void vtkm::filter::Filter::SetPassCoordinateSystems(bool flag)`

Specify whether to always pass coordinate systems.

`vtkm::cont::CoordinateSystems` in a `DataSet` are really just point fields marked as being a coordinate system. Thus, a coordinate system is passed if and only if the associated field is passed.

By default, the filter will pass all fields associated with a coordinate system regardless of the `FieldsToPass` marks the field as passing. If this option is set to `false`, then coordinate systems will only be passed if it is marked so by `FieldsToPass`.

`inline bool vtkm::filter::Filter::GetPassCoordinateSystems() const`

Specify whether to always pass coordinate systems.

`vtkm::cont::CoordinateSystems` in a `DataSet` are really just point fields marked as being a coordinate system. Thus, a coordinate system is passed if and only if the associated field is passed.

By default, the filter will pass all fields associated with a coordinate system regardless of the `FieldsToPass` marks the field as passing. If this option is set to `false`, then coordinate systems will only be passed if it is marked so by `FieldsToPass`.

Example 11: Turning off the automatic selection of fields associated with a `vtkm::cont::DataSet`'s coordinate system.

```
filter.SetPassCoordinateSystems(false);
```

### 9.2.3 Output Field Names

Many filters will create fields of data. A common way to set the name of the output field is to use the `vtkm::filter::Filter::SetOutputFieldName()` method.

```
inline void vtkm::filter::Filter::SetOutputFieldName(const std::string &name)
```

Specifies the name of the output field generated.

Not all filters create an output field.

```
inline const std::string &vtkm::filter::Filter::GetOutputFieldName() const
```

Specifies the name of the output field generated.

Not all filters create an output field.

Most filters will have a default name to use for its generated fields. It is also common for filters to provide convenience methods to name the output fields.



## PROVIDED FILTERS

VTK-m comes with the implementation of many filters. Filters in VTK-m are divided into a collection of modules, each with its own namespace and library. This section is organized by each filter module, each of which contains one or more filters that are related to each other.

Note that this is not an exhaustive list of filters available in VTK-m. More can be found in the namespaces under `vtkm::filter` (and likewise the subdirectories under `vtkm/filter` in the VTK-m source).

### 10.1 Cleaning Grids

The `vtkm::filter::clean_grid` module contains filters that resolve issues with mesh structure. This could include finding and merging coincident points, removing degenerate cells, or converting the grid to a known type.

#### 10.1.1 Clean Grid

`vtkm::filter::clean_grid::CleanGrid` is a filter that converts a cell set to an explicit representation and potentially removes redundant or unused data. It does this by iterating over all cells in the data set, and for each one creating the explicit cell representation that is stored in the output. (Explicit cell sets are described in [Section 7.2.2 \(Explicit Cell Sets\)](#).) One benefit of using `vtkm::filter::clean_grid::CleanGrid` is that it can optionally remove unused points and combine coincident points. Another benefit is that the resulting cell set will be of a known specific type.

---

#### Common Errors

The result of `vtkm::filter::clean_grid::CleanGrid` is not necessarily smaller, memory-wise, than its input. For example, “cleaning” a data set with a structured topology will actually result in a data set that requires much more memory to store an explicit topology.

---

class **CleanGrid** : public `vtkm::filter::Filter`

Clean a mesh to an unstructured grid.

This filter converts the cells of its input to an explicit representation and potentially removes redundant or unused data. The newly constructed data set will have the same cells as the input and the topology will be stored in a `vtkm::cont::CellSetExplicit<>`. The filter will also optionally remove all unused points.

Note that the result of `CleanGrid` is not necessarily smaller than the input. For example, “cleaning” a data set with a `vtkm::cont::CellSetStructured` topology will actually result in a much larger data set.

`CleanGrid` can optionally merge close points. The closeness of points is determined by the coordinate system. If there are multiple coordinate systems, the desired coordinate system can be selected with the `SetActiveCoordinateSystem()` method.

## Public Functions

inline bool **GetCompactPointFields()** const

When the CompactPointFields flag is true, the filter will identify and remove any points that are not used by the topology.

This is on by default.

inline void **SetCompactPointFields**(bool flag)

When the CompactPointFields flag is true, the filter will identify and remove any points that are not used by the topology.

This is on by default.

inline bool **GetMergePoints()** const

When the MergePoints flag is true, the filter will identify any coincident points and merge them together.

The distance two points can be to considered coincident is set with the tolerance flags. This is on by default.

inline void **SetMergePoints**(bool flag)

When the MergePoints flag is true, the filter will identify any coincident points and merge them together.

The distance two points can be to considered coincident is set with the tolerance flags. This is on by default.

inline vtkm::Float64 **GetTolerance()** const

Defines the tolerance used when determining whether two points are considered coincident.

Because floating point parameters have limited precision, point coordinates that are essentially the same might not be bit-wise exactly the same. Thus, the *CleanGrid* filter has the ability to find and merge points that are close but perhaps not exact. If the ToleranceIsAbsolute flag is false (the default), then this tolerance is scaled by the diagonal of the points.

inline void **SetTolerance**(vtkm::Float64 tolerance)

Defines the tolerance used when determining whether two points are considered coincident.

Because floating point parameters have limited precision, point coordinates that are essentially the same might not be bit-wise exactly the same. Thus, the *CleanGrid* filter has the ability to find and merge points that are close but perhaps not exact. If the ToleranceIsAbsolute flag is false (the default), then this tolerance is scaled by the diagonal of the points.

inline bool **GetToleranceIsAbsolute()** const

When ToleranceIsAbsolute is false (the default) then the tolerance is scaled by the diagonal of the bounds of the dataset.

If true, then the tolerance is taken as the actual distance to use.

inline void **SetToleranceIsAbsolute**(bool flag)

When ToleranceIsAbsolute is false (the default) then the tolerance is scaled by the diagonal of the bounds of the dataset.

If true, then the tolerance is taken as the actual distance to use.

inline bool **GetRemoveDegenerateCells()** const

When RemoveDegenerateCells is true (the default), then *CleanGrid* will look for repeated points in cells and, if the repeated points cause the cell to drop dimensionality, the cell is removed.

This is particularly useful when point merging is on as this operation can create degenerate cells.

```
inline void SetRemoveDegenerateCells(bool flag)
```

When `RemoveDegenerateCells` is true (the default), then *CleanGrid* will look for repeated points in cells and, if the repeated points cause the cell to drop dimensionality, the cell is removed.

This is particularly useful when point merging is on as this operation can create degenerate cells.

```
inline bool GetFastMerge() const
```

When `FastMerge` is true (the default), some corners are cut when computing coincident points.

The point merge will go faster but the tolerance will not be strictly followed.

```
inline void SetFastMerge(bool flag)
```

When `FastMerge` is true (the default), some corners are cut when computing coincident points.

The point merge will go faster but the tolerance will not be strictly followed.

## 10.2 Connected Components

Connected components in a mesh are groups of mesh elements that are connected together in some way. For example, if two cells are neighbors, then they are in the same component. Likewise, a cell is also in the same component as its neighbor's neighbors as well as their neighbors and so on. Connected components help identify when features in a simulation fragment or meld.

The `vtkm::filter::connected_components` module contains filters that find groups of cells that are connected. There are different ways to define what it means to be connected. One way is to use the topological connections of the cells. That is, two cells that share a point, edge, or face are connected. Another way is to use a field that classifies each cell, and cells are only connected if they have the same classification.

### 10.2.1 Cell Connectivity

The `vtkm::filter::connected_components::CellSetConnectivity` filter finds groups of cells that are connected together through their topology.

```
class CellSetConnectivity : public vtkm::filter::Filter
```

Finds and labels groups of cells that are connected together through their topology.

Two cells are considered connected if they share an edge. *CellSetConnectivity* identifies some number of components and assigns each component a unique integer.

The result of the filter is a cell field of type `vtkm::Id` with the default name of “component” (which can be changed with the `SetOutputFieldName` method). Each entry in the cell field will be a number that identifies to which component the cell belongs.

### 10.2.2 Classification Field on Image Data

The `vtkm::filter::connected_components::ImageConnectivity` filter finds groups of points that have the same field value and are connected together through their topology.

```
class ImageConnectivity : public vtkm::filter::Filter
```

## 10.3 Contouring

The `vtkm::filter::contour` module contains filters that extract regions that match some field or spatial criteria. Unlike [entity extraction filters](#) (Section 10.5), the geometry will be clipped or sliced to extract the exact matching region. (In contrast, entity extraction filters will pull unmodified points, edges, faces, or cells from the input.)

### 10.3.1 Contour

*Contouring* is one of the most fundamental filters in scientific visualization. A contour is the locus where a field is equal to a particular value. A topographic map showing curves of various elevations often used when hiking in hilly regions is an example of contours of an elevation field in 2 dimensions. Extended to 3 dimensions, a contour gives a surface. Thus, a contour is often called an *isosurface*. The contouring/isosurface algorithm is implemented by `vtkm::filter::contour::Contour`.

class **Contour** : public `vtkm::filter::contour::AbstractContour`

Generate contours or isosurfaces from a region of space.

*Contour* takes as input a mesh, often a volume, and generates on output one or more surfaces where a field equals a specified value.

This filter implements multiple algorithms for contouring, and the best algorithm will be selected based on the type of the input.

The scalar field to extract the contour from is selected with the `SetActiveField()` and related methods.

Subclassed by `vtkm::filter::contour::Slice`, `vtkm::filter::contour::SliceMultiple`

`vtkm::filter::contour::Contour` also inherits the following methods.

inline void `vtkm::filter::contour::AbstractContour::SetIsoValue`(`vtkm::Float64` v)

Set a field value on which to extract a contour.

This form of the method is usually used when only one contour is being extracted.

inline void `vtkm::filter::contour::AbstractContour::SetIsoValue`(`vtkm::Id` index, `vtkm::Float64` v)

Set a field value on which to extract a contour.

This form is used to specify multiple contours. The method is called multiple times with different *index* parameters.

inline void `vtkm::filter::contour::AbstractContour::SetIsoValues`(const std::vector<`vtkm::Float64`> &values)

Set multiple iso values at once.

The iso values can be specified as either a `std::vector` or an initializer list. So, both

```
std::vector<vtkm::Float64> isovalues = { 0.2, 0.5, 0.7 };
contour.SetIsoValues(isovalues);
```

and

```
contour.SetIsoValues({ 0.2, 0.5, 0.7 });
```

work.

inline `vtkm::Float64` `vtkm::filter::contour::AbstractContour::GetIsoValue`(`vtkm::Id` index = 0) const

Return a value used to contour the mesh.

```
inline void vtkm::filter::contour::AbstractContour::SetGenerateNormals(bool flag)
```

Set whether normals should be generated.

Normals are used in shading calculations during rendering and can make the surface appear more smooth.

Off by default.

```
inline bool vtkm::filter::contour::AbstractContour::GetGenerateNormals() const
```

Get whether normals should be generated.

```
inline void vtkm::filter::contour::AbstractContour::SetComputeFastNormals(bool flag)
```

Set whether the fast path should be used for normals computation.

When this flag is off (the default), the generated normals are based on the gradient of the field being contoured and can be quite expensive to compute. When the flag is on, a faster method that computes the normals based on the faces of the isosurface mesh is used, but the normals do not look as good as the gradient based normals.

This flag has no effect if SetGenerateNormals is false.

```
inline bool vtkm::filter::contour::AbstractContour::GetComputeFastNormals() const
```

Get whether the fast path should be used for normals computation.

```
inline void vtkm::filter::contour::AbstractContour::SetNormalArrayName(const std::string &name)
```

Set the name of the field for the generated normals.

```
inline const std::string &vtkm::filter::contour::AbstractContour::GetNormalArrayName() const
```

Get the name of the field for the generated normals.

```
inline void vtkm::filter::contour::AbstractContour::SetMergeDuplicatePoints(bool on)
```

Set whether the points generated should be unique for every triangle or will duplicate points be merged together.

Duplicate points are identified by the unique edge it was generated from.

Because the contour filter (like all filters in VTK-m) runs in parallel, parallel threads can (and often do) create duplicate versions of points. When this flag is set to true, a secondary operation will find all duplicated points and combine them together. If false, points will be duplicated. In addition to requiring more storage, duplicated points mean that triangles next to each other will not be considered adjacent to subsequent filters.

```
inline bool vtkm::filter::contour::AbstractContour::GetMergeDuplicatePoints()
```

Get whether the points generated should be unique for every triangle or will duplicate points be merged together.

Example 1: Using `vtkm::filter::contour::Contour`.

```
1 vtkm::filter::contour::Contour contour;  
2  
3 contour.SetActiveField("pointvar");  
4 contour.SetIsoValue(10.0);  
5  
6 vtkm::cont::DataSet isosurface = contour.Execute(inData);
```

### 10.3.2 Slice

A slice operation intersects a mesh with a surface. The `vtkm::filter::contour::Slice` filter uses a `vtkm::ImplicitFunctionGeneral` to specify an implicit surface to slice on. A plane is a common thing to slice on, but other surfaces are available. See [Chapter 15 \(Implicit Functions\)](#) for information on implicit functions.

class **Slice** : public `vtkm::filter::contour::Contour`

Intersect a mesh with an implicit surface.

This filter accepts a `vtkm::ImplicitFunction` that defines the surface to slice on. A `vtkm::Plane` is a common function to use that cuts the mesh along a plane.

#### Public Functions

inline void **SetImplicitFunction**(const `vtkm::ImplicitFunctionGeneral` &func)

Set the implicit function that is used to perform the slicing.

Only a limited number of implicit functions are supported. See `vtkm::ImplicitFunctionGeneral` for information on which ones.

inline const `vtkm::ImplicitFunctionGeneral` &**GetImplicitFunction**() const

Get the implicit function that us used to perform the slicing.

The `vtkm::filter::contour::Slice` filter inherits from the `vtkm::filter::contour::Contour`, uses its implementation to extract the slices, and several of the inherited methods are useful including `vtkm::filter::contour::AbstractContour::SetGenerateNormals()`, `vtkm::filter::contour::AbstractContour::GetGenerateNormals()`, `vtkm::filter::contour::AbstractContour::SetComputeFastNormals()`, `vtkm::filter::contour::AbstractContour::GetComputeFastNormals()`, `vtkm::filter::contour::AbstractContour::SetNormalArrayName()`, `vtkm::filter::contour::AbstractContour::GetNormalArrayName()`, `vtkm::filter::contour::AbstractContour::SetMergeDuplicatePoints()`, `vtkm::filter::contour::AbstractContour::GetMergeDuplicatePoints()`, `vtkm::filter::Field::SetActiveCoordinateSystem()`, and `vtkm::filter::Field::GetActiveCoordinateSystemIndex()`.

### 10.3.3 Clip with Field

Clipping is an operation that removes regions from the data set based on a user-provided value or function. The `vtkm::filter::contour::ClipWithField` filter takes a clip value as an argument and removes regions where a named scalar field is below (or above) that value. (A companion filter that discards a region of the data based on an implicit function is described later.)

The result of `vtkm::filter::contour::ClipWithField` is a volume. If a cell has field values at its vertices that are all below the specified value, then it will be discarded entirely. Likewise, if a cell has field values at its vertices that are all above the specified value, then it will be retained in its entirety. If a cell has some vertices with field values

below the specified value and some above, then the cell will be split into the portions above the value (which will be retained) and the portions below the value (which will be discarded).

This operation is sometimes called an *isovolume* because it extracts the volume of a mesh that is inside the iso-region of a scalar. This is in contrast to an *isosurface*, which extracts only the surface of that iso-value. That said, a more appropriate name is *interval volume* as the volume is defined by a range of values, not a single “iso” value.

`vtkm::filter::contour::ClipWithField` is also similar to a threshold operation, which extracts cells based on the value of field. The difference is that threshold will either keep or remove entire cells based on the field values whereas clip with carve cells that straddle the valid regions. See [Section 10.5.6 \(Threshold\)](#) for information on threshold extraction.

```
class ClipWithField : public vtkm::filter::Filter
```

Clip a dataset using a field.

Clip a dataset using a given field value. All points that are less than that value are considered outside, and will be discarded. All points that are greater are kept.

To select the scalar field, use the `SetActiveField()` and related methods.

### Public Functions

```
inline void SetClipValue(vtkm::Float64 value)
```

Specifies the field value for the clip operation.

Regions where the active field is less than this value are clipped away from each input cell.

```
inline void SetInvertClip(bool invert)
```

Specifies if the result for the clip filter should be inverted.

If set to false (the default), regions where the active field is less than the specified clip value are removed.

If set to true, regions where the active field is more than the specified clip value are removed.

```
inline vtkm::Float64 GetClipValue() const
```

Specifies the field value for the clip operation.

```
inline bool GetInvertClip() const
```

Specifies if the result for the clip filter should be inverted.

Example 2: Using `vtkm::filter::contour::ClipWithField`.

```
1 // Create an instance of a clip filter that discards all regions with scalar
2 // value less than 25.
3 vtkm::filter::contour::ClipWithField clip;
4 clip.SetClipValue(25.0);
5 clip.SetActiveField("pointvar");
6
7 // Execute the clip filter
8 vtkm::cont::DataSet outData = clip.Execute(inData);
```

### 10.3.4 Clip with Implicit Function

The `vtkm::filter::contour::ClipWithImplicitFunction` function takes an implicit function and removes all parts of the data that are inside (or outside) that function. See [Chapter 15 \(Implicit Functions\)](#) for more detail on how implicit functions are represented in VTK-m. A companion filter that discards a region of the data based on the value of a scalar field is described in [Section 10.5.2 \(Extract Geometry\)](#).

The result of `vtkm::filter::contour::ClipWithImplicitFunction` is a volume. If a cell has its vertices positioned all outside the implicit function, then it will be discarded entirely. Likewise, if a cell its vertices all inside the implicit function, then it will be retained in its entirety. If a cell has some vertices inside the implicit function and some outside, then the cell will be split into the portions inside (which will be retained) and the portions outside (which will be discarded).

class **ClipWithImplicitFunction** : public `vtkm::filter::Filter`

Clip a dataset using an implicit function.

Clip a dataset using a given implicit function value, such as `vtkm::Sphere` or `vtkm::Frustum`. The implicit function uses the point coordinates as its values. If there is more than one coordinate system in the input `vtkm::cont::DataSet`, it can be selected with `SetActiveCoordinateSystem()`.

#### Public Functions

inline void **SetImplicitFunction**(const `vtkm::ImplicitFunctionGeneral` &func)

Specifies the implicit function to be used to perform the clip operation.

Only a limited number of implicit functions are supported. See `vtkm::ImplicitFunctionGeneral` for information on which ones.

inline void **SetInvertClip**(bool invert)

Specifies whether the result of the clip filter should be inverted.

If set to false (the default), all regions where the implicit function is negative will be removed. If set to true, all regions where the implicit function is positive will be removed.

inline const `vtkm::ImplicitFunctionGeneral` &**GetImplicitFunction**() const

Specifies the implicit function to be used to perform the clip operation.

In the example provided below the `vtkm::Sphere` implicit function is used. This function evaluates to a negative value if points from the original dataset occur within the sphere, evaluates to 0 if the points occur on the surface of the sphere, and evaluates to a positive value if the points occur outside the sphere.



Example 3: Using `vtkm::filter::contour::ClipWithImplicitFunction`.

```

1 // Parameters needed for implicit function
2 vtkm::Sphere implicitFunction(vtkm::make_Vec(1, 0, 1), 0.5);
3
4 // Create an instance of a clip filter with this implicit function.
5 vtkm::filter::contour::ClipWithImplicitFunction clip;
6 clip.SetImplicitFunction(implicitFunction);
7
8 // By default, ClipWithImplicitFunction will remove everything inside the sphere.
9 // Set the invert clip flag to keep the inside of the sphere and remove everything
10 // else.
11 clip.SetInvertClip(true);
12
13 // Execute the clip filter
14 vtkm::cont::DataSet outData = clip.Execute(inData);

```

## 10.4 Density Estimation

Density estimation takes a collection of samples and estimates the density of the samples in each part of the domain (or estimate the probability that a sample would be at a location in the domain). The domain of samples could be a physical space, such as with particle density, or in an abstract place, such as with a histogram. The `vtkm::filter::density_estimate` module contains filters that estimate density in a variety of ways.

### 10.4.1 Histogram

The `vtkm::filter::density_estimate::Histogram` filter computes a histogram of a given scalar field.

class **Histogram** : public `vtkm::filter::Filter`

Construct the histogram of a given field.

The range of the field is evenly split to a set number of bins (set by `SetNumberOfBins()`). This filter then counts the number of values in the filter that are in each bin.

The result of this filter is stored in a `vtkm::cont::DataSet` with no points or cells. It contains only a single field containing the histogram (bin counts). The field has an association of `vtkm::cont::Field::Association::WholeDataSet`. The field contains an array of `vtkm::Id` with the bin counts. By default, the field is named “histogram”, but that can be changed with the `SetOutputFieldName()` method.

If this filter is run on a partitioned data set, the result will be a `vtkm::cont::PartitionedDataSet` containing a single `vtkm::cont::DataSet` as previously described.

## Public Functions

inline void **SetNumberOfBins**(vtkm::Id count)

Set the number of bins for the resulting histogram.

By default, a histogram with 10 bins is created.

inline vtkm::Id **GetNumberOfBins**() const

Get the number of bins for the resulting histogram.

inline void **SetRange**(const vtkm::Range &range)

Set the range to use to generate the histogram.

If range is set to empty, the field's global range (computed using `vtkm::cont::FieldRangeGlobalCompute`) will be used.

inline const vtkm::Range &**GetRange**() const

Get the range used to generate the histogram.

If the returned range is empty, then the field's global range will be used.

inline vtkm::Float64 **GetBinDelta**() const

Returns the size of bin in the computed histogram.

This value is only valid after a call to `Execute`.

inline vtkm::Range **GetComputedRange**() const

Returns the range used for most recent execute.

If `SetRange` is used to specify a non-empty range, then this range will be returned. Otherwise, the computed range is returned. This value is only valid after a call to `Execute`.

## 10.4.2 Particle Density

VTK-m provides multiple filters to take as input a collection of points and build a regular mesh containing an estimate of the density of particles in that space. These filters inherit from `vtkm::filter::density_estimate::ParticleDensityBase`.

class **ParticleDensityBase** : public vtkm::filter::Filter

Subclassed by `vtkm::filter::density_estimate::ParticleDensityCloudInCell`, `vtkm::filter::density_estimate::ParticleDensityNearest`

## Public Functions

inline void **SetComputeNumberDensity**(bool flag)

Toggles between summing mass and computing instances.

When this flag is false (the default), the active field of the input is accumulated in each bin of the output.

When this flag is set to true, the active field is ignored and the associated particles are simply counted.

inline bool **GetComputeNumberDensity**() const

Toggles between summing mass and computing instances.

When this flag is false (the default), the active field of the input is accumulated in each bin of the output.

When this flag is set to true, the active field is ignored and the associated particles are simply counted.

inline void **SetDivideByVolume**(bool flag)

Specifies whether the accumulated mass (or count) is divided by the volume of the cell.

When this flag is on (the default), the computed mass will be divided by the volume of the bin to give a density value. Turning off this flag provides an accumulated mass or count.

inline bool **GetDivideByVolume**() const

Specifies whether the accumulated mass (or count) is divided by the volume of the cell.

When this flag is on (the default), the computed mass will be divided by the volume of the bin to give a density value. Turning off this flag provides an accumulated mass or count.

inline void **SetDimension**(const vtkm::Id3 &dimension)

The number of bins in the grid used as regions to estimate density.

To estimate particle density, this filter defines a uniform grid in space.

The numbers specify the number of *bins* (i.e. cells in the output mesh) in each dimension, not the number of points in the output mesh.

inline vtkm::Id3 **GetDimension**() const

The number of bins in the grid used as regions to estimate density.

To estimate particle density, this filter defines a uniform grid in space.

The numbers specify the number of *bins* (i.e. cells in the output mesh) in each dimension, not the number of points in the output mesh.

inline void **SetOrigin**(const vtkm::Vec3f &origin)

The lower-left (minimum) corner of the domain of density estimation.

inline vtkm::Vec3f **GetOrigin**() const

The lower-left (minimum) corner of the domain of density estimation.

inline void **SetSpacing**(const vtkm::Vec3f &spacing)

The spacing of the grid points used to form the grid for density estimation.

inline vtkm::Vec3f **GetSpacing**() const

The spacing of the grid points used to form the grid for density estimation.

inline void **SetBounds**(const vtkm::Bounds &bounds)

The bounds of the region where density estimation occurs.

This method can be used in place of **SetOrigin** and **SetSpacing**. It is often easiest to compute the bounds of the input coordinate system (or other spatial region) to use as the input.

The dimensions must be set before the bounds are set. Calling **SetDimension** will change the ranges of the bounds.

## Nearest Grid Point

The `vtkm::filter::density_estimate::ParticleDensityNearestGridPoint` filter defines a 3D grid of bins. It then takes from the input a collection of particles, identifies which bin each particle lies in, and sums some attribute from a field of the input (or the particles can simply be counted).

class **ParticleDensityNearestGridPoint** : public vtkm::filter::density\_estimate::ParticleDensityBase

Estimate the density of particles using the Nearest Grid Point method.

This filter takes a collection of particles. The particles are infinitesimal in size with finite mass (or other scalar attributes such as charge). The filter estimates density by imposing a regular grid (as specified by `SetDimensions`, `SetOrigin`, and `SetSpacing`) and summing the mass of particles within each cell in the grid. Each input particle is assigned to one bin that it falls in.

The mass of particles is established by setting the active field (using `SetActiveField`). Note that the “mass” can actually be another quantity. For example, you could use electrical charge in place of mass to compute the charge density. Once the sum of the mass is computed for each grid cell, the mass is divided by the volume of the cell. Thus, the density will be computed as the units of the mass field per the cubic units of the coordinate system. If you just want a sum of the mass in each cell, turn off the `DivideByVolume` feature of this filter. In addition, you can also simply count the number of particles in each cell by calling `SetComputeNumberDensity(true)`.

This operation is helpful in the analysis of particle-based simulation where the data often requires conversion or deposition of particles' attributes, such as mass, to an overlaying mesh. This allows further identification of regions of interest based on the spatial distribution of particles attributes, for example, high density regions could be considered as clusters or halos while low density regions could be considered as bubbles or cavities in the particle data.

Since there is no specific `vtkm::cont::CellSet` for particles in VTK-m, this filter treats the `vtkm::cont::CoordinateSystem` of the `vtkm::cont::DataSet` as the positions of the particles while ignoring the details of the `vtkm::cont::CellSet`.

## Cloud in Cell

The `vtkm::filter::density_estimate::ParticleDensityCloudInCell` filter defines a 3D grid of bins. It then takes from the input a collection of particles, identifies which bin each particle lies in, and then redistributes each particle's attribute to the 8 vertices of the containing bin. The filter then sums up all the contributions of particles for each bin in the grid.

class **ParticleDensityCloudInCell** : public `vtkm::filter::density_estimate::ParticleDensityBase`

Estimate the density of particles using the Cloud-in-Cell method.

This filter takes a collection of particles. The particles are infinitesimal in size with finite mass (or other scalar attributes such as charge). The filter estimates density by imposing a regular grid (as specified by `SetDimensions`, `SetOrigin`, and `SetSpacing`) and summing the mass of particles within each cell in the grid. The particle's mass is divided among the 8 nearest neighboring bins. This differs from `ParticleDensityNearestGridPoint`, which just finds the nearest containing bin.

The mass of particles is established by setting the active field (using `SetActiveField`). Note that the “mass” can actually be another quantity. For example, you could use electrical charge in place of mass to compute the charge density. Once the sum of the mass is computed for each grid cell, the mass is divided by the volume of the cell. Thus, the density will be computed as the units of the mass field per the cubic units of the coordinate system. If you just want a sum of the mass in each cell, turn off the `DivideByVolume` feature of this filter. In addition, you can also simply count the number of particles in each cell by calling `SetComputeNumberDensity(true)`.

This operation is helpful in the analysis of particle-based simulation where the data often requires conversion or deposition of particles' attributes, such as mass, to an overlaying mesh. This allows further identification of regions of interest based on the spatial distribution of particles attributes, for example, high density regions could be considered as clusters or halos while low density regions could be considered as bubbles or cavities in the particle data.

### 10.4.3 Statistics

Simple descriptive statistics for data in field arrays can be computed with `vtkm::filter::density_estimate::Statistics`.

class **Statistics** : public `vtkm::filter::Filter`

Computes descriptive statistics of an input field.

This filter computes the following statistics on the active field of the input.

- N
- Min
- Max
- Sum
- Mean
- M2
- M3
- M4
- SampleStddev
- PopulationStddev
- SampleVariance
- PopulationVariance
- Skewness
- Kurtosis

M2, M3, and M4 are the second, third, and fourth moments, respectively.

Note that this filter treats the “sample” and the “population” as the same with the same mean. The difference between the two forms of variance is how they are normalized. The population variance is normalized by dividing the second moment by N. The sample variance uses Bessel’s correction and divides the second moment by N-1 instead. The standard deviation, which is just the square root of the variance, follows the same difference.

The result of this filter is stored in a `vtkm::cont::DataSet` with no points or cells. It contains only fields with the same names as the list above. All fields have an association of `vtkm::cont::Field::Association::WholeDataSet`.

If `Execute` is called with a `vtkm::cont::PartitionedDataSet`, then the partitions of the output will match those of the input. Additionally, the containing `vtkm::cont::PartitionedDataSet` will contain the same fields associated with `vtkm::cont::Field::Association::Global` that provide the overall statistics of all partitions.

If this filter is used inside of an MPI job, then each `vtkm::cont::DataSet` result will be *local* to the MPI rank. If `Execute` is called with a `vtkm::cont::PartitionedDataSet`, then the fields attached to the `vtkm::cont::PartitionedDataSet` container will have the overall statistics across all MPI ranks (in addition to all partitions). Global MPI statistics for a single `vtkm::cont::DataSet` can be computed by creating a `vtkm::cont::PartitionedDataSet` with that as a single partition.

## 10.5 Entity Extraction

VTK-m contains a collection of filters that extract a portion of one `vtkm::cont::DataSet` and construct a new `vtkm::cont::DataSet` based on that portion of the geometry. These filters are collected in the `vtkm::filter::entity_extraction` module.

### 10.5.1 External Faces

`vtkm::filter::entity_extraction::ExternalFaces` is a filter that extracts all the external faces from a polyhedral data set. An external face is any face that is on the boundary of a mesh. Thus, if there is a hole in a volume, the boundary of that hole will be considered external. More formally, an external face is one that belongs to only one cell in a mesh.

class **ExternalFaces** : public `vtkm::filter::Filter`

Extract external faces of a geometry.

`ExternalFaces` is a filter that extracts all external faces from a data set. An external face is defined as a face/side of a cell that belongs only to one cell in the entire mesh.

#### Public Functions

inline virtual bool **CanThread()** const override

Returns whether the filter can execute on partitions in concurrent threads.

If a derived class's implementation of `DoExecute` cannot run on multiple threads, then the derived class should override this method to return false.

inline bool **GetCompactPoints()** const

Option to remove unused points and compact result into a smaller array.

When `CompactPoints` is on, instead of copying the points and point fields from the input, the filter will create new compact fields without the unused elements. When off (the default), unused points will remain listed in the topology, but point fields and coordinate systems will be shallow-copied to the output.

inline void **SetCompactPoints**(bool value)

Option to remove unused points and compact result into a smaller array.

When `CompactPoints` is on, instead of copying the points and point fields from the input, the filter will create new compact fields without the unused elements. When off (the default), unused points will remain listed in the topology, but point fields and coordinate systems will be shallow-copied to the output.

inline bool **GetPassPolyData()** const

Specify how polygonal data (polygons, lines, and vertices) will be handled.

When on (the default), these cells will be passed to the output. When off, these cells will be removed from the output. (Because they have less than 3 topological dimensions, they are not considered to have any "faces.")

void **SetPassPolyData**(bool value)

Specify how polygonal data (polygons, lines, and vertices) will be handled.

When on (the default), these cells will be passed to the output. When off, these cells will be removed from the output. (Because they have less than 3 topological dimensions, they are not considered to have any "faces.")

## 10.5.2 Extract Geometry

The `vtkm::filter::entity_extraction::ExtractGeometry` filter extracts all of the cells in a `vtkm::cont::DataSet` that is inside or outside of an implicit function. Implicit functions are described in Chapter 15 (Implicit Functions). They define a function in 3D space that follow a geometric shape. The inside of the implicit function is the region of negative values.

class **ExtractGeometry** : public `vtkm::filter::Filter`

Extract a subset of geometry based on an implicit function.

Extracts from its input geometry all cells that are either completely inside or outside of a specified implicit function. Any type of data can be input to this filter.

To use this filter you must specify an implicit function. You must also specify whether to extract cells laying inside or outside of the implicit function. (The inside of an implicit function is the negative values region.) An option exists to extract cells that are neither inside or outside (i.e., boundary).

This differs from `vtkm::filter::contour::ClipWithImplicitFunction` in that `vtkm::filter::contour::ClipWithImplicitFunction` will subdivide boundary cells into new cells whereas this filter will not, producing a more “crinkly” output.

### Public Functions

inline void **SetImplicitFunction**(const `vtkm::ImplicitFunctionGeneral` &func)

Specifies the implicit function to be used to perform extract geometry.

Only a limited number of implicit functions are supported. See `vtkm::ImplicitFunctionGeneral` for information on which ones.

inline bool **GetExtractInside**() const

Specify the region of the implicit function to keep cells.

Determines whether to extract the geometry that is on the inside of the implicit function (where the function is less than 0) or the outside (where the function is greater than 0). This flag is true by default (i.e., the interior of the implicit function will be extracted).

inline void **SetExtractInside**(bool value)

Specify the region of the implicit function to keep cells.

Determines whether to extract the geometry that is on the inside of the implicit function (where the function is less than 0) or the outside (where the function is greater than 0). This flag is true by default (i.e., the interior of the implicit function will be extracted).

inline void **ExtractInsideOn**()

Specify the region of the implicit function to keep cells.

Determines whether to extract the geometry that is on the inside of the implicit function (where the function is less than 0) or the outside (where the function is greater than 0). This flag is true by default (i.e., the interior of the implicit function will be extracted).

inline void **ExtractInsideOff**()

Specify the region of the implicit function to keep cells.

Determines whether to extract the geometry that is on the inside of the implicit function (where the function is less than 0) or the outside (where the function is greater than 0). This flag is true by default (i.e., the interior of the implicit function will be extracted).

inline bool **GetExtractBoundaryCells()** const

Specify whether cells on the boundary should be extracted.

The implicit function used to extract geometry is likely to intersect some of the cells of the input. If this flag is true, then any cells intersected by the implicit function are extracted and included in the output. This flag is false by default.

inline void **SetExtractBoundaryCells**(bool value)

Specify whether cells on the boundary should be extracted.

The implicit function used to extract geometry is likely to intersect some of the cells of the input. If this flag is true, then any cells intersected by the implicit function are extracted and included in the output. This flag is false by default.

inline void **ExtractBoundaryCellsOn()**

Specify whether cells on the boundary should be extracted.

The implicit function used to extract geometry is likely to intersect some of the cells of the input. If this flag is true, then any cells intersected by the implicit function are extracted and included in the output. This flag is false by default.

inline void **ExtractBoundaryCellsOff()**

Specify whether cells on the boundary should be extracted.

The implicit function used to extract geometry is likely to intersect some of the cells of the input. If this flag is true, then any cells intersected by the implicit function are extracted and included in the output. This flag is false by default.

inline bool **GetExtractOnlyBoundaryCells()** const

Specify whether to extract cells only on the boundary.

When this flag is off (the default), this filter extract the geometry in the region specified by the implicit function. When this flag is on, then only those cells that intersect the surface of the implicit function are extracted.

inline void **SetExtractOnlyBoundaryCells**(bool value)

GetExtractOnlyBoundaryCells.

inline void **ExtractOnlyBoundaryCellsOn()**

GetExtractOnlyBoundaryCells.

inline void **ExtractOnlyBoundaryCellsOff()**

GetExtractOnlyBoundaryCells.

### 10.5.3 Extract Points

The `vtkm::filter::entity_extraction::ExtractPoints` filter behaves the same as `vtkm::filter::entity_extraction::ExtractGeometry` (Section 10.5.2) except that the geometry is converted into a point cloud. The filter determines whether each point is inside or outside the implicit function and passes only those that match the criteria. The cell information of the input is thrown away and replaced with a cell set of “vertex” cells, one per point.

class **ExtractPoints** : public vtkm::filter::Filter

Extract only points from a geometry using an implicit function.

Extract only the points that are either inside or outside of a VTK-m implicit function. Examples include planes, spheres, boxes, etc.



Note that while any geometry type can be provided as input, the output is represented by an explicit representation of points using `vtkm::cont::CellSetSingleType` with one vertex cell per point.

## Public Functions

inline bool **GetCompactPoints()** const

Option to remove unused points and compact result into a smaller array.

When CompactPoints is on, instead of copying the points and point fields from the input, the filter will create new compact fields without the unused elements. When off (the default), unused points will remain listed in the topology, but point fields and coordinate systems will be shallow-copied to the output.

inline void **SetCompactPoints**(bool value)

Option to remove unused points and compact result into a smaller array.

When CompactPoints is on, instead of copying the points and point fields from the input, the filter will create new compact fields without the unused elements. When off (the default), unused points will remain listed in the topology, but point fields and coordinate systems will be shallow-copied to the output.

inline void **SetImplicitFunction**(const vtkm::ImplicitFunctionGeneral &func)

Specifies the implicit function to be used to perform extract points.

Only a limited number of implicit functions are supported. See `vtkm::ImplicitFunctionGeneral` for information on which ones.

inline bool **GetExtractInside()** const

Specify the region of the implicit function to keep points.

Determines whether to extract the points that are on the inside of the implicit function (where the function is less than 0) or the outside (where the function is greater than 0). This flag is true by default (i.e., the interior of the implicit function will be extracted).

inline void **SetExtractInside**(bool value)

Specify the region of the implicit function to keep points.

Determines whether to extract the points that are on the inside of the implicit function (where the function is less than 0) or the outside (where the function is greater than 0). This flag is true by default (i.e., the interior of the implicit function will be extracted).

inline void **ExtractInsideOn**()

Specify the region of the implicit function to keep points.

Determines whether to extract the points that are on the inside of the implicit function (where the function is less than 0) or the outside (where the function is greater than 0). This flag is true by default (i.e., the interior of the implicit function will be extracted).

inline void **ExtractInsideOff**()

Specify the region of the implicit function to keep points.

Determines whether to extract the points that are on the inside of the implicit function (where the function is less than 0) or the outside (where the function is greater than 0). This flag is true by default (i.e., the interior of the implicit function will be extracted).

## 10.5.4 Extract Structured

`vtkm::filter::entity_extraction::ExtractStructured` is a filter that extracts a volume of interest (VOI) from a structured data set. In addition the filter is able to subsample the VOI while doing the extraction. The input and output of this filter are a structured data sets.

class **ExtractStructured** : public `vtkm::filter::Filter`

Select a piece (e.g., volume of interest) and/or subsample structured points dataset.

Select or subsample a portion of an input structured dataset. The selected portion of interested is referred to as the Volume Of Interest, or VOI. The output of this filter is a structured dataset. The filter treats input data of any topological dimension (i.e., point, line, plane, or volume) and can generate output data of any topological dimension.

To use this filter set the VOI ivar which are i-j-k min/max indices that specify a rectangular region in the data. (Note that these are 0-offset.) You can also specify a sampling rate to subsample the data.

Typical applications of this filter are to extract a slice from a volume for image processing, subsampling large volumes to reduce data size, or extracting regions of a volume with interesting data.

### Public Functions

inline `vtkm::RangeId3` **GetVOI**() const

Specifies what volume of interest (VOI) should be extracted by the filter.

The VOI is specified using the 3D indices of the structured mesh. Meshes with fewer than 3 dimensions will ignore the extra dimensions in the VOI. The VOI is inclusive on the minium index and exclusive on the maximum index.

By default the VOI is the entire input.

inline void **SetVOI**(`vtkm::Id` i0, `vtkm::Id` i1, `vtkm::Id` j0, `vtkm::Id` j1, `vtkm::Id` k0, `vtkm::Id` k1)

Specifies what volume of interest (VOI) should be extracted by the filter.

The VOI is specified using the 3D indices of the structured mesh. Meshes with fewer than 3 dimensions will ignore the extra dimensions in the VOI. The VOI is inclusive on the minium index and exclusive on the maximum index.

By default the VOI is the entire input.

inline void **SetVOI**(`vtkm::Id` extents[6])

Specifies what volume of interest (VOI) should be extracted by the filter.

The VOI is specified using the 3D indices of the structured mesh. Meshes with fewer than 3 dimensions will ignore the extra dimensions in the VOI. The VOI is inclusive on the minium index and exclusive on the maximum index.

By default the VOI is the entire input.

inline void **SetVOI**(`vtkm::Id3` minPoint, `vtkm::Id3` maxPoint)

Specifies what volume of interest (VOI) should be extracted by the filter.

The VOI is specified using the 3D indices of the structured mesh. Meshes with fewer than 3 dimensions will ignore the extra dimensions in the VOI. The VOI is inclusive on the minium index and exclusive on the maximum index.

By default the VOI is the entire input.

inline void **SetVOI**(const vtkm::RangeId3 &voi)

Specifies what volume of interest (VOI) should be extracted by the filter.

The VOI is specified using the 3D indices of the structured mesh. Meshes with fewer than 3 dimensions will ignore the extra dimensions in the VOI. The VOI is inclusive on the minimum index and exclusive on the maximum index.

By default the VOI is the entire input.

inline vtkm::Id3 **GetSampleRate**() const

Specifies the sample rate of the VOI.

The input data can be subsampled by selecting every n-th value. The sampling can be different in each dimension. The default sampling rate is (1,1,1), meaning that no subsampling will occur.

inline void **SetSampleRate**(vtkm::Id i, vtkm::Id j, vtkm::Id k)

Specifies the sample rate of the VOI.

The input data can be subsampled by selecting every n-th value. The sampling can be different in each dimension. The default sampling rate is (1,1,1), meaning that no subsampling will occur.

inline void **SetSampleRate**(vtkm::Id3 sampleRate)

Specifies the sample rate of the VOI.

The input data can be subsampled by selecting every n-th value. The sampling can be different in each dimension. The default sampling rate is (1,1,1), meaning that no subsampling will occur.

### 10.5.5 Ghost Cell Removal

The `vtkm::filter::entity_extraction::GhostCellRemove` filter is used to remove cells from a data set according to a cell centered field that specifies whether a cell is a regular cell or a ghost cell. By default, the filter will get the ghost cell information that is registered in the input `vtkm::cont::DataSet`, but it is also possible to specify an arbitrary field for this purpose.

class **GhostCellRemove** : public vtkm::filter::Filter

Removes cells marked as ghost cells.

This filter inspects the ghost cell field of the input and removes any cells marked as ghost cells. Although this filter nominally operates on ghost cells, other classifications, such as blanked cells, can also be recorded in the ghost cell array. See `vtkm::CellClassification` for the list of flags typical in a ghost array.

By default, if the input is a structured data set the filter will attempt to output a structured data set. This will be the case if all the cells along a boundary are marked as ghost cells together, which is common. If creating a structured data set is not possible, an explicit data set is produced.

#### Public Functions

inline void **SetRemoveGhostField**(bool flag)

Specify whether the ghost cell array should be removed from the input.

If this flag is true, then the ghost cell array will not be passed to the output.

inline bool **GetRemoveGhostField**() const

Specify whether the ghost cell array should be removed from the input.

If this flag is true, then the ghost cell array will not be passed to the output.

inline void **SetTypesToRemove**(vtkm::UInt8 typeFlags)

Specify which types of cells to remove.

The types to remove are specified by the flags in `vtkm::CellClassification`. Any cell with a ghost array flag matching one or more of these flags will be removed.

inline vtkm::UInt8 **GetTypesToRemove**() const

Specify which types of cells to remove.

The types to remove are specified by the flags in `vtkm::CellClassification`. Any cell with a ghost array flag matching one or more of these flags will be removed.

inline void **SetTypesToRemoveToAll**()

Set filter to remove any special cell type.

This method sets the state to remove any cell that does not have a “normal” ghost cell value of 0. Any other value represents a cell that is placeholder or otherwise not really considered part of the cell set.

inline bool **AreAllTypesRemoved**() const

Returns true if all abnormal cell types are removed.

inline bool **GetUseGhostCellsAsField**() const

Specify whether the marked ghost cells or a named field should be used as the ghost field.

When this flag is true (the default), the filter will get from the input `vtkm::cont::DataSet` the field (with the `GetGhostCellField` method). When this flag is false, the `SetActiveField` method of this class should be used to select which field to use as ghost cells.

inline void **SetUseGhostCellsAsField**(bool flag)

Specify whether the marked ghost cells or a named field should be used as the ghost field.

When this flag is true (the default), the filter will get from the input `vtkm::cont::DataSet` the field (with the `GetGhostCellField` method). When this flag is false, the `SetActiveField` method of this class should be used to select which field to use as ghost cells.

## 10.5.6 Threshold

A threshold operation removes topology elements from a data set that do not meet a specified criterion. The `vtkm::filter::entity_extraction::Threshold` filter removes all cells where the a field is outside a range of values.

Note that `vtkm::filter::entity_extraction::Threshold` either passes an entire cell or discards an entire cell. This can consequently lead to jagged surfaces at the interface of the threshold caused by the shape of cells that jut inside or outside the removed region. See [Section 10.3.3 \(Clip with Field\)](#) for a clipping filter that will clip off a smooth region of the mesh.

class **Threshold** : public vtkm::filter::Filter

Extracts cells that satisfy a threshold criterion.

Extracts all cells from any dataset type that satisfy a threshold criterion. The output of this filter stores its connectivity in a `vtkm::cont::CellSetExplicit<>` regardless of the input dataset type or which cells are passed.

You can threshold either on point or cell fields. If thresholding on point fields, you must specify whether a cell should be kept if some but not all of its incident points meet the criteria.

Although *Threshold* is primarily designed for scalar fields, there is support for thresholding on 1 or all of the components in a vector field. See the *SetComponentToTest()*, *SetComponentToTestToAny()*, and *SetComponentToTestToAll()* methods for more information.

Use *SetActiveField()* and related methods to set the field to threshold on.

## Public Functions

inline void **SetLowerThreshold**(vtkm::Float64 value)

Specifies the lower scalar value.

Any cells where the scalar field is less than this value are removed.

inline void **SetUpperThreshold**(vtkm::Float64 value)

Specifies the upper scalar value.

Any cells where the scalar field is more than this value are removed.

inline vtkm::Float64 **GetLowerThreshold**() const

Specifies the lower scalar value.

Any cells where the scalar field is less than this value are removed.

inline vtkm::Float64 **GetUpperThreshold**() const

Specifies the upper scalar value.

Any cells where the scalar field is more than this value are removed.

void **SetThresholdBelow**(vtkm::Float64 value)

Sets the threshold criterion to pass any value less than or equal to *value*.

void **SetThresholdAbove**(vtkm::Float64 value)

Sets the threshold criterion to pass any value greater than or equal to *value*.

void **SetThresholdBetween**(vtkm::Float64 value1, vtkm::Float64 value2)

Set the threshold criterion to pass any value between (inclusive) the given values.

This method is equivalent to calling *SetLowerThreshold(value1)* and *SetUpperThreshold(value2)*.

inline void **SetComponentToTest**(vtkm::IdComponent component)

Specifies that the threshold criteria should be applied to a specific vector component.

When thresholding on a vector field (which has more than one component per entry), the *Threshold* filter will by default compare the threshold criterion to the first component of the vector (component index 0).

Use this method to change the component to test against.

inline void **SetComponentToTestToAny**()

Specifies that the threshold criteria should be applied to a specific vector component.

This method sets that the threshold criteria should be applied to all the components of the input vector field and a cell will pass if *any* the components match.

inline void **SetComponentToTestToAll**()

Specifies that the threshold criteria should be applied to a specific vector component.

This method sets that the threshold criteria should be applied to all the components of the input vector field and a cell will pass if *all* the components match.

inline void **SetAllInRange**(bool value)

Specify criteria for cells that have some points matching.

When thresholding on a point field, each cell must consider the multiple values associated with all incident points. When this flag is false (the default), the cell is passed if *any* of the incident points matches the threshold criterion. When this flag is true, the cell is passed only if *all* the incident points match the threshold criterion.

inline bool **GetAllInRange**() const

Specify criteria for cells that have some points matching.

When thresholding on a point field, each cell must consider the multiple values associated with all incident points. When this flag is false (the default), the cell is passed if *any* of the incident points matches the threshold criterion. When this flag is true, the cell is passed only if *all* the incident points match the threshold criterion.

inline void **SetInvert**(bool value)

Inverts the threshold result.

When set to true, the threshold result is inverted. That is, cells that would have been in the output with this option set to false (the default) are excluded while cells that would have been excluded from the output are included.

inline bool **GetInvert**() const

Inverts the threshold result.

When set to true, the threshold result is inverted. That is, cells that would have been in the output with this option set to false (the default) are excluded while cells that would have been excluded from the output are included.

## 10.6 Field Conversion

Field conversion modifies a field of a `vtkm::cont::DataSet` to have roughly equivalent values but with a different structure. These filters allow the field to be used in places where they otherwise would not be applicable.

### 10.6.1 Cell Average

`vtkm::filter::field_conversion::CellAverage` is the cell average filter. It will take a data set with a collection of cells and a field defined on the points of the data set and create a new field defined on the cells. The values of this new derived field are computed by averaging the values of the input field at all the incident points. This is a simple way to convert a point field to a cell field.

class **CellAverage** : public `vtkm::filter::Filter`

Point to cell interpolation filter.

`CellAverage` is a filter that transforms point data (i.e., data specified at cell points) into cell data (i.e., data specified per cell). The method of transformation is based on averaging the data values of all points used by particular cell.

The point field to convert comes from the active scalars. The default name for the output cell field is the same name as the input point field. The name can be overridden as always using the `SetOutputFieldName()` method.

## 10.6.2 Point Average

`vtkm::filter::field_conversion::PointAverage` is the point average filter. It will take a data set with a collection of cells and a field defined on the cells of the data set and create a new field defined on the points. The values of this new derived field are computed by averaging the values of the input field at all the incident cells. This is a simple way to convert a cell field to a point field.

class **PointAverage** : public `vtkm::filter::Filter`

Cell to Point interpolation filter.

*PointAverage* is a filter that transforms cell data (i.e., data specified per cell) into point data (i.e., data specified at cell points). The method of transformation is based on averaging the data values of all cells using a particular point.

The cell field to convert comes from the active scalars. The default name for the output cell field is the same name as the input point field. The name can be overridden as always using the `SetOutputFieldName()` method.

## 10.7 Field Transform

VTK-m provides multiple filters to convert fields through some mathematical relationship.

### 10.7.1 Composite Vectors

The `vtkm::filter::field_transform::CompositeVectors` filter allows you to group multiple scalar fields into a single vector field. This is convenient when importing data from a source that stores vector components in separate arrays.

class **CompositeVectors** : public `vtkm::filter::Filter`

Combine multiple scalar fields into a single vector field.

Scalar fields are selected as the active input fields, and the combined vector field is set at the output. The `SetFieldNameList()` method takes a `std::vector` of field names to use as the component fields. Alternately, the `SetActiveField()` method can be used to select the fields independently.

All of the input fields must be scalar values. The type of the first field determines the type of the output vector field.

#### Public Functions

void **SetFieldNameList**(const `std::vector<std::string>` &fieldNameList, `vtkm::cont::Field::Association` association = `vtkm::cont::Field::Association::Any`)

Specifies the names of the fields to use as components for the output.

`vtkm::IdComponent` **GetNumberOfFields**() const

The number of fields specified as inputs.

This will be the number of components in the generated field.

## 10.7.2 Cylindrical Coordinate System Transform

The `vtkm::filter::field_transform::CylindricalCoordinateTransform` filter is a coordinate system transformation. The filter will take a data set and transform the points of the coordinate system. By default, the filter will transform the coordinates from a Cartesian coordinate system to a cylindrical coordinate system. The order for cylindrical coordinates is  $(R, \theta, Z)$ . The output coordinate system will be set to the new computed coordinates.

class **CylindricalCoordinateTransform** : public `vtkm::filter::Filter`

Transform coordinates between Cartesian and cylindrical.

By default, this filter will transform the first coordinate system, but this can be changed by setting the active field.

The resulting transformation will be set as the first coordinate system in the output.

### Public Functions

inline void **SetCartesianToCylindrical()**

Establish a transformation from Cartesian to cylindrical coordinates.

inline void **SetCylindricalToCartesian()**

Establish a transformation from cylindrical to Cartesian coordinates.

## 10.7.3 Field to Colors

The `vtkm::filter::field_transform::FieldToColors` filter takes a field in a data set, looks up each value in a color table, and writes the resulting colors to a new field. The color to be used for each field value is specified using a `vtkm::cont::ColorTable` object. `vtkm::cont::ColorTable` objects are also used with VTK-m's rendering module and are described in [Section 11.8 \(Color Tables\)](#).

`vtkm::filter::field_transform::FieldToColors` has three modes it can use to select how it should treat the input field. These input modes are contained in `vtkm::filter::field_transform::FieldToColors::InputMode`. Additionally, `vtkm::filter::field_transform::FieldToColors` has different modes in which it can represent colors in its output. These output modes are contained in `vtkm::filter::field_transform::FieldToColors::OutputMode`.

class **FieldToColors** : public `vtkm::filter::Filter`

Convert an arbitrary field to an RGB or RGBA field.

This filter is useful for generating colors that could be used for rendering or other purposes.

### Public Types

enum class **InputMode**

Identifiers used to specify how `FieldToColors` should treat its input scalars.

*Values:*

enumerator **Scalar**

Treat the field as a scalar field.

It is an error to provide a field of any type that cannot be directly converted to a basic floating point number (such as a vector).



enumerator **Magnitude**

Map the magnitude of the field.

Given a vector field, the magnitude of each field value is taken before looking it up in the color table.

enumerator **Component**

Map a component of a vector field as if it were a scalar.

Given a vector field, a particular component is looked up in the color table as if that component were in a scalar field. The component to map is selected with [SetMappingComponent\(\)](#).

enum class **OutputMode**

Identifiers used to specify what output [FieldToColors](#) will generate.

*Values:*

enumerator **RGB**

Write out RGB fixed precision color values.

Output colors are represented as RGB values with each component represented by an unsigned byte. Specifically, these are [vtkm::Vec3ui\\_8](#) values.

enumerator **RGBA**

Write out RGBA fixed precision color values.

Output colors are represented as RGBA values with each component represented by an unsigned byte. Specifically, these are [vtkm::Vec4ui\\_8](#) values.

**Public Functions**

inline void **SetColorTable**(const vtkm::cont::ColorTable &table)

Specifies the [vtkm::cont::ColorTable](#) object to use to map field values to colors.

inline const vtkm::cont::ColorTable &**GetColorTable**() const

Specifies the [vtkm::cont::ColorTable](#) object to use to map field values to colors.

inline void **SetMappingMode**(InputMode mode)

Specify the mapping mode.

inline void **SetMappingToScalar**()

Treat the field as a scalar field.

It is an error to provide a field of any type that cannot be directly converted to a basic floating point number (such as a vector).

inline void **SetMappingToMagnitude**()

Map the magnitude of the field.

Given a vector field, the magnitude of each field value is taken before looking it up in the color table.

inline void **SetMappingToComponent**()

Map a component of a vector field as if it were a scalar.

Given a vector field, a particular component is looked up in the color table as if that component were in a scalar field. The component to map is selected with [SetMappingComponent\(\)](#).

inline *InputMode* **GetMappingMode**() const

Specify the mapping mode.

inline bool **IsMappingScalar**() const

Returns true if this filter is in scalar mapping mode.

inline bool **IsMappingMagnitude**() const

Returns true if this filter is in magnitude mapping mode.

inline bool **IsMappingComponent**() const

Returns true if this filter is vector component mapping mode.

inline void **SetMappingComponent**(*vtkm::IdComponent* comp)

Specifies the component of the vector to use in the mapping.

This only has an effect if the input mapping mode is set to *FieldToColors::InputMode::Component*.

inline *vtkm::IdComponent* **GetMappingComponent**() const

Specifies the component of the vector to use in the mapping.

This only has an effect if the input mapping mode is set to *FieldToColors::InputMode::Component*.

inline void **SetOutputMode**(*OutputMode* mode)

Specify the output mode.

inline void **SetOutputToRGB**()

Write out RGB fixed precision color values.

Output colors are represented as RGB values with each component represented by an unsigned byte. Specifically, these are *vtkm::Vec3ui\_8* values.

inline void **SetOutputToRGBA**()

Write out RGBA fixed precision color values.

Output colors are represented as RGBA values with each component represented by an unsigned byte. Specifically, these are *vtkm::Vec4ui\_8* values.

inline *OutputMode* **GetOutputMode**() const

Specify the output mode.

inline bool **IsOutputRGB**() const

Returns true if this filter is in RGB output mode.

inline bool **IsOutputRGBA**() const

Returns true if this filter is in RGBA output mode.

void **SetNumberOfSamplingPoints**(*vtkm::Int32* count)

Specifies how many samples to use when looking up color values.

The implementation of *FieldToColors* first builds an array of color samples to quickly look up colors for particular values. The size of this lookup array can be adjusted with this parameter. By default, an array of 256 colors is used.

inline *vtkm::Int32* **GetNumberOfSamplingPoints**() const

Specifies how many samples to use when looking up color values.

The implementation of *FieldToColors* first builds an array of color samples to quickly look up colors for particular values. The size of this lookup array can be adjusted with this parameter. By default, an array of 256 colors is used.

## 10.7.4 Generate Ids

The `vtkm::filter::field_transform::GenerateIds` filter creates point and/or cell fields that mimic the identifier for the respective element.

class **GenerateIds** : public `vtkm::filter::Filter`

Adds fields to a `vtkm::cont::DataSet` that give the ids for the points and cells.

This filter will add (by default) a point field named `pointids` that gives the index of the associated point and likewise a cell field named `cellids` for the associated cell indices. These fields are useful for tracking the provenance of the elements of a `vtkm::cont::DataSet` as it gets manipulated by filters. It is also convenient for adding indices to operations designed for fields and generally creating test data.

### Public Functions

inline const std::string &**GetPointFieldName**() const

The name given to the generated point field.

By default, the name is `pointids`.

inline void **SetPointFieldName**(const std::string &name)

The name given to the generated point field.

By default, the name is `pointids`.

inline const std::string &**GetCellFieldName**() const

The name given to the generated cell field.

By default, the name is `cellids`.

inline void **SetCellFieldName**(const std::string &name)

The name given to the generated cell field.

By default, the name is `cellids`.

inline bool **GetGeneratePointIds**() const

Specify whether the point id field is generated.

When `GeneratePointIds` is `true` (the default), a field echoing the point indices is generated. When set to `false`, this output is not created.

inline void **SetGeneratePointIds**(bool flag)

Specify whether the point id field is generated.

When `GeneratePointIds` is `true` (the default), a field echoing the point indices is generated. When set to `false`, this output is not created.

inline bool **GetGenerateCellIds**() const

Specify whether the cell id field is generated.

When `GenerateCellIds` is `true` (the default), a field echoing the cell indices is generated. When set to `false`, this output is not created.

inline void **SetGenerateCellIds**(bool flag)

Specify whether the cell id field is generated.

When `GenerateCellIds` is `true` (the default), a field echoing the cell indices is generated. When set to `false`, this output is not created.

inline bool **GetUseFloat**() const

Specify whether the generated fields should be integer or float.

When `UseFloat` is `false` (the default), then the fields generated will have type `vtkm::Id`. If it is set to `true`, then the fields will be generated with type `vtkm::FloatDefault`.

inline void **SetUseFloat**(bool flag)

Specify whether the generated fields should be integer or float.

When `UseFloat` is `false` (the default), then the fields generated will have type `vtkm::Id`. If it is set to `true`, then the fields will be generated with type `vtkm::FloatDefault`.

## 10.7.5 Log Values

The `vtkm::filter::field_transform::LogValues` filter can be used to take the logarithm of all values in a field. The filter is able to take the logarithm to a number of predefined bases identified by `vtkm::filter::field_transform::LogValues::LogBase`.

class **LogValues** : public `vtkm::filter::Filter`

Adds field to a `vtkm::cont::DataSet` that gives the log values for the user specified field.

By default, `LogValues` takes a natural logarithm (of base e). The base of the logarithm can be set to one of the bases listed in `LogBase` with `SetBaseValue()`.

Logarithms are often used to rescale data to simultaneously show data at different orders of magnitude. It allows small changes in small numbers be visible next to much larger numbers with less precision. One problem with this approach is if there exist numbers very close to zero, the scale at the low range could make all but the smallest numbers comparatively hard to see. Thus, `LogValues` supports setting a minimum value (with `SetMinValue()`) that will clamp any smaller values to that.

### Public Types

enum class **LogBase**

Identifies a type of logarithm as specified by its base.

*Values:*

enumerator **E**

Take the natural logarithm.

The logarithm is set to the mathematical constant e (about 2.718). This is a constant that has many uses in calculus and other mathematics, and a logarithm of base e is often referred to as the “natural” logarithm.

enumerator **TWO**

Take the base 2 logarithm.

The base 2 logarithm is particularly useful for estimating the depth of a binary hierarchy.

enumerator **TEN**

Take the base 10 logarithm.

The base 10 logarithm is handy to convert a number to its order of magnitude based on our standard base 10 human counting system.

## Public Functions

inline const *LogBase* &GetBaseValue() const

Specify the base of the logarithm.

inline void SetBaseValue(const *LogBase* &base)

Specify the base of the logarithm.

inline void SetBaseValueToE()

Take the natural logarithm.

The logarithm is set to the mathematical constant e (about 2.718). This is a constant that has many uses in calculus and other mathematics, and a logarithm of base e is often referred to as the “natural” logarithm.

inline void SetBaseValueTo2()

Take the base 2 logarithm.

The base 2 logarithm is particularly useful for estimating the depth of a binary hierarchy.

inline void SetBaseValueTo10()

Take the base 10 logarithm.

The base 10 logarithm is handy to convert a number to its order of magnitude based on our standard base 10 human counting system.

inline vtkm::FloatDefault GetMinValue() const

Specifies the minimum value to take the logarithm of.

Before taking the logarithm, this filter will check the value to this minimum value and clamp it to the minimum value if it is lower. This is useful to prevent values from approaching negative infinity.

By default, no minimum value is used.

inline void SetMinValue(const vtkm::FloatDefault &value)

Specifies the minimum value to take the logarithm of.

Before taking the logarithm, this filter will check the value to this minimum value and clamp it to the minimum value if it is lower. This is useful to prevent values from approaching negative infinity.

By default, no minimum value is used.

## 10.7.6 Point Elevation

The `vtkm::filter::field_transform::PointElevation` filter computes the “elevation” of a field of point coordinates in space. [Example 1](#) gives a demonstration of the elevation filter.

class **PointElevation** : public vtkm::filter::Filter

Generate a scalar field along a specified direction.

The filter will take a data set and a field of 3 dimensional vectors and compute the distance along a line defined by a low point and a high point. Any point in the plane touching the low point and perpendicular to the line is set to the minimum range value in the elevation whereas any point in the plane touching the high point and perpendicular to the line is set to the maximum range value. All other values are interpolated linearly between these two planes. This filter is commonly used to compute the elevation of points in some direction, but can be repurposed for a variety of measures.

The default name for the output field is ‘elevation’, but that can be overridden as always using the `SetOutputFieldName()` method.

## Public Functions

inline void **SetLowPoint**(const vtkm::Vec3f\_64 &point)

Specify the coordinate of the low point.

The plane of low values is defined by the plane that contains the low point and is normal to the direction from the low point to the high point. All vector values on this plane are assigned the low value.

inline void **SetLowPoint**(vtkm::Float64 x, vtkm::Float64 y, vtkm::Float64 z)

Specify the coordinate of the low point.

The plane of low values is defined by the plane that contains the low point and is normal to the direction from the low point to the high point. All vector values on this plane are assigned the low value.

inline void **SetHighPoint**(const vtkm::Vec3f\_64 &point)

Specify the coordinate of the high point.

The plane of high values is defined by the plane that contains the high point and is normal to the direction from the low point to the high point. All vector values on this plane are assigned the high value.

inline void **SetHighPoint**(vtkm::Float64 x, vtkm::Float64 y, vtkm::Float64 z)

Specify the coordinate of the high point.

The plane of high values is defined by the plane that contains the high point and is normal to the direction from the low point to the high point. All vector values on this plane are assigned the high value.

inline void **SetRange**(vtkm::Float64 low, vtkm::Float64 high)

Specify the range of values to output.

Values at the low plane are given **low** and values at the high plane are given **high**. Values in between the planes have a linearly interpolated value based on the relative distance between the two planes.

## 10.7.7 Point Transform

The `vtkm::filter::field_transform::PointTransform` filter performs affine transforms is the point transform filter.

class **PointTransform** : public vtkm::filter::Filter

Perform affine transforms to point coordinates or vector fields.

This filter will take a data set and a field of 3 dimensional vectors and perform the specified point transform operation. Several methods are provided to apply many common affine transformations (e.g., translation, rotation, and scale). You can also provide a general 4x4 transformation matrix with `SetTransform()`.

The main use case for `PointTransform` is to perform transformations of objects in 3D space, which is done by applying these transforms to the coordinate system. This filter will operate on the `vtkm::cont::CoordinateSystem` of the input data unless a different active field is specified. Likewise, this filter will save its results as the first coordinate system in the output unless `SetChangeCoordinateSystem()` is set to say otherwise.

The default name for the output field is “transform”, but that can be overridden as always using the `SetOutputFieldName()` method.

## Public Functions

inline void **SetTranslation**(const vtkm::FloatDefault &tx, const vtkm::FloatDefault &ty, const vtkm::FloatDefault &tz)

Translates, or moves, each point in the input field by a given direction.

inline void **SetTranslation**(const vtkm::Vec3f &v)

Translates, or moves, each point in the input field by a given direction.

inline void **SetRotation**(const vtkm::FloatDefault &angleDegrees, const vtkm::Vec3f &axis)

Rotate the input field about a given axis.

### Parameters

- **angleDegrees** – [in] The amount of rotation to perform, given in degrees.
- **axis** – [in] The rotation is made around a line that goes through the origin and pointing in this direction in the counterclockwise direction.

inline void **SetRotation**(const vtkm::FloatDefault &angleDegrees, const vtkm::FloatDefault &axisX, const vtkm::FloatDefault &axisY, const vtkm::FloatDefault &axisZ)

Rotate the input field about a given axis.

The rotation is made around a line that goes through the origin and pointing in the direction specified by axisX, axisY, and axisZ in the counterclockwise direction.

### Parameters

- **angleDegrees** – [in] The amount of rotation to perform, given in degrees.
- **axisX** – [in] The X value of the rotation axis.
- **axisY** – [in] The Y value of the rotation axis.
- **axisZ** – [in] The Z value of the rotation axis.

inline void **SetRotationX**(const vtkm::FloatDefault &angleDegrees)

Rotate the input field around the X axis by the given degrees.

inline void **SetRotationY**(const vtkm::FloatDefault &angleDegrees)

Rotate the input field around the Y axis by the given degrees.

inline void **SetRotationZ**(const vtkm::FloatDefault &angleDegrees)

Rotate the input field around the Z axis by the given degrees.

inline void **SetScale**(const vtkm::FloatDefault &s)

Scale the input field.

Each coordinate is multiplied by the associated scale factor.

inline void **SetScale**(const vtkm::FloatDefault &sx, const vtkm::FloatDefault &sy, const vtkm::FloatDefault &sz)

Scale the input field.

Each coordinate is multiplied by the associated scale factor.

inline void **SetScale**(const vtkm::Vec3f &v)

Scale the input field.

Each coordinate is multiplied by the associated scale factor.

```
inline void SetTransform(const vtkm::Matrix<vtkm::FloatDefault, 4, 4> &mtx)
```

Set a general transformation matrix.

Each field value is multiplied by this 4x4 as a homogeneous coordinate. That is a 1 component is added to the end of each 3D vector to put it in the form [x, y, z, 1]. The matrix is then premultiplied to this as a column vector.

The functions in `vtkm/Transform3D.h` can be used to help build these transform matrices.

```
void SetChangeCoordinateSystem(bool flag)
```

Specify whether the result should become the coordinate system of the output.

When this flag is on (the default) the first coordinate system in the output `vtkm::cont::DataSet` is set to the transformed point coordinates.

## 10.7.8 Spherical Coordinate System Transform

The `vtkm::filter::field_transform::SphericalCoordinateTransform` filter is a coordinate system transformation. The filter will take a data set and transform the points of the coordinate system. By default, the filter will transform the coordinates from a Cartesian coordinate system to a spherical coordinate system. The order for spherical coordinates is  $(R, \theta, \phi)$  where  $R$  is the radius,  $\theta$  is the azimuthal angle and  $\phi$  is the polar angle. The output coordinate system will be set to the new computed coordinates.

```
class SphericalCoordinateTransform : public vtkm::filter::Filter
```

Transform coordinates between Cartesian and spherical.

By default, this filter will transform the first coordinate system, but this can be changed by setting the active field.

The resulting transformation will be set as the first coordinate system in the output.

### Public Functions

```
inline void SetCartesianToSpherical()
```

Establish a transformation from Cartesian to spherical coordinates.

```
inline void SetSphericalToCartesian()
```

Establish a transformation from spherical to Cartesian coordinates.

## 10.7.9 Warp

The `vtkm::filter::field_transform::Warp` filter modifies points in a `vtkm::cont::DataSet` by moving points along scaled direction vectors. By default, the `vtkm::filter::field_transform::Warp` filter modifies the coordinate system and writes its results to the coordinate system. A vector field can be selected as directions, or a constant direction can be specified. A constant direction is particularly useful for generating a carpet plot. A scalar field can be selected to scale the displacement, and a constant scale factor adjustment can be specified.

```
class Warp : public vtkm::filter::Filter
```

Modify points by moving points along scaled direction vectors.

This filter displaces the point coordinates of a dataset either in the direction of a direction vector field or in a constant direction.

The filter starts with a set of point coordinates or other vectors. By default these vectors are the coordinate system, but they can be changed by modifying active field 0. These vectors are then displaced by a set of vectors.



This is done by selecting a field of directions, a field of scales, and an additional scale factor. The directions are multiplied by the scale field and the scale factor, and this displacement is added to the vector.

It is common to wish to warp in a constant direction by a scaled amount. To support this so called “WarpScalar”, the *Warp* filter allows you to specify a constant direction with the *SetConstantDirection()* method. When this is set, no direction field is retrieved. By default *Warp* uses (0, 0, 1) as the direction.

It is also common to wish to simply apply a vector direction field (with a possible constant scale). To support this so called “WarpVector”, the *Warp* filter allows you to ignore the scale field with the *SetUseScaleField()* method. When this is unset, no scale field is retrieved. Calling *SetScaleField()* turns on the UseScaleField flag. By default, *Warp* uses will not use the scale field unless specified.

The main use case for *Warp* is to adjust the spatial location and shape of objects in 3D space. This filter will operate on the *vtkm::cont::CoordinateSystem* of the input data unless a different active field is specified. Likewise, this filter will save its results as the first coordinate system in the output unless *SetChangeCoordinateSystem()* is set to say otherwise.

Subclassed by *vtkm::filter::field\_transform::WarpScalar*, *vtkm::filter::field\_transform::WarpVector*

## Public Functions

inline void **SetDirectionField**(const std::string &name)

Specify a field to use as the directions.

The directions, when not set to use constant directions, are set as active field index 1.

inline std::string **GetDirectionFieldName**() const

Specify a field to use as the directions.

The directions, when not set to use constant directions, are set as active field index 1.

inline void **SetConstantDirection**(const vtkm::Vec3f &direction)

Specify a constant value to use as the directions.

This will provide a (constant) direction of the direction, and the direction field will be ignored.

inline const vtkm::Vec3f &**GetConstantDirection**() const

Specify a constant value to use as the directions.

This will provide a (constant) direction of the direction, and the direction field will be ignored.

inline void **SetUseConstantDirection**(bool flag)

Specifies whether a direction field or a constant direction is used.

When true, the constant direction is used. When false, the direction field (active field index 1) is used.

inline bool **GetUseConstantDirection**() const

Specifies whether a direction field or a constant direction is used.

When true, the constant direction is used. When false, the direction field (active field index 1) is used.

inline void **SetScaleField**(const std::string &name)

Specify a field to use to scale the directions.

The scale factor field scales the size of the direction. The scale factor, when not set to use a constant factor, is set as active field index 2.

inline std::string **GetScaleFieldName**() const

Specify a field to use to scale the directions.

The scale factor field scales the size of the direction. The scale factor, when not set to use a constant factor, is set as active field index 2.

inline void **SetUseScaleField**(bool flag)

Specifies whether a scale factor field is used.

When true, a scale factor field the constant scale factor is used. When false, the scale factor field (active field index 2) is used.

inline bool **GetUseScaleField**() const

Specifies whether a scale factor field is used.

When true, a scale factor field the constant scale factor is used. When false, the scale factor field (active field index 2) is used.

inline void **SetScaleFactor**(vtkm::FloatDefault scale)

Specifies an additional scale factor to scale the displacements.

When using a non-constant scale field, it is possible that the scale field is of the wrong units and needs to be rescaled. This scale factor is multiplied to the direction and scale to re-adjust the overall scale.

inline vtkm::FloatDefault **GetScaleFactor**() const

Specifies an additional scale factor to scale the displacements.

When using a non-constant scale field, it is possible that the scale field is of the wrong units and needs to be rescaled. This scale factor is multiplied to the direction and scale to re-adjust the overall scale.

inline void **SetChangeCoordinateSystem**(bool flag)

Specify whether the result should become the coordinate system of the output.

When this flag is on (the default) the first coordinate system in the output `vtkm::cont::DataSet` is set to the transformed point coordinates.

inline bool **GetChangeCoordinateSystem**() const

Specify whether the result should become the coordinate system of the output.

When this flag is on (the default) the first coordinate system in the output `vtkm::cont::DataSet` is set to the transformed point coordinates.

## 10.8 Flow Analysis

Flow visualization is used to analyze vector fields that represent the movement of a fluid. The basic operation of most flow visualization algorithms is particle advection, which traces the path a particle would take given the direction and speed dictated by the vector field. There are multiple ways in which to represent flow in this manner, and consequently VTK-m contains several filters that trace streams in different ways. These filters inherit from `vtkm::filter::flow::FilterParticleAdvection`, which provides several important methods.

class **FilterParticleAdvection** : public vtkm::filter::Filter

base class for advecting particles in a vector field.

Takes as input a vector field and seed locations and advects the seeds through the flow field.

Subclassed	by	vtkm::filter::flow::FilterParticleAdvectionSteadyState<	ParticleAd-
vection	> ,	vtkm::filter::flow::FilterParticleAdvectionSteadyState<	WarpXStream-
line	> ,	vtkm::filter::flow::FilterParticleAdvectionSteadyState<	Streamline > ,

```

vtkm::filter::flow::FilterParticleAdvectionUnsteadyState< PathParticle >, vtkm::filter::flow::FilterParticleAdvectionUnsteadyState<
Pathline >, vtkm::filter::flow::FilterParticleAdvectionSteadyState< Derived >,
vtkm::filter::flow::FilterParticleAdvectionUnsteadyState< Derived >

```

## Public Functions

inline virtual bool **CanThread**() const override

Returns whether the filter can execute on partitions in concurrent threads.

If a derived class's implementation of `DoExecute` cannot run on multiple threads, then the derived class should override this method to return false.

inline void **SetStepSize**(vtkm::FloatDefault s)

Specifies the step size used for the numerical integrator.

The numerical integrators operate by advancing each particle by a finite amount. This parameter defines the distance to advance each time. Smaller values are more accurate but take longer to integrate. An appropriate step size is usually around the size of each cell.

inline void **SetNumberOfSteps**(vtkm::Id n)

Specifies the maximum number of integration steps for each particle.

Some particle paths may loop and continue indefinitely. This parameter sets an upper limit on the total length of advection.

template<typename **ParticleType**>

inline void **SetSeeds**(vtkm::cont::ArrayHandle<ParticleType> &seeds)

Specify the seed locations for the particle advection.

Each seed represents one particle that is advected by the vector field. The particles are represented by a `vtkm::Particle` object or similar type of object (such as `vtkm::ChargedParticle`).

template<typename **ParticleType**>

inline void **SetSeeds**(const std::vector<ParticleType> &seeds, vtkm::CopyFlag copyFlag =  
vtkm::CopyFlag::On)

Specify the seed locations for the particle advection.

Each seed represents one particle that is advected by the vector field. The particles are represented by a `vtkm::Particle` object or similar type of object (such as `vtkm::ChargedParticle`).

Flow filters operate either on a “steady state” flow that does not change or on an “unsteady state” flow that is continually changing over time. An unsteady state filter must be executed multiple times for subsequent time steps. The filter operates with data from two adjacent time steps. This is managed by the `vtkm::filter::flow::FilterParticleAdvectionUnsteadyState` superclass.

### 10.8.1 Streamlines

*Streamlines* are a powerful technique for the visualization of flow fields. A streamline is a curve that is parallel to the velocity vector of the flow field. Individual streamlines are computed from an initial point location (seed) using a numerical method to integrate the point through the flow field.

class **Streamline** : public vtkm::filter::flow::FilterParticleAdvectionSteadyState<Streamline>

Advect particles in a vector field and display the path they take.

This filter takes as input a velocity vector field and seed locations. It then traces the path each seed point would take if moving at the velocity specified by the field. Mathematically, this is the curve that is tangent to the velocity field everywhere.

The output of this filter is a `vtkm::cont::DataSet` containing a collection of poly-lines representing the paths the seed particles take.

The `vtkm::filter::flow::Streamline` filter also uses several inherited methods: `vtkm::filter::flow::FilterParticleAdvection::SetSeeds()`, `vtkm::filter::flow::FilterParticleAdvection::SetNumberOfSteps()`, and `vtkm::filter::flow::FilterParticleAdvection::SetNumberOfSteps()`.

Example 4: Using `vtkm::filter::flow::Streamline`.

```

1  vtkm::filter::flow::Streamline streamlines;
2
3  // Specify the seeds.
4  vtkm::cont::ArrayHandle<vtkm::Particle> seedArray;
5  seedArray.Allocate(2);
6  seedArray.WritePortal().Set(0, vtkm::Particle({ 0, 0, 0 }, 0));
7  seedArray.WritePortal().Set(1, vtkm::Particle({ 1, 1, 1 }, 1));
8
9  streamlines.SetActiveField("vectorvar");
10 streamlines.SetStepSize(0.1f);
11 streamlines.SetNumberOfSteps(100);
12 streamlines.SetSeeds(seedArray);
13
14 vtkm::cont::DataSet output = streamlines.Execute(inData);

```

## 10.8.2 Pathlines

*Pathlines* are the analog to streamlines for time varying vector fields. Individual pathlines are computed from an initial point location (seed) using a numerical method to integrate the point through the flow field.

This filter requires two data sets as input, which represent the data for two sequential time steps. The “Previous” data set, which marks the data at the earlier time step, is passed into the filter through the standard `Execute` method. The “Next” data set, which marks the data at the later time step, is specified as state to the filter using methods.

class **Pathline** : public `vtkm::filter::flow::FilterParticleAdvectionUnsteadyState<Pathline>`

Advect particles in a time-varying vector field and display the path they take.

This filter takes as input a velocity vector field, changing between two time steps, and seed locations. It then traces the path each seed point would take if moving at the velocity specified by the field.

The output of this filter is a `vtkm::cont::DataSet` containing a collection of poly-lines representing the paths the seed particles take.

As an unsteady state flow filter, `vtkm::filter::flow::Pathline` must be executed multiple times for subsequent time steps. The filter operates with data from two adjacent time steps. This is managed by the `vtkm::filter::flow::FilterParticleAdvectionUnsteadyState` superclass.

The `vtkm::filter::flow::Pathline` filter uses several other inherited methods: `vtkm::filter::flow::FilterParticleAdvectionUnsteadyState::SetPreviousTime()`, `vtkm::filter::flow::FilterParticleAdvectionUnsteadyState::SetNextTime()`, `vtkm::filter::flow::FilterParticleAdvectionUnsteadyState::SetNextDataSet()`, `vtkm::filter::flow::FilterParticleAdvection::SetSeeds()`, `vtkm::filter::flow::FilterParticleAdvection::SetNumberOfSteps()`, and `vtkm::filter::flow::FilterParticleAdvection::SetNumberOfSteps()`.

Example 5: Using `vtkm::filter::flow::Pathline`.

```

1  vtkm::filter::flow::Pathline pathlines;
2
3  // Specify the seeds.
4  vtkm::cont::ArrayHandle<vtkm::Particle> seedArray;
5  seedArray.Allocate(2);
6  seedArray.WritePortal().Set(0, vtkm::Particle({ 0, 0, 0 }, 0));
7  seedArray.WritePortal().Set(1, vtkm::Particle({ 1, 1, 1 }, 1));
8
9  pathlines.SetActiveField("vectorvar");
10 pathlines.SetStepSize(0.1f);
11 pathlines.SetNumberOfSteps(100);
12 pathlines.SetSeeds(seedArray);
13 pathlines.SetPreviousTime(0.0f);
14 pathlines.SetNextTime(1.0f);
15 pathlines.SetNextDataSet(inData2);
16
17 vtkm::cont::DataSet pathlineCurves = pathlines.Execute(inData1);

```

### 10.8.3 Stream Surface

A *stream surface* is defined as a continuous surface that is everywhere tangent to a specified vector field. The `vtkm::filter::flow::StreamSurface` filter computes a stream surface from a set of input points and the vector field of the input data set. The stream surface is created by creating streamlines from each input point and then connecting adjacent streamlines with a series of triangles.

class **StreamSurface** : public `vtkm::filter::Filter`

Generate stream surfaces from a vector field.

This filter takes as input a velocity vector field and seed locations. The seed locations should be arranged in a line or curve. The filter then traces the path each seed point would take if moving at the velocity specified by the field and connects all the lines together into a surface. Mathematically, this is the surface that is tangent to the velocity field everywhere.

The output of this filter is a `vtkm::cont::DataSet` containing a mesh for the created surface.

#### Public Functions

inline void **SetStepSize**(`vtkm::FloatDefault` s)

Specifies the step size used for the numerical integrator.

The numerical integrators operate by advancing each particle by a finite amount. This parameter defines the distance to advance each time. Smaller values are more accurate but take longer to integrate. An appropriate step size is usually around the size of each cell.

inline void **SetNumberOfSteps**(`vtkm::Id` n)

Specifies the maximum number of integration steps for each particle.

Some particle paths may loop and continue indefinitely. This parameter sets an upper limit on the total length of advection.

template<typename **ParticleType**>

```
inline void SetSeeds(vtkm::cont::ArrayHandle<ParticleType> &seeds)
```

Specify the seed locations for the particle advection.

Each seed represents one particle that is advected by the vector field. The particles are represented by a `vtkm::Particle` object.

```
template<typename ParticleType>
```

```
inline void SetSeeds(const std::vector<ParticleType> &seeds, vtkm::CopyFlag copyFlag =  
                    vtkm::CopyFlag::On)
```

Specify the seed locations for the particle advection.

Each seed represents one particle that is advected by the vector field. The particles are represented by a `vtkm::Particle` object.

Example 6: Using `vtkm::filter::flow::StreamSurface`.

```
1  vtkm::filter::flow::StreamSurface streamSurface;  
2  
3  // Specify the seeds.  
4  vtkm::cont::ArrayHandle<vtkm::Particle> seedArray;  
5  seedArray.Allocate(2);  
6  seedArray.WritePortal().Set(0, vtkm::Particle({ 0, 0, 0 }, 0));  
7  seedArray.WritePortal().Set(1, vtkm::Particle({ 1, 1, 1 }, 1));  
8  
9  streamSurface.SetActiveField("vectorvar");  
10 streamSurface.SetStepSize(0.1f);  
11 streamSurface.SetNumberOfSteps(100);  
12 streamSurface.SetSeeds(seedArray);  
13  
14 vtkm::cont::DataSet output = streamSurface.Execute(inData);
```

## 10.8.4 Lagrangian Coherent Structures

Lagrangian coherent structures (LCS) are distinct structures present in a flow field that have a major influence over nearby trajectories over some interval of time. Some of these structures may be sources, sinks, saddles, or vortices in the flow field. Identifying Lagrangian coherent structures is part of advanced flow analysis and is an important part of studying flow fields. These structures can be studied by calculating the finite time Lyapunov exponent (FTLE) for a flow field at various locations, usually over a regular grid encompassing the entire flow field. If the provided input dataset is structured, then by default the points in this data set will be used as seeds for advection. The `vtkm::filter::flow::LagrangianStructures` filter is used to compute the FTLE of a flow field.

```
class LagrangianStructures : public vtkm::filter::Filter
```

Compute the finite time Lyapunov exponent (FTLE) of a vector field.

The FTLE is computed by advecting particles throughout the vector field and analyzing where they diverge or converge. By default, the points of the input `vtkm::cont::DataSet` are all advected for this computation unless an auxiliary grid is established.

## Public Functions

inline virtual bool **CanThread**() const override

Returns whether the filter can execute on partitions in concurrent threads.

If a derived class's implementation of **DoExecute** cannot run on multiple threads, then the derived class should override this method to return false.

inline void **SetStepSize**(vtkm::FloatDefault s)

Specifies the step size used for the numerical integrator.

The numerical integrators operate by advancing each particle by a finite amount. This parameter defines the distance to advance each time. Smaller values are more accurate but take longer to integrate. An appropriate step size is usually around the size of each cell.

inline vtkm::FloatDefault **GetStepSize**()

Specifies the step size used for the numerical integrator.

The numerical integrators operate by advancing each particle by a finite amount. This parameter defines the distance to advance each time. Smaller values are more accurate but take longer to integrate. An appropriate step size is usually around the size of each cell.

inline void **SetNumberOfSteps**(vtkm::Id n)

Specify the maximum number of steps each particle is allowed to traverse.

This can limit the total length of displacements used when computing the FTLE.

inline vtkm::Id **GetNumberOfSteps**()

Specify the maximum number of steps each particle is allowed to traverse.

This can limit the total length of displacements used when computing the FTLE.

inline void **SetAdvectionTime**(vtkm::FloatDefault advectionTime)

Specify the time interval for the advection.

The FTLE works by advecting all points a finite distance, and this parameter specifies how far to advect.

inline vtkm::FloatDefault **GetAdvectionTime**()

Specify the time interval for the advection.

The FTLE works by advecting all points a finite distance, and this parameter specifies how far to advect.

inline void **SetUseAuxiliaryGrid**(bool useAuxiliaryGrid)

Specify whether to use an auxiliary grid.

When this flag is off (the default), then the points of the mesh representing the vector field are advected and used for computing the FTLE. However, if the mesh is too coarse, the FTLE will likely be inaccurate. Or if the mesh is unstructured the FTLE may be less efficient to compute. When this flag is on, an auxiliary grid of uniformly spaced points is used for the FTLE computation.

inline bool **GetUseAuxiliaryGrid**()

Specify whether to use an auxiliary grid.

When this flag is off (the default), then the points of the mesh representing the vector field are advected and used for computing the FTLE. However, if the mesh is too coarse, the FTLE will likely be inaccurate. Or if the mesh is unstructured the FTLE may be less efficient to compute. When this flag is on, an auxiliary grid of uniformly spaced points is used for the FTLE computation.

inline void **SetAuxiliaryGridDimensions**(vtkm::Id3 auxiliaryDims)

Specify the dimensions of the auxiliary grid for FTLE calculation.

Seeds for advection will be placed along the points of this auxiliary grid. This option has no effect unless the UseAuxiliaryGrid option is on.

inline vtkm::Id3 **GetAuxiliaryGridDimensions**()

Specify the dimensions of the auxiliary grid for FTLE calculation.

Seeds for advection will be placed along the points of this auxiliary grid. This option has no effect unless the UseAuxiliaryGrid option is on.

inline void **SetUseFlowMapOutput**(bool useFlowMapOutput)

Specify whether to use flow maps instead of advection.

If the start and end points for FTLE calculation are known already, advection is an unnecessary step. This flag allows users to bypass advection, and instead use a precalculated flow map. By default this option is off.

inline bool **GetUseFlowMapOutput**()

Specify whether to use flow maps instead of advection.

If the start and end points for FTLE calculation are known already, advection is an unnecessary step. This flag allows users to bypass advection, and instead use a precalculated flow map. By default this option is off.

inline void **SetOutputFieldName**(std::string outputFieldName)

Specify the name of the output field in the data set returned.

By default, the field will be named FTLE.

inline std::string **GetOutputFieldName**()

Specify the name of the output field in the data set returned.

By default, the field will be named FTLE.

inline void **SetFlowMapOutput**(vtkm::cont::ArrayHandle<vtkm::Vec3f> &flowMap)

Specify the array representing the flow map output to be used for FTLE calculation.

inline vtkm::cont::ArrayHandle<vtkm::Vec3f> **GetFlowMapOutput**()

Specify the array representing the flow map output to be used for FTLE calculation.

## 10.9 Geometry Refinement

Geometry refinement modifies the geometry of a `vtkm::cont::DataSet`. It might add, change, or remove components of the structure, but the general representation will be the same.



### 10.9.1 Convert to a Point Cloud

Data in a `vtkm::cont::DataSet` is typically connected together by cells in a mesh structure. However, it is sometimes the case where data are simply represented as a cloud of unconnected points. These meshless data sets are best represented in a `vtkm::cont::DataSet` by a collection of “vertex” cells.

The `vtkm::filter::geometry_refinement::ConvertToPointCloud` filter converts a data to a point cloud. It does this by throwing away any existing cell set and replacing it with a collection of vertex cells, one per point. `vtkm::filter::geometry_refinement::ConvertToPointCloud` is useful to add a cell set to a `vtkm::cont::DataSet` that has points but no cells. It is also useful to treat data as a collection of sample points rather than an interconnected mesh.

```
class ConvertToPointCloud : public vtkm::filter::Filter
```

Convert a `DataSet` to a point cloud.

A point cloud in VTK-m is represented as a data set with “vertex” shape cells. This filter replaces the `CellSet` in a `DataSet` with a `CellSet` of only vertex cells. There will be one cell per point.

This filter is useful for dropping the cells of any `DataSet` so that you can operate on it as just a collection of points. It is also handy for completing a `DataSet` that does not have a `CellSet` associated with it or has points that do not belong to cells.

Note that all fields associated with cells are dropped. This is because the cells are dropped.

#### Public Functions

```
inline void SetAssociateFieldsWithCells(bool flag)
```

By default, all the input point fields are kept as point fields in the output.

However, the output has exactly one cell per point and it might be easier to treat the fields as cell fields. When this flag is turned on, the point field association is changed to cell.

Note that any field that is marked as point coordinates will remain as point fields. It is not valid to set a cell field as the point coordinates.

```
inline bool GetAssociateFieldsWithCells() const
```

By default, all the input point fields are kept as point fields in the output.

However, the output has exactly one cell per point and it might be easier to treat the fields as cell fields. When this flag is turned on, the point field association is changed to cell.

Note that any field that is marked as point coordinates will remain as point fields. It is not valid to set a cell field as the point coordinates.

### 10.9.2 Shrink

The `vtkm::filter::geometry_refinement::Shrink` independently reduces the size of each class. Rather than uniformly reduce the size of the whole data set (which can be done with `vtkm::filter::field_transform::PointTransform`), this filter separates the cells from each other and shrinks them around their centroid. This is useful for making an “exploded view” of the data where the facets of the data are moved away from each other to see inside.

```
class Shrink : public vtkm::filter::Filter
```

*Shrink* cells of an arbitrary dataset by a constant factor.

The *Shrink* filter shrinks the cells of a DataSet towards their centroid, computed as the average position of the cell points. This filter disconnects the cells, duplicating the points connected to multiple cells. The resulting CellSet is always an ExplicitCellSet.

### Public Functions

inline void **SetShrinkFactor**(vtkm::FloatDefault factor)

Specify the scale factor to size each cell.

The shrink factor specifies the ratio of the shrunk cell to its original size. This value must be between 0 and 1. A value of 1 is the same size as the input, and a value of 0 shrinks each cell to a point.

inline vtkm::FloatDefault **GetShrinkFactor**() const

Specify the scale factor to size each cell.

The shrink factor specifies the ratio of the shrunk cell to its original size. This value must be between 0 and 1. A value of 1 is the same size as the input, and a value of 0 shrinks each cell to a point.

## 10.9.3 Split Sharp Edges

The *vtkm::filter::geometry\_refinement::SplitSharpEdges* filter splits sharp manifold edges where the feature angle between the adjacent surfaces are larger than a threshold value. This is most useful to preserve sharp edges when otherwise applying smooth shading during rendering.

class **SplitSharpEdges** : public vtkm::filter::Filter

Split sharp polygon mesh edges with a large feature angle between the adjacent cells.

Split sharp manifold edges where the feature angle between the adjacent polygonal cells are larger than a threshold value. The feature angle is the angle between the normals of the two polygons. Two polygons on the same plane have a feature angle of 0. Perpendicular polygons have a feature angle of 90 degrees.

When an edge is split, it adds a new point to the coordinates and updates the connectivity of an adjacent surface. For example, consider two adjacent triangles (0,1,2) and (2,1,3) where edge (1,2) needs to be split. Two new points 4 (duplication of point 1) and 5 (duplication of point 2) would be added and the later triangle's connectivity would be changed to (5,4,3). By default, all old point's fields would be copied to the new point.

Note that “split” edges do not have space added between them. They are still adjacent visually, but the topology becomes disconnected there. Splitting sharp edges is most useful to duplicate normal shading vectors to make a sharp shading effect.

### Public Functions

inline void **SetFeatureAngle**(vtkm::FloatDefault value)

Specify the feature angle threshold to split on.

The feature angle is the angle between the normals of the two polygons. Two polygons on the same plane have a feature angle of 0. Perpendicular polygons have a feature angle of 90 degrees.

Any edge with a feature angle larger than this threshold will be split. The feature angle is specified in degrees. The default value is 30 degrees.

```
inline vtkm::FloatDefault GetFeatureAngle() const
```

Specify the feature angle threshold to split on.

The feature angle is the angle between the normals of the two polygons. Two polygons on the same plane have a feature angle of 0. Perpendicular polygons have a feature angle of 90 degrees.

Any edge with a feature angle larger than this threshold will be split. The feature angle is specified in degrees. The default value is 30 degrees.

## 10.9.4 Tetrahedralize

The `vtkm::filter::geometry_refinement::Tetrahedralize` filter converts all the polyhedra in a `vtkm::cont::DataSet` into tetrahedra.

```
class Tetrahedralize : public vtkm::filter::Filter
```

Convert all polyhedra of a `vtkm::cont::DataSet` into tetrahedra.

Note that although the tetrahedra will occupy the same space of the cells that they replace, the interpolation of point fields within these cells might differ. For example, the first order interpolation of a hexahedron uses trilinear interpolation, which actually results in cubic equations. This differs from the purely linear field in a tetrahedron, so the tetrahedra replacement of the hexahedron will not have exactly the same interpolation.

## 10.9.5 Triangulate

The `vtkm::filter::geometry_refinement::Triangulate` filter converts all the polyhedra in a `vtkm::cont::DataSet` into tetrahedra.

```
class Triangulate : public vtkm::filter::Filter
```

Convert all polygons of a `vtkm::cont::DataSet` into triangles.

Note that although the triangles will occupy the same space of the cells that they replace, the interpolation of point fields within these cells might differ. For example, the first order interpolation of a quadrilateral uses bilinear interpolation, which actually results in quadratic equations. This differs from the purely linear field in a triangle, so the triangle replacement of the quadrilateral will not have exactly the same interpolation.

## 10.9.6 Tube

The `vtkm::filter::geometry_refinement::Tube` filter generates a tube around each line and polyline in the input data set.

```
class Tube : public vtkm::filter::Filter
```

Generate a tube around each line and polyline.

The radius, number of sides, and end capping can be specified for each tube. The orientation of the geometry of the tube are computed automatically using a heuristic to minimize the twisting along the input data set.

## Public Functions

inline void **SetRadius**(vtkm::FloatDefault r)

Specify the radius of each tube.

inline void **SetNumberOfSides**(vtkm::Id n)

Specify the number of sides for each tube.

The tubes are generated using a polygonal approximation. This option determines how many facets will be generated around the tube.

inline void **SetCapping**(bool v)

The *Tube* filter can optionally add a cap at the ends of each tube.

This option specifies whether that cap is generated.

Example 7: Using *vtkm::filter::geometry\_refinement::Tube*.

```
1  vtkm::filter::geometry_refinement::Tube tubeFilter;  
2  
3  tubeFilter.SetRadius(0.5f);  
4  tubeFilter.SetNumberOfSides(7);  
5  tubeFilter.SetCapping(true);  
6  
7  vtkm::cont::DataSet output = tubeFilter.Execute(inData);
```

## 10.9.7 Vertex Clustering

The *vtkm::filter::geometry\_refinement::VertexClustering* filter simplifies a polygonal mesh. It does so by dividing space into a uniform grid of bin and then merges together all points located in the same bin. The smaller the dimensions of this binning grid, the fewer polygons will be in the output cells and the coarser the representation. This surface simplification is an important operation to support level of detail (LOD) rendering in visualization applications.

class **VertexClustering** : public vtkm::filter::Filter

Reduce the number of triangles in a mesh.

*VertexClustering* is a filter to reduce the number of triangles in a triangle mesh, forming a good approximation to the original geometry. The input must be a *vtkm::cont::DataSet* that contains only triangles.

The general approach of the algorithm is to cluster vertices in a uniform binning of space, accumulating to an average point within each bin. In more detail, the algorithm first gets the bounds of the input poly data. It then breaks this bounding volume into a user-specified number of spatial bins. It then reads each triangle from the input and hashes its vertices into these bins. Then, if 2 or more vertices of the triangle fall in the same bin, the triangle is discarded. If the triangle is not discarded, it adds the triangle to the list of output triangles as a list of vertex identifiers. (There is one vertex id per bin.) After all the triangles have been read, the representative vertex for each bin is computed. This determines the spatial location of the vertices of each of the triangles in the output.

To use this filter, specify the divisions defining the spatial subdivision in the x, y, and z directions. Compared to algorithms such as *vtkQuadricClustering*, a significantly higher bin count is recommended as it doesn't increase the computation or memory of the algorithm and will produce significantly better results.

## Public Functions

inline void **SetNumberOfDivisions**(const vtkm::Id3 &num)

Specifies the dimensions of the uniform grid that establishes the bins used for clustering.

Setting smaller numbers of dimensions produces a smaller output, but with a coarser representation of the surface.

inline const vtkm::Id3 &**GetNumberOfDivisions**() const

Specifies the dimensions of the uniform grid that establishes the bins used for clustering.

Setting smaller numbers of dimensions produces a smaller output, but with a coarser representation of the surface.

Example 8: Using `vtkm::filter::geometry_refinement::VertexClustering`.

```
1 vtkm::filter::geometry_refinement::VertexClustering vertexClustering;
2
3 vertexClustering.SetNumberOfDivisions(vtkm::Id3(128, 128, 128));
4
5 vtkm::cont::DataSet simplifiedSurface = vertexClustering.Execute(originalSurface);
```

## 10.10 Mesh Information

VTK-m provides several filters that derive information about the structure of the geometry. This can be information about the shape of cells or their connections.

### 10.10.1 Cell Size Measurements

The `vtkm::filter::mesh_info::CellMeasures` filter integrates the size of each cell in a mesh and reports the size in a new cell field.

class **CellMeasures** : public vtkm::filter::Filter

Compute the size measure of each cell in a dataset.

`CellMeasures` is a filter that generates a new cell data array (i.e., one value specified per cell) holding the signed measure of the cell or 0 (if measure is not well defined or the cell type is unsupported).

By default, the new cell-data array is named “measure”.

## Public Functions

inline void **SetMeasure**(vtkm::filter::mesh\_info::IntegrationType measure)

Specify the type of integrations to support.

This filter can support integrating the size of 1D elements (arclength measurements), 2D elements (area measurements), and 3D elements (volume measurements). The measures to perform are specified with a `vtkm::filter::mesh_info::IntegrationType`.

By default, the size measure for all types of elements is performed.

```
inline vtkm::filter::mesh_info::IntegrationType GetMeasure() const
```

Specify the type of integrations to support.

This filter can support integrating the size of 1D elements (arclength measurements), 2D elements (area measurements), and 3D elements (volume measurements). The measures to perform are specified with a `vtkm::filter::mesh_info::IntegrationType`.

By default, the size measure for all types of elements is performed.

```
inline void SetMeasureToArcLength()
```

Compute the length of 1D elements.

```
inline void SetMeasureToArea()
```

Compute the area of 2D elements.

```
inline void SetMeasureToVolume()
```

Compute the volume of 3D elements.

```
inline void SetMeasureToAll()
```

Compute the size of all types of elements.

```
inline void SetCellMeasureName(const std::string &name)
```

Specify the name of the field generated.

If not set, `measure` is used.

```
inline const std::string &GetCellMeasureName() const
```

Specify the name of the field generated.

If not set, `measure` is used.

By default, `vtkm::filter::mesh_info::CellMeasures` will compute the measures of all types of cells. It is sometimes desirable to limit the types of cells to measure to prevent the resulting field from mixing values of different units. The appropriate measure to compute can be specified with the `vtkm::filter::mesh_info::IntegrationType` enumeration.

```
enum class vtkm::filter::mesh_info::IntegrationType
```

Specifies over what types of mesh elements *CellMeasures* will operate.

The values of `IntegrationType` may be `|`-ed together to select multiple

*Values:*

enumerator **None**

enumerator **ArcLength**

Compute the length of 1D elements.

enumerator **Area**

Compute the area of 2D elements.

enumerator **Volume**

Compute the volume of 3D elements.

enumerator **AllMeasures**

Compute the size of all types of elements.

### 10.10.2 Ghost Cell Classification

The `vtkm::filter::mesh_info::GhostCellClassify` filter determines which cells should be considered ghost cells in a structured data set. The ghost cells are expected to be on the border.

class **GhostCellClassify** : public `vtkm::filter::Filter`

Determines which cells should be considered ghost cells in a structured data set.

The ghost cells are expected to be on the border. The outer layer of cells are marked as ghost cells and the remainder marked as normal.

This filter generates a new cell-centered field marking the status of each cell. Each entry is set to either `vtkm::CellClassification::Normal` or `vtkm::CellClassification::Ghost`.

#### Public Functions

inline void **SetGhostCellName**(const std::string &fieldName)

Set the name of the output field name.

The output field is also marked as the ghost cell field in the output `vtkm::cont::DataSet`.

inline const std::string &**GetGhostCellName**()

Set the name of the output field name.

The output field is also marked as the ghost cell field in the output `vtkm::cont::DataSet`.

### 10.10.3 Mesh Quality Metrics

VTK-m provides several filters to compute metrics about the mesh quality. These filters produce a new cell field that holds a given metric for the shape of the cell. The metrics for this filter come from the Verdict library, and full mathematical descriptions for each metric can be found in the Verdict documentation (Sandia technical report SAND2007-1751, [https://coreform.com/papers/verdict\\_quality\\_library.pdf](https://coreform.com/papers/verdict_quality_library.pdf)).

class **MeshQualityArea** : public `vtkm::filter::Filter`

Compute the area of each cell.

This only produces values for triangles and quadrilaterals.

#### Public Functions

`vtkm::Float64` **ComputeTotalArea**(const `vtkm::cont::DataSet` &input)

Computes the area of all polygonal cells and returns the total area.

`vtkm::Float64` **ComputeAverageArea**(const `vtkm::cont::DataSet` &input)

Computes the average area of cells.

This method first computes the total area of all cells and then divides that by the number of cells in the dataset.

class **MeshQualityAspectGamma** : public `vtkm::filter::Filter`

For each cell, compute the normalized root-mean-square of the edge lengths.

This only produces values for tetrahedra.

The root-mean-square edge length is normalized to the volume such that the value is 1 for an equilateral tetrahedron. The acceptable range for good quality meshes is considered to be [1, 3]. The normal range of values is [1, FLOAT\_MAX].

class **MeshQualityAspectRatio** : public vtkm::filter::Filter

Compute for each cell the ratio of its longest edge to its circumradius.

This only produces values for triangles, quadrilaterals, tetrahedra, and hexahedra.

An acceptable range of this mesh for a good quality polygon is [1, 1.3], and the acceptable range for a good quality polyhedron is [1, 3]. Normal values for any cell type have the range [1, FLOAT\_MAX].

class **MeshQualityCondition** : public vtkm::filter::Filter

Compute for each cell the condition number of the weighted Jacobian matrix.

This only produces values for triangles, quadrilaterals, and tetrahedra.

The acceptable range of values for a good quality cell is [1, 1.3] for triangles, [1, 4] for quadrilaterals, and [1, 3] for tetrahedra.

class **MeshQualityDiagonalRatio** : public vtkm::filter::Filter

Compute for each cell the ratio of the maximum diagonal to the minimum diagonal.

This only produces values for quadrilaterals and hexahedra.

An acceptable range for a good quality cell is [0.65, 1]. The normal range is [0, 1], but a degenerate cell with no size will have the value of infinity.

class **MeshQualityDimension** : public vtkm::filter::Filter

Compute for each cell a metric specifically designed for Sandia's Pronto code.

This only produces values for hexahedra.

class **MeshQualityJacobian** : public vtkm::filter::Filter

Compute for each cell the minimum determinant of the Jacobian matrix, over corners and cell center.

This only produces values for quadrilaterals, tetrahedra, and hexahedra.

class **MeshQualityMaxAngle** : public vtkm::filter::Filter

Computes the maximum angle within each cell in degrees.

This only produces values for triangles and quadrilaterals.

For a good quality triangle, this value should be in the range [60, 90]. Poorer quality triangles can have a value as high as 180. For a good quality quadrilateral, this value should be in the range [90, 135]. Poorer quality quadrilaterals can have a value as high as 360.

class **MeshQualityMaxDiagonal** : public vtkm::filter::Filter

Computes the maximum diagonal length within each cell in degrees.

This only produces values for hexahedra.

class **MeshQualityMinAngle** : public vtkm::filter::Filter

Computes the minimum angle within each cell in degrees.

This only produces values for triangles and quadrilaterals.



For a good quality triangle, this value should be in the range [30, 60]. Poorer quality triangles can have a value as low as 0. For a good quality quadrilateral, this value should be in the range [45, 90]. Poorer quality quadrilaterals can have a value as low as 0.

class **MeshQualityMinDiagonal** : public vtkm::filter::Filter

Computes the minimal diagonal length within each cell in degrees.

This only produces values for hexahedra.

class **MeshQualityOddy** : public vtkm::filter::Filter

Compute for each cell the maximum deviation of a metric tensor from an identity matrix, over all corners and cell center.

This only produces values for quadrilaterals and hexahedra.

For a good quality quadrilateral or hexahedron, this value should be in the range [0, 0.5]. Poorer quality cells can have unboundedly larger values.

class **MeshQualityRelativeSizeSquared** : public vtkm::filter::Filter

Compute for each cell the ratio of area or volume to the mesh average.

If  $S$  is the size of a cell and  $\text{avg}S$  is the average cell size in the mesh, then let  $R = S/\text{avg}S$ .  $R$  is “normalized” to be in the range [0, 1] by taking the minimum of  $R$  and  $1/R$ . This value is then squared.

This only produces values for triangles, quadrilaterals, tetrahedra, and hexahedra.

For a good quality triangle, the relative sized squared should be in the range [0.25, 1]. For a good quality quadrilateral, it should be in the range [0.3, 1]. For a good quality tetrahedron, it should be in the range [0.3, 1]. For a good quality hexahedron, it should be in the range [0.5, 1]. Poorer quality cells can have a relative size squared as low as 0.

class **MeshQualityScaledJacobian** : public vtkm::filter::Filter

Compute for each cell a metric derived from the Jacobian matrix with normalization involving edge length.

This only produces values for triangles, quadrilaterals, tetrahedra, and hexahedra.

For a triangle, an acceptable range for good quality is  $[0.5, 2 \cdot \sqrt{3}/3]$ . The value for an equilateral triangle is 1. The normal range is  $[-2 \cdot \sqrt{3}/3, 2 \cdot \sqrt{3}/3]$ , but malformed cells can have plus or minus the maximum float value.

For a quadrilateral, an acceptable range for good quality is [0.3, 1]. The unit square has a value of 1. The normal range as well as the full range is [-1, 1].

For a tetrahedron, an acceptable range for good quality is  $[0.5, \sqrt{2}/2]$ . The value for a unit equilateral triangle is 1. The normal range of values is  $[-\sqrt{2}/2, \sqrt{2}/2]$ , but malformed cells can have plus or minus the maximum float value.

For a hexahedron, an acceptable range for good quality is [0.5, 1]. The unit cube has a value of 1. The normal range is [-1, 1], but malformed cells can have a maximum float value.

class **MeshQualityShape** : public vtkm::filter::Filter

Compute a shape-based metric for each cell.

This metric is based on the condition number of the Jacobian matrix.

This only produces values for triangles, quadrilaterals, tetrahedra, and hexahedra.

For good quality triangles, the acceptable range is [0.25, 1]. Good quality quadrilaterals, tetrahedra, hexahedra are in the range [0.3, 1]. Poorer quality cells can have values as low as 0.

class **MeshQualityShapeAndSize** : public vtkm::filter::Filter

Compute a metric for each cell based on the shape scaled by the cell size.

This filter multiplies the values of the shape metric by the relative size squared metric. See [vtkm::filter::mesh\\_info::MeshQualityShape](#) and [vtkm::filter::mesh\\_info::MeshQualityRelativeSizeSquared](#) for details on each of those metrics.

This only produces values for triangles, quadrilaterals, tetrahedra, and hexahedra.

For a good quality cell, this value will be in the range [0.2, 1]. Poorer quality cells can have values as low as 0.

class **MeshQualityShear** : public vtkm::filter::Filter

Compute the shear of each cell.

The shear of a cell is computed by taking the minimum of the Jacobian at each corner divided by the length of the corner's adjacent edges.

This only produces values for quadrilaterals and hexahedra. Good quality cells will have values in the range [0.3, 1]. Poorer quality cells can have values as low as 0.

class **MeshQualitySkew** : public vtkm::filter::Filter

Compute the skew of each cell.

The skew is computed as the dot product between unit vectors in the principal directions. (For 3D objects, the skew is taken as the maximum of all planes.)

This only produces values for quadrilaterals and hexahedra.

Good quality cells will have a skew in the range [0, 0.5]. A unit square or cube will have a skew of 0. Poor quality cells can have a skew up to 1 although a malformed cell might have its skew be infinite.

class **MeshQualityStretch** : public vtkm::filter::Filter

Compute the stretch of each cell.

The stretch of a cell is computed as the ratio of the minimum edge length to the maximum diagonal, normalized for the unit cube. A good quality cell will have a stretch in the range [0.25, 1]. Poorer quality cells can have a stretch as low as 0 although a malformed cell might return a stretch of infinity.

This only produces values for quadrilaterals and hexahedra.

class **MeshQualityTaper** : public vtkm::filter::Filter

Compute the taper of each cell.

The taper of a quadrilateral is computed as the maximum ratio of the cross-derivative with its shortest associated principal axis.

This only produces values for quadrilaterals and hexahedra.

A good quality quadrilateral will have a taper in the range of [0, 0.7]. A good quality hexahedron will have a taper in the range of [0, 0.5]. The unit square or cube will have a taper of 0. Poorer quality cells will have larger values (with no upper limit).

class **MeshQualityVolume** : public vtkm::filter::Filter

Compute the volume each cell.

This only produces values for tetrahedra, pyramids, wedges, and hexahedra.

## Public Functions

`vtkm::Float64 ComputeTotalVolume(const vtkm::cont::DataSet &input)`

Computes the volume of all polyhedral cells and returns the total area.

`vtkm::Float64 ComputeAverageVolume(const vtkm::cont::DataSet &input)`

Computes the average volume of cells.

This method first computes the total volume of all cells and then divides that by the number of cells in the dataset.

class **MeshQualityWarpage** : public `vtkm::filter::Filter`

Compute the flatness of cells.

This only produces values for quadrilaterals. It is defined as the cosine of the minimum dihedral angle formed by the planes intersecting in diagonals (to the fourth power).

This metric will be 1 for a perfectly flat quadrilateral and be lower as the quadrilateral deviates from the plane. A good quality quadrilateral will have a value in the range [0.3, 1]. Poorer quality cells having lower values down to -1, although malformed cells might have an infinite value.

Note that the value of this filter is consistent with the equivalent metric in VisIt, and it differs from the implementation in the Verdict library. The Verdict library returns 1 - value.

The `vtkm::filter::mesh_info::MeshQuality` filter consolidates all of these metrics into a single filter. The metric to compute is selected with the `vtkm::filter::mesh_info::MeshQuality::SetMetric()`.

class **MeshQuality** : public `vtkm::filter::Filter`

Computes the quality of an unstructured cell-based mesh.

The quality is defined in terms of the summary statistics (frequency, mean, variance, min, max) of metrics computed over the mesh cells. One of several different metrics can be specified for a given cell type, and the mesh can consist of one or more different cell types. The resulting mesh quality is stored as one or more new fields in the output dataset of this filter, with a separate field for each cell type. Each field contains the metric summary statistics for the cell type. Summary statistics with all 0 values imply that the specified metric does not support the cell type.

## Public Functions

void **SetMetric**(*CellMetric* metric)

Specify the metric to compute on the mesh.

inline *CellMetric* **GetMetric**() const

Specify the metric to compute on the mesh.

std::string **GetMetricName**() const

Return a string describing the metric selected.

The metric to compute is identified using the `vtkm::filter::mesh_info::CellMetric` enum.

enum class `vtkm::filter::mesh_info::CellMetric`

Values:

enumerator **Area**

Compute the area of each cell.

This only produces values for triangles and quadrilaterals.

enumerator **AspectGamma**

For each cell, compute the normalized root-mean-square of the edge lengths.

This only produces values for tetrahedra.

The root-mean-square edge length is normalized to the volume such that the value is 1 for an equilateral tetrahedron. The acceptable range for good quality meshes is considered to be [1, 3]. The normal range of values is [1, FLOAT\_MAX].

enumerator **AspectRatio**

Compute for each cell the ratio of its longest edge to its circumradius.

This only produces values for triangles, quadrilaterals, tetrahedra, and hexahedra.

An acceptable range of this mesh for a good quality polygon is [1, 1.3], and the acceptable range for a good quality polyhedron is [1, 3]. Normal values for any cell type have the range [1, FLOAT\_MAX].

enumerator **Condition**

Compute for each cell the condition number of the weighted Jacobian matrix.

This only produces values for triangles, quadrilaterals, and tetrahedra.

The acceptable range of values for a good quality cell is [1, 1.3] for triangles, [1, 4] for quadrilaterals, and [1, 3] for tetrahedra.

enumerator **DiagonalRatio**

Compute for each cell the ratio of the maximum diagonal to the minimum diagonal.

This only produces values for quadrilaterals and hexahedra.

An acceptable range for a good quality cell is [0.65, 1]. The normal range is [0, 1], but a degenerate cell with no size will have the value of infinity.

enumerator **Dimension**

Compute for each cell a metric specifically designed for Sandia's Pronto code.

This only produces values for hexahedra.

enumerator **Jacobian**

Compute for each cell the minimum determinant of the Jacobian matrix, over corners and cell center.

This only produces values for quadrilaterals, tetrahedra, and hexahedra.

enumerator **MaxAngle**

Computes the maximum angle within each cell in degrees.

This only produces values for triangles and quadrilaterals.

For a good quality triangle, this value should be in the range [60, 90]. Poorer quality triangles can have a value as high as 180. For a good quality quadrilateral, this value should be in the range [90, 135]. Poorer quality quadrilaterals can have a value as high as 360.

**enumerator MaxDiagonal**

Computes the maximum diagonal length within each cell in degrees.

This only produces values for hexahedra.

**enumerator MinAngle**

Computes the minimum angle within each cell in degrees.

This only produces values for triangles and quadrilaterals.

For a good quality triangle, this value should be in the range [30, 60]. Poorer quality triangles can have a value as low as 0. For a good quality quadrilateral, this value should be in the range [45, 90]. Poorer quality quadrilaterals can have a value as low as 0.

**enumerator MinDiagonal**

Computes the minimal diagonal length within each cell in degrees.

This only produces values for hexahedra.

**enumerator Oddy**

Compute for each cell the maximum deviation of a metric tensor from an identity matrix, over all corners and cell center.

This only produces values for quadrilaterals and hexahedra.

For a good quality quadrilateral or hexahedron, this value should be in the range [0, 0.5]. Poorer quality cells can have unboundedly larger values.

**enumerator RelativeSizeSquared**

Compute for each cell the ratio of area or volume to the mesh average.

If  $S$  is the size of a cell and  $\text{avgS}$  is the average cell size in the mesh, then let  $R = S/\text{avgS}$ .  $R$  is “normalized” to be in the range [0, 1] by taking the minimum of  $R$  and  $1/R$ . This value is then squared.

This only produces values for triangles, quadrilaterals, tetrahedra, and hexahedra.

For a good quality triangle, the relative sized squared should be in the range [0.25, 1]. For a good quality quadrilateral, it should be in the range [0.3, 1]. For a good quality tetrahedron, it should be in the range [0.3, 1]. For a good quality hexahedron, it should be in the range [0.5, 1]. Poorer quality cells can have a relative size squared as low as 0.

**enumerator ScaledJacobian**

Compute for each cell a metric derived from the Jacobian matrix with normalization involving edge length.

This only produces values for triangles, quadrilaterals, tetrahedra, and hexahedra.

For a triangle, an acceptable range for good quality is  $[0.5, 2\sqrt{3}/3]$ . The value for an equilateral triangle is 1. The normal range is  $[-2\sqrt{3}/3, 2\sqrt{3}/3]$ , but malformed cells can have plus or minus the maximum float value.

For a quadrilateral, an acceptable range for good quality is [0.3, 1]. The unit square has a value of 1. The normal range as well as the full range is [-1, 1].

For a tetrahedron, an acceptable range for good quality is  $[0.5, \sqrt{2}/2]$ . The value for a unit equilateral triangle is 1. The normal range of values is  $[-\sqrt{2}/2, \sqrt{2}/2]$ , but malformed cells can have plus or minus the maximum float value.

For a hexahedron, an acceptable range for good quality is [0.5, 1]. The unit cube has a value of 1. The normal range is [ -1, 1 ], but malformed cells can have a maximum float value.

enumerator **Shape**

Compute a shape-based metric for each cell.

This metric is based on the condition number of the Jacobian matrix.

This only produces values for triangles, quadrilaterals, tetrahedra, and hexahedra.

For good quality triangles, the acceptable range is [0.25, 1]. Good quality quadrilaterals, tetrahedra, hexahedra are in the range [0.3, 1]. Poorer quality cells can have values as low as 0.

enumerator **ShapeAndSize**

Compute a metric for each cell based on the shape scaled by the cell size.

This filter multiplies the values of the shape metric by the relative size squared metric. See [`vtkm::filter::mesh\_info::MeshQualityShape`](#) and [`vtkm::filter::mesh\_info::MeshQualityRelativeSizeSquared`](#) for details on each of those metrics.

This only produces values for triangles, quadrilaterals, tetrahedra, and hexahedra.

For a good quality cell, this value will be in the range [0.2, 1]. Poorer quality cells can have values as low as 0.

enumerator **Shear**

Compute the shear of each cell.

The shear of a cell is computed by taking the minimum of the Jacobian at each corner divided by the length of the corner's adjacent edges.

This only produces values for quadrilaterals and hexahedra. Good quality cells will have values in the range [0.3, 1]. Poorer quality cells can have values as low as 0.

enumerator **Skew**

Compute the skew of each cell.

The skew is computed as the dot product between unit vectors in the principal directions. (For 3D objects, the skew is taken as the maximum of all planes.)

This only produces values for quadrilaterals and hexahedra.

Good quality cells will have a skew in the range [0, 0.5]. A unit square or cube will have a skew of 0. Poor quality cells can have a skew up to 1 although a malformed cell might have its skew be infinite.

enumerator **Stretch**

Compute the stretch of each cell.

The stretch of a cell is computed as the ratio of the minimum edge length to the maximum diagonal, normalized for the unit cube. A good quality cell will have a stretch in the range [0.25, 1]. Poorer quality cells can have a stretch as low as 0 although a malformed cell might return a stretch of infinity.

This only produces values for quadrilaterals and hexahedra.

enumerator **Taper**

Compute the taper of each cell.

The taper of a quadrilateral is computed as the maximum ratio of the cross-derivative with its shortest associated principal axis.

This only produces values for quadrilaterals and hexahedra.

A good quality quadrilateral will have a taper in the range of [0, 0.7]. A good quality hexahedron will have a taper in the range of [0, 0.5]. The unit square or cube will have a taper of 0. Poorer quality cells will have larger values (with no upper limit).

enumerator **Volume**

Compute the volume each cell.

This only produces values for tetrahedra, pyramids, wedges, and hexahedra.

enumerator **Warpage**

Compute the flatness of cells.

This only produces values for quadrilaterals. It is defined as the cosine of the minimum dihedral angle formed by the planes intersecting in diagonals (to the fourth power).

This metric will be 1 for a perfectly flat quadrilateral and be lower as the quadrilateral deviates from the plane. A good quality quadrilateral will have a value in the range [0.3, 1]. Poorer quality cells having lower values down to -1, although malformed cells might have an infinite value.

Note that the value of this filter is consistent with the equivalent metric in VisIt, and it differs from the implementation in the Verdict library. The Verdict library returns 1 - value.

enumerator **None**

## 10.11 Multi-Block

Data with multiple blocks are stored in `vtkm::cont::PartitionedDataSet` objects. Most VTK-m filters operate correctly on `vtkm::cont::PartitionedDataSet` just like they do with `vtkm::cont::DataSet`. However, there are some filters that are designed with operations specific to multi-block datasets.

### 10.11.1 AMR Arrays

An AMR mesh is a `vtkm::cont::PartitionedDataSet` with a special structure in the partitions. Each partition has a `vtkm::cont::CellSetStructured` cell set. The partitions form a hierarchy of grids where each level of the hierarchy refines the one above.

`vtkm::cont::PartitionedDataSet` does not explicitly store the structure of an AMR grid. The `vtkm::filter::multi_block::AmrArrays` filter determines the hierarchical structure of the AMR partitions and stores information about them in cell field arrays on each partition.

class **AmrArrays** : public `vtkm::filter::Filter`

Generate arrays describing the AMR structure in a partitioned data set.

AMR grids are represented by `vtkm::cont::PartitionedDataSet`, but this class does not explicitly store the hierarchical structure of the mesh refinement. This hierarchical arrangement needs to be captured in fields

that describe where blocks reside in the hierarchy. This filter analyses the arrangement of partitions in a `vtkm::cont::PartitionedDataSet` and generates the following field arrays.

- `vtkAmrLevel` The AMR level at which the partition resides (with 0 being the most coarse level). All the values for a particular partition are set to the same value.
- `vtkAmrIndex` A unique identifier for each partition of a particular level. Each partition of the same level will have a unique index, but the indices will repeat across levels. All the values for a particular partition are set to the same value.
- `vtkCompositeIndex` A unique identifier for each partition. This index is the same as the index used for the partition in the containing `vtkm::cont::PartitionedDataSet`. All the values for a particular partition are set to the same value.
- `vtkGhostType` It is common for refinement levels in an AMR structure to overlap more coarse grids. In this case, the overlapped coarse cells have invalid data. The `vtkGhostType` field will track which cells are overlapped and should be ignored. This array will have a 0 value for all valid cells and a non-zero value for all invalid cells. (Specifically, if the bit specified by `vtkm::CellClassification::BLANKED` is set, then the cell is overlapped with a cell in a finer level.)

These arrays are stored as cell fields in the partitions.

This filter only operates on partitioned data sets where all the partitions have cell sets of type `vtkm::cont::CellSetStructured`. This is characteristic of AMR data sets.

---

### Did You Know?

The names of the generated field arrays (e.g. `vtkAmrLevel`) are chosen to be compatible with the equivalent arrays in VTK. This is why they use the prefix of “vtk” instead of “vtkm”. Likewise, the flags used for `vtkGhostType` are compatible with VTK.

---

## 10.11.2 Merge Data Sets

A `vtkm::cont::PartitionedDataSet` can often be treated the same as a `vtkm::cont::DataSet` as both can be passed to a filter's `Execute` method. However, it is sometimes important to have all the data contained in a single `DataSet`. The `vtkm::filter::multi_block::MergeDataSets` filter can do just that to the partitions of a `vtkm::cont::PartitionedDataSet`.

```
class MergeDataSets : public vtkm::filter::Filter
```

Merging multiple data sets into one data set.

This filter merges multiple data sets into one data set. We assume that the input data sets have the same coordinate system. If there are missing fields in a specific data set, the filter uses the `InvalidValue` specified by the user to fill in the associated position of the field array.

`MergeDataSets` is used by passing a `vtkm::cont::PartitionedDataSet` to its `Execute()` method. The `Execute()` will return a `vtkm::cont::PartitionedDataSet` because that is the common interface for all filters. However, the `vtkm::cont::PartitionedDataSet` will have one partition that is all the blocks merged together.



## Public Functions

inline void **SetInvalidValue**(vtkm::Float64 invalidValue)

Specify the value to use where field values are missing.

One issue when merging blocks in a partitioned dataset is that the blocks/partitions may have different fields. That is, one partition might not have all the fields of another partition. When these partitions are merged together, the values for this missing field must be set to something. They will be set to this value, which defaults to NaN.

inline vtkm::Float64 **GetInvalidValue**()

Specify the value to use where field values are missing.

One issue when merging blocks in a partitioned dataset is that the blocks/partitions may have different fields. That is, one partition might not have all the fields of another partition. When these partitions are merged together, the values for this missing field must be set to something. They will be set to this value, which defaults to NaN.

## 10.12 Resampling

All data in `vtkm::cont::DataSet` objects are discrete representations. It is sometimes necessary to resample this data in different ways.

### 10.12.1 Histogram Sampling

The `vtkm::filter::resampling::HistSampling` filter randomly samples the points of an input data set. The sampling is random but adaptive to preserve rare field value points.

class **HistSampling** : public vtkm::filter::Filter

Adaptively sample points to preserve tail features.

This filter randomly samples the points of a `vtkm::cont::DataSet` and generates a new `vtkm::cont::DataSet` with a subsampling of the points. The sampling is adaptively selected to preserve tail and outlying features of the active field. That is, the more rare a field value is, the more likely the point will be selected in the sampling. This is done by creating a histogram of the field and using that to derive the importance level of each field value. Details of the algorithm can be found in the paper “In Situ Data-Driven Adaptive Sampling

for Large-scale Simulation Data Summarization” by Biswas, Dutta, Pulido, and Ahrens as published in *In Situ Infrastructures for Enabling Extreme-scale Analysis and Visualization* (ISAV 2018).

The cell set of the input data is removed and replaced with a set with a vertex cell for each point. This effectively converts the data to a point cloud.

## Public Functions

inline void **SetNumberOfBins**(vtkm::Id numberOfBins)

Specify the number of bins used when computing the histogram.

The histogram is used to select the importance of each field value. More rare field values are more likely to be selected.

inline vtkm::Id **GetNumberOfBins**()

Specify the number of bins used when computing the histogram.

The histogram is used to select the importance of each field value. More rare field values are more likely to be selected.

inline void **SetSampleFraction**(vtkm::FloatDefault fraction)

Specify the fraction of points to create in the sampled data.

A fraction of 1 means that all the points will be sampled and be in the output. A fraction of 0 means that none of the points will be sampled. A fraction of 0.5 means that half the points will be selected to be in the output.

inline vtkm::FloatDefault **GetSampleFraction**() const

Specify the fraction of points to create in the sampled data.

A fraction of 1 means that all the points will be sampled and be in the output. A fraction of 0 means that none of the points will be sampled. A fraction of 0.5 means that half the points will be selected to be in the output.

inline void **SetSeed**(vtkm::UInt32 seed)

Specify the seed used for random number generation.

The random numbers are used to select which points to pull from the input. If the same seed is used for multiple invocations, the results will be the same.

inline vtkm::UInt32 **GetSeed**()

Specify the seed used for random number generation.

The random numbers are used to select which points to pull from the input. If the same seed is used for multiple invocations, the results will be the same.

## 10.12.2 Probe

The `vtkm::filter::resampling::Probe` filter maps the fields of one `vtkm::cont::DataSet` onto another. This is useful for redefining meshes as well as comparing field data from two data sets with different geometries.

class **Probe** : public vtkm::filter::Filter

Sample the fields of a data set at specified locations.

The `vtkm::filter::resampling::Probe` filter samples the fields of one `vtkm::cont::DataSet` and places them in the fields of another `vtkm::cont::DataSet`.

To use this filter, first specify a geometry to probe with with `SetGeometry()`. The most important feature of this geometry is its coordinate system. When you call `Execute()`, the output will be the data specified with `SetGeometry()` but will have the fields of the input to `Execute()` transferred to it. The fields are transferred by probing the input data set at the point locations of the geometry.

## Public Functions

inline void **SetGeometry**(const vtkm::cont::DataSet &geometry)

Specify the geometry to probe with.

When *Execute()* is called, the input data will be probed at all the point locations of this geometry as specified by its coordinate system.

inline const vtkm::cont::DataSet &**GetGeometry**() const

Specify the geometry to probe with.

When *Execute()* is called, the input data will be probed at all the point locations of this geometry as specified by its coordinate system.

inline void **SetInvalidValue**(vtkm::Float64 invalidValue)

Specify the value to use for points outside the bounds of the input.

It is possible that the sampling geometry will have points outside the bounds of the input. When this happens, the field will be set to this “invalid” value. By default, the invalid value is NaN.

inline vtkm::Float64 **GetInvalidValue**() const

Specify the value to use for points outside the bounds of the input.

It is possible that the sampling geometry will have points outside the bounds of the input. When this happens, the field will be set to this “invalid” value. By default, the invalid value is NaN.

## 10.13 Vector Analysis

VTK-m’s vector analysis filters compute operations on fields related to vectors (usually in 3-space).

### 10.13.1 Cross Product

The *vtkm::filter::vector\_analysis::CrossProduct* filter computes the cross product of two vector fields for every element in the input data set. The cross product filter computes (PrimaryField × SecondaryField). The cross product computation works for either point or cell centered vector fields.

class **CrossProduct** : public vtkm::filter::Filter

Compute the cross product of 3D vector fields.

The left part of the operand is the “primary” field and the right part of the operand is the “secondary” field.

## Public Functions

inline void **SetPrimaryField**(const std::string &name, vtkm::cont::Field::Association association = vtkm::cont::Field::Association::Any)

Specify the primary field to operate on.

In the cross product operation  $A \times B$ , A is the primary field.

The primary field is an alias for active field index 0. As with any active field, it can be set as a named field or as a coordinate system.

inline const std::string &**GetPrimaryFieldName**() const

Specify the primary field to operate on.

In the cross product operation  $A \times B$ , A is the primary field.

The primary field is an alias for active field index 0. As with any active field, it can be set as a named field or as a coordinate system.

inline vtkm::cont::Field::Association **GetPrimaryFieldAssociation**() const

Specify the primary field to operate on.

In the cross product operation  $A \times B$ , A is the primary field.

The primary field is an alias for active field index 0. As with any active field, it can be set as a named field or as a coordinate system.

inline void **SetUseCoordinateSystemAsPrimaryField**(bool flag)

Specify the primary field to operate on.

In the cross product operation  $A \times B$ , A is the primary field.

The primary field is an alias for active field index 0. As with any active field, it can be set as a named field or as a coordinate system.

inline bool **GetUseCoordinateSystemAsPrimaryField**() const

Specify the primary field to operate on.

In the cross product operation  $A \times B$ , A is the primary field.

The primary field is an alias for active field index 0. As with any active field, it can be set as a named field or as a coordinate system.

inline void **SetPrimaryCoordinateSystem**(vtkm::Id index)

Specify the primary field to operate on.

In the cross product operation  $A \times B$ , A is the primary field.

The primary field is an alias for active field index 0. As with any active field, it can be set as a named field or as a coordinate system.

inline void **SetSecondaryField**(const std::string &name, vtkm::cont::Field::Association association =  
vtkm::cont::Field::Association::Any)

Specify the secondary field to operate on.

In the cross product operation  $A \times B$ , B is the secondary field.

The secondary field is an alias for the active field index 1. As with any active field, it can be set as a named field or as a coordinate system.

inline const std::string &**GetSecondaryFieldName**() const

Specify the secondary field to operate on.

In the cross product operation  $A \times B$ , B is the secondary field.

The secondary field is an alias for the active field index 1. As with any active field, it can be set as a named field or as a coordinate system.

inline vtkm::cont::Field::Association **GetSecondaryFieldAssociation**() const

Specify the secondary field to operate on.

In the cross product operation  $A \times B$ , B is the secondary field.

The secondary field is an alias for the active field index 1. As with any active field, it can be set as a named field or as a coordinate system.

inline void **SetUseCoordinateSystemAsSecondaryField**(bool flag)

Specify the secondary field to operate on.

In the cross product operation  $A \times B$ , B is the secondary field.

The secondary field is an alias for the active field index 1. As with any active field, it can be set as a named field or as a coordinate system.

inline bool **GetUseCoordinateSystemAsSecondaryField**() const

Specify the secondary field to operate on.

In the cross product operation  $A \times B$ , B is the secondary field.

The secondary field is an alias for the active field index 1. As with any active field, it can be set as a named field or as a coordinate system.

inline void **SetSecondaryCoordinateSystem**(vtkm::Id index)

Specify the secondary field to operate on.

In the cross product operation  $A \times B$ , B is the secondary field.

The secondary field is an alias for the active field index 1. As with any active field, it can be set as a named field or as a coordinate system.

inline vtkm::Id **GetSecondaryCoordinateSystemIndex**() const

Specify the secondary field to operate on.

In the cross product operation  $A \times B$ , B is the secondary field.

The secondary field is an alias for the active field index 1. As with any active field, it can be set as a named field or as a coordinate system.

### 10.13.2 Dot Product

The `vtkm::filter::vector_analysis::DotProduct` filter computes the dot product of two vector fields for every element in the input data set. The dot product filter computes (PrimaryField . SecondaryField). The dot product computation works for either point or cell centered vector fields.

class **DotProduct** : public vtkm::filter::Filter

Compute the dot product of vector fields.

The left part of the operand is the “primary” field and the right part of the operand is the “secondary” field (although the dot product is commutative, so the order of primary and secondary seldom matters).

The dot product can operate on vectors of any length.

#### Public Functions

inline void **SetPrimaryField**(const std::string &name, vtkm::cont::Field::Association association = vtkm::cont::Field::Association::Any)

Specify the primary field to operate on.

In the dot product operation  $A \cdot B$ , A is the primary field.

The primary field is an alias for active field index 0. As with any active field, it can be set as a named field or as a coordinate system.

inline const std::string &**GetPrimaryFieldName**() const

Specify the primary field to operate on.

In the dot product operation  $A \cdot B$ ,  $A$  is the primary field.

The primary field is an alias for active field index 0. As with any active field, it can be set as a named field or as a coordinate system.

inline vtkm::cont::Field::Association **GetPrimaryFieldAssociation**() const

Specify the primary field to operate on.

In the dot product operation  $A \cdot B$ ,  $A$  is the primary field.

The primary field is an alias for active field index 0. As with any active field, it can be set as a named field or as a coordinate system.

inline void **SetUseCoordinateSystemAsPrimaryField**(bool flag)

Specify the primary field to operate on.

In the dot product operation  $A \cdot B$ ,  $A$  is the primary field.

The primary field is an alias for active field index 0. As with any active field, it can be set as a named field or as a coordinate system.

inline bool **GetUseCoordinateSystemAsPrimaryField**() const

Specify the primary field to operate on.

In the dot product operation  $A \cdot B$ ,  $A$  is the primary field.

The primary field is an alias for active field index 0. As with any active field, it can be set as a named field or as a coordinate system.

inline void **SetPrimaryCoordinateSystem**(vtkm::Id coord\_idx)

Specify the primary field to operate on.

In the dot product operation  $A \cdot B$ ,  $A$  is the primary field.

The primary field is an alias for active field index 0. As with any active field, it can be set as a named field or as a coordinate system.

inline vtkm::Id **GetPrimaryCoordinateSystemIndex**() const

Specify the primary field to operate on.

In the dot product operation  $A \cdot B$ ,  $A$  is the primary field.

The primary field is an alias for active field index 0. As with any active field, it can be set as a named field or as a coordinate system.

inline void **SetSecondaryField**(const std::string &name, vtkm::cont::Field::Association association =  
vtkm::cont::Field::Association::Any)

Specify the secondary field to operate on.

In the dot product operation  $A \cdot B$ ,  $B$  is the secondary field.

The secondary field is an alias for active field index 1. As with any active field, it can be set as a named field or as a coordinate system.

inline const std::string &**GetSecondaryFieldName**() const

Specify the secondary field to operate on.

In the dot product operation  $A \cdot B$ ,  $B$  is the secondary field.

The secondary field is an alias for active field index 1. As with any active field, it can be set as a named field or as a coordinate system.

```
inline vtkm::cont::Field::Association GetSecondaryFieldAssociation() const
```

Specify the secondary field to operate on.

In the dot product operation  $A \cdot B$ ,  $B$  is the secondary field.

The secondary field is an alias for active field index 1. As with any active field, it can be set as a named field or as a coordinate system.

```
inline void SetUseCoordinateSystemAsSecondaryField(bool flag)
```

Specify the secondary field to operate on.

In the dot product operation  $A \cdot B$ ,  $B$  is the secondary field.

The secondary field is an alias for active field index 1. As with any active field, it can be set as a named field or as a coordinate system.

```
inline bool GetUseCoordinateSystemAsSecondaryField() const
```

Specify the secondary field to operate on.

In the dot product operation  $A \cdot B$ ,  $B$  is the secondary field.

The secondary field is an alias for active field index 1. As with any active field, it can be set as a named field or as a coordinate system.

```
inline void SetSecondaryCoordinateSystem(vtkm::Id index)
```

Specify the secondary field to operate on.

In the dot product operation  $A \cdot B$ ,  $B$  is the secondary field.

The secondary field is an alias for active field index 1. As with any active field, it can be set as a named field or as a coordinate system.

```
inline vtkm::Id GetSecondaryCoordinateSystemIndex() const
```

Specify the secondary field to operate on.

In the dot product operation  $A \cdot B$ ,  $B$  is the secondary field.

The secondary field is an alias for active field index 1. As with any active field, it can be set as a named field or as a coordinate system.

### 10.13.3 Gradients

The `vtkm::filter::vector_analysis::Gradient` filter estimates the gradient of a point based input field for every element in the input data set. The gradient computation can either generate cell center based gradients, which are fast but less accurate, or more accurate but slower point based gradients. The default for the filter is output as cell centered gradients, but can be changed by using the `vtkm::filter::vector_analysis::Gradient::SetComputePointGradient()` method. The default name for the output fields is “Gradients”, but that can be overridden as always using the `vtkm::filter::vector_analysis::Gradient::SetOutputFieldName()` method.

```
class Gradient : public vtkm::filter::Filter
```

A general filter for gradient estimation.

Estimates the gradient of a point field in a data set. The created gradient array can be determined at either each point location or at the center of each cell.

The default for the filter is output as cell centered gradients. To enable point based gradient computation enable `SetComputePointGradient()`

If no explicit name for the output field is provided the filter will default to “Gradients”

## Public Functions

inline void **SetComputePointGradient**(bool enable)

Specify whether to compute gradients.

When this flag is on (default is off), the gradient filter will provide a point based gradients, which are significantly more costly since for each point we need to compute the gradient of each cell that uses it.

inline bool **GetComputePointGradient**() const

Specify whether to compute gradients.

When this flag is on (default is off), the gradient filter will provide a point based gradients, which are significantly more costly since for each point we need to compute the gradient of each cell that uses it.

inline void **SetComputeDivergence**(bool enable)

Add divergence field to the output data.

The input array must have 3 components to compute this. The default is off.

inline bool **GetComputeDivergence**() const

Add divergence field to the output data.

The input array must have 3 components to compute this. The default is off.

inline void **SetDivergenceName**(const std::string &name)

When [SetComputeDivergence\(\)](#) is enabled, the result is stored in a field of this name.

If not specified, the name of the field will be Divergence.

inline const std::string &**GetDivergenceName**() const

When [SetComputeDivergence\(\)](#) is enabled, the result is stored in a field of this name.

If not specified, the name of the field will be Divergence.

inline void **SetComputeVorticity**(bool enable)

Add vorticity/curl field to the output data.

The input array must have 3 components to compute this. The default is off.

inline bool **GetComputeVorticity**() const

Add vorticity/curl field to the output data.

The input array must have 3 components to compute this. The default is off.

inline void **SetVorticityName**(const std::string &name)

When [SetComputeVorticity\(\)](#) is enabled, the result is stored in a field of this name.

If not specified, the name of the field will be Vorticity.

inline const std::string &**GetVorticityName**() const

When [SetComputeVorticity\(\)](#) is enabled, the result is stored in a field of this name.

If not specified, the name of the field will be Vorticity.

inline void **SetComputeQCriterion**(bool enable)

Add Q-criterion field to the output data.

The input array must have 3 components to compute this. The default is off.



inline bool **GetComputeQCriterion()** const

Add Q-criterion field to the output data.

The input array must have 3 components to compute this. The default is off.

inline void **SetQCriterionName**(const std::string &name)

When [SetComputeQCriterion\(\)](#) is enabled, the result is stored in a field of this name.

If not specified, the name of the field will be QCriterion.

inline const std::string &**GetQCriterionName()** const

When [SetComputeQCriterion\(\)](#) is enabled, the result is stored in a field of this name.

If not specified, the name of the field will be QCriterion.

inline void **SetComputeGradient**(bool enable)

Add gradient field to the output data.

The name of the array will be Gradients unless otherwise specified with [SetOutputFieldName](#) and will be a cell field unless [ComputePointGradient\(\)](#) is enabled. It is useful to turn this off when you are only interested in the results of Divergence, Vorticity, or QCriterion. The default is on.

inline bool **GetComputeGradient()** const

Add gradient field to the output data.

The name of the array will be Gradients unless otherwise specified with [SetOutputFieldName](#) and will be a cell field unless [ComputePointGradient\(\)](#) is enabled. It is useful to turn this off when you are only interested in the results of Divergence, Vorticity, or QCriterion. The default is on.

inline void **SetColumnMajorOrdering()**

Make the vector gradient output format be in FORTRAN Column-major order.

This is only used when the input field is a vector field. Enabling column-major is important if integrating with other projects such as VTK. Default: Row Order.

inline void **SetRowMajorOrdering()**

Make the vector gradient output format be in C Row-major order.

This is only used when the input field is a vector field. Default: Row Order.

### 10.13.4 Surface Normals

The [vtkm::filter::vector\\_analysis::SurfaceNormals](#) filter computes the surface normals of a polygonal data set at its points and/or cells. The filter takes a data set as input and by default, uses the active coordinate system to compute the normals.

class **SurfaceNormals** : public vtkm::filter::Filter

Computes normals for polygonal mesh.

This filter computes surface normals on points and/or cells of a polygonal dataset. The cell normals are faceted and are computed based on the plane where a face lies. The point normals are smooth normals, computed by averaging the face normals of incident cells. The normals will be consistently oriented to point in the direction of the same connected surface if possible.

The point and cell normals may be oriented to a point outside of the manifold surface by turning on the auto orient normals option ([SetAutoOrientNormals\(\)](#)), or they may point inward by also setting flip normals ([SetFlipNormals\(\)](#)) to true.

Triangle vertices will be reordered to be wound counter-clockwise around the cell normals when the consistency option (*SetConsistency()*) is enabled.

For non-polygonal cells, a zeroed vector is assigned. The point normals are computed by averaging the cell normals of the incident cells of each point.

The default name for the output fields is `Normals`, but that can be overridden using the *SetCellNormalsName()* and *SetPointNormalsName()* methods. The filter will also respect the name in *SetOutputFieldName()* if neither of the others are set.

## Public Functions

### **SurfaceNormals()**

Create *SurfaceNormals* filter.

This calls this->SetUseCoordinateSystemAsField(true) since that is the most common use-case for surface normals.

inline void **SetGenerateCellNormals**(bool value)

Specify whether cell normals should be generated.

Default is off.

inline bool **GetGenerateCellNormals**() const

Specify whether cell normals should be generated.

Default is off.

inline void **SetNormalizeCellNormals**(bool value)

Specify whether the cell normals should be normalized.

Default value is true. The intended use case of this flag is for faster, approximate point normals generation by skipping the normalization of the face normals. Note that when set to false, the result cell normals will not be unit length normals and the point normals will be different.

inline bool **GetNormalizeCellNormals**() const

Specify whether the cell normals should be normalized.

Default value is true. The intended use case of this flag is for faster, approximate point normals generation by skipping the normalization of the face normals. Note that when set to false, the result cell normals will not be unit length normals and the point normals will be different.

inline void **SetGeneratePointNormals**(bool value)

Specify whether the point normals should be generated.

Default is on.

inline bool **GetGeneratePointNormals**() const

Specify whether the point normals should be generated.

Default is on.

inline void **SetCellNormalsName**(const std::string &name)

Specify the name of the cell normals field.

Default is `Normals`.

inline const std::string &**GetCellNormalsName**() const

Specify the name of the cell normals field.

Default is `Normals`.

inline void **SetPointNormalsName**(const std::string &name)

Specify the name of the point normals field.

Default is Normals.

inline const std::string &**GetPointNormalsName**() const

Specify the name of the point normals field.

Default is Normals.

inline void **SetAutoOrientNormals**(bool v)

Specify whether to orient the normals outwards from the surface.

This requires a closed manifold surface or the behavior is undefined. This option is expensive but might be necessary for rendering. To make the normals point inward, set FlipNormals to true. Default is off.

inline bool **GetAutoOrientNormals**() const

Specify whether to orient the normals outwards from the surface.

This requires a closed manifold surface or the behavior is undefined. This option is expensive but might be necessary for rendering. To make the normals point inward, set FlipNormals to true. Default is off.

inline void **SetFlipNormals**(bool v)

Specify the direction to point normals when [SetAutoOrientNormals\(\)](#) is true.

When this flag is false (the default), the normals will be oriented to point outward. When the flag is true, the normals will point inward. This option has no effect if auto orient normals is off.

inline bool **GetFlipNormals**() const

Specify the direction to point normals when [SetAutoOrientNormals\(\)](#) is true.

When this flag is false (the default), the normals will be oriented to point outward. When the flag is true, the normals will point inward. This option has no effect if auto orient normals is off.

inline void **SetConsistency**(bool v)

Specify whether polygon winding should be made consistent with normal orientation.

Triangles are wound such that their points are counter-clockwise around the generated cell normal. Default is true. This currently only affects triangles. This is only applied when cell normals are generated.

inline bool **GetConsistency**() const

Specify whether polygon winding should be made consistent with normal orientation.

Triangles are wound such that their points are counter-clockwise around the generated cell normal. Default is true. This currently only affects triangles. This is only applied when cell normals are generated.

### 10.13.5 Vector Magnitude

The `vtkm::filter::vector_analysis::VectorMagnitude` filter takes a field comprising vectors and computes the magnitude for each vector. The vector field is selected as usual with the `vtkm::filter::vector_analysis::VectorMagnitude::SetActiveField()` method. The default name for the output field is `magnitude`, but that can be overridden as always using the `vtkm::filter::vector_analysis::VectorMagnitude::SetOutputFieldName()` method.

class **VectorMagnitude** : public vtkm::filter::Filter

Compute the magnitudes of a vector field.

The vector field is selected with the [SetActiveField\(\)](#) method. The default name for the output field is `magnitude`, but that can be overridden using the [SetOutputFieldName\(\)](#) method.

## 10.14 ZFP Compression

`vtkm::filter::zfp::ZFPCompressor1D`, `vtkm::filter::zfp::ZFPCompressor2D`, and `vtkm::filter::zfp::ZFPCompressor3D` are a set of filters that take a 1D, 2D, and 3D field, respectively, and compresses the values using the compression algorithm ZFP.

The field is selected as usual with the `vtkm::filter::zfp::ZFPCompressor3D::SetActiveField()` method. The rate of compression is set using `vtkm::filter::zfp::ZFPCompressor3D::SetRate()`. The default name for the output field is `compressed`.

class **ZFPCompressor1D** : public `vtkm::filter::Filter`

Compress a scalar field using ZFP.

Takes as input a 1D array and generates an output of compressed data.

**Warning:** This filter currently only supports 1D structured cell sets.

### Public Functions

inline void **SetRate**(`vtkm::Float64` \_rate)

Specifies the rate of compression.

inline `vtkm::Float64` **GetRate**()

Specifies the rate of compression.

class **ZFPCompressor2D** : public `vtkm::filter::Filter`

Compress a scalar field using ZFP.

Takes as input a 2D array and generates an output of compressed data.

**Warning:** This filter is currently only supports 2D structured cell sets.

### Public Functions

inline void **SetRate**(`vtkm::Float64` \_rate)

Specifies the rate of compression.

inline `vtkm::Float64` **GetRate**()

Specifies the rate of compression.

class **ZFPCompressor3D** : public `vtkm::filter::Filter`

Compress a scalar field using ZFP.

Takes as input a 3D array and generates an output of compressed data.

**Warning:** This filter is currently only supports 3D structured cell sets.

## Public Functions

inline void **SetRate**(vtkm::Float64 \_rate)

Specifies the rate of compression.

inline vtkm::Float64 **GetRate**()

Specifies the rate of compression.

**vtkm::filter::zfp::ZFPDecompressor1D**, **vtkm::filter::zfp::ZFPDecompressor2D**, and **vtkm::filter::zfp::ZFPDecompressor3D** are a set of filters that take a compressed 1D, 2D, and 3D field, respectively, and decompress the values using the compression algorithm ZFP.

The field is selected as usual with the **vtkm::filter::zfp::ZFPDecompressor3D::SetActiveField()** method. The rate of compression is set using **vtkm::filter::zfp::ZFPDecompressor3D::SetRate()**. The default name for the output field is **decompressed**.

class **ZFPDecompressor1D** : public vtkm::filter::Filter

Decompress a scalar field using ZFP.

Takes as input a 1D compressed array and generates the decompressed version of the data.

**Warning:** This filter is currently only supports 1D structured cell sets.

## Public Functions

inline void **SetRate**(vtkm::Float64 \_rate)

Specifies the rate of compression.

inline vtkm::Float64 **GetRate**()

Specifies the rate of compression.

class **ZFPDecompressor2D** : public vtkm::filter::Filter

Decompress a scalar field using ZFP.

Takes as input a 2D compressed array and generates the decompressed version of the data.

**Warning:** This filter is currently only supports 2D structured cell sets.

## Public Functions

inline void **SetRate**(vtkm::Float64 \_rate)

Specifies the rate of compression.

inline vtkm::Float64 **GetRate**()

Specifies the rate of compression.

class **ZFPDecompressor3D** : public vtkm::filter::Filter

Decompress a scalar field using ZFP.

Takes as input a 3D compressed array and generates the decompressed version of the data.

**Warning:** This filter is currently only supports 3D structured cell sets.

### Public Functions

inline void **SetRate**(vtkm::Float64 \_rate)

Specifies the rate of compression.

inline vtkm::Float64 **GetRate**()

Specifies the rate of compression.

## RENDERING

Rendering, the generation of images from data, is a key component to visualization. To assist with rendering, VTK-m provides a rendering package to produce imagery from data, which is located in the `vtkm::rendering` namespace.

The rendering package in VTK-m is not intended to be a fully featured rendering system or library. Rather, it is a lightweight rendering package with two primary use cases:

- New users getting started with VTK-m need a “quick and dirty” render method to see their visualization results.
- In situ visualization that integrates VTK-m with a simulation or other data-generation system might need a lightweight rendering method.

Both of these use cases require just a basic rendering platform. Because VTK-m is designed to be integrated into larger systems, it does not aspire to have a fully featured rendering system.

---

### Did You Know?

VTK-m’s big sister toolkit VTK is already integrated with VTK-m and has its own fully featured rendering system. If you need more rendering capabilities than what VTK-m provides, you can leverage VTK instead.

---

## 11.1 Scenes and Actors

The primary intent of the rendering package in VTK-m is to visually display the data that is loaded and processed. Data are represented in VTK-m by `vtkm::cont::DataSet` objects, which are described in [Chapter 7 \(Data Sets\)](#). They are also the object created from [Chapter 8 \(File I/O\)](#) and [Chapter 9 \(Running Filters\)](#).

To render a `vtkm::cont::DataSet`, the data are wrapped in a `vtkm::rendering::Actor` class. The `vtkm::rendering::Actor` holds the components of the `vtkm::cont::DataSet` to render (a cell set, a coordinate system, and a field). A color table can also be optionally be specified, but a default color table will be specified otherwise.

Example 1: Creating an `vtkm::rendering::Actor` and adding it to a `vtkm::rendering::Scene`.

```
1  vtkm::rendering::Actor actor(surfaceData.GetCellSet(),
2                               surfaceData.GetCoordinateSystem(),
3                               surfaceData.GetField("RandomPointScalars"));
4
5  vtkm::rendering::Scene scene;
6  scene.AddActor(actor);
```

class **Actor**

An item to be rendered.

The **Actor** holds the geometry from a `vtkm::cont::DataSet` as well as other visual properties that define how the geometry should look when it is rendered.

### Public Functions

**Actor**(const vtkm::cont::UnknownCellSet &cells, const vtkm::cont::CoordinateSystem &coordinates, const vtkm::cont::Field &scalarField)

Create an **Actor** object that renders a set of cells positioned by a given coordiante system.

A field to apply psudocoloring is also provided. The default colormap is applied. The cells, coordinates, and field are typically pulled from a `vtkm::cont::DataSet` object.

**Actor**(const vtkm::cont::UnknownCellSet &cells, const vtkm::cont::CoordinateSystem &coordinates, const vtkm::cont::Field &scalarField, const vtkm::cont::ColorTable &colorTable)

Create an **Actor** object that renders a set of cells positioned by a given coordiante system.

A field to apply psudocoloring is also provided. A color table providing the map from scalar values to colors is also provided. The cells, coordinates, and field are typically pulled from a `vtkm::cont::DataSet` object.

**Actor**(const vtkm::cont::UnknownCellSet &cells, const vtkm::cont::CoordinateSystem &coordinates, const vtkm::cont::Field &scalarField, const vtkm::rendering::Color &color)

Create an **Actor** object that renders a set of cells positioned by a given coordiante system.

A constant color to apply to the object is also provided. The cells and coordinates are typically pulled from a `vtkm::cont::DataSet` object.

void **SetScalarRange**(const vtkm::Range &scalarRange)

Specifies the range for psudocoloring.

When coloring data by mapping a scalar field to colors, this is the range used for the colors provided by the table. If a range is not provided, the range of data in the field is used.

**vtkm::rendering::Actor** objects are collected together in an object called **vtkm::rendering::Scene**.

An `vtkm::rendering::Actor` is added to a `vtkm::rendering::Scene` with the `vtkm::rendering::Scene::AddActor()` method.

class **Scene**

A simple collection of things to render.

The **Scene** is a simple collection of **Actor** objects.

### Public Functions

void **AddActor**(vtkm::rendering::Actor actor)

Add an **Actor** to the scene.

const vtkm::rendering::Actor &**GetActor**(vtkm::IdComponent index) const

Get one of the **Actors** from the scene.

vtkm::IdComponent **GetNumberOfActors**() const

Get the number of **Actors** in the scene.



`vtkm::Bounds GetSpatialBounds()` const

The computed spatial bounds of combined data from all contained *Actors*.

The following example demonstrates creating a `vtkm::rendering::Scene` with one `vtkm::rendering::Actor`.

## 11.2 Canvas

A canvas is a unit that represents the image space that is the target of the rendering. The canvas' primary function is to manage the buffers that hold the working image data during the rendering. The canvas also manages the context and state of the rendering subsystem.

`vtkm::rendering::Canvas` is the base class of all canvas objects. Each type of rendering system has its own canvas subclass, but currently the only rendering system provided by VTK-m is the internal ray tracer. The canvas for the ray tracer is `vtkm::rendering::CanvasRayTracer`. `vtkm::rendering::CanvasRayTracer` is typically constructed by giving the width and height of the image to render.

Example 2: Creating a canvas for rendering.

```
1 vtkm::rendering::CanvasRayTracer canvas(1920, 1080);
```

class **CanvasRayTracer** : public `vtkm::rendering::Canvas`

Represents the image space that is the target of rendering using the internal ray tracing code.

### Public Functions

**CanvasRayTracer**(`vtkm::Id` width = 1024, `vtkm::Id` height = 1024)

Construct a canvas of a given width and height.

virtual `vtkm::rendering::Canvas *NewCopy()` const override

Create a new *Canvas* object of the same subtype as this one.

class **Canvas**

Represents the image space that is the target of rendering.

Subclassed by `vtkm::rendering::CanvasRayTracer`

### Public Functions

**Canvas**(`vtkm::Id` width = 1024, `vtkm::Id` height = 1024)

Construct a canvas of a given width and height.

virtual `vtkm::rendering::Canvas *NewCopy()` const

Create a new *Canvas* object of the same subtype as this one.

virtual void **Clear()**

Clear out the image buffers.

virtual void **BlendBackground()**

Blend the foreground data with the background color.

When a render is started, it is given a zeroed background rather than the background color specified by `SetBackgroundColor()`. This is because when blending pixel fragments of transparent objects the background color can interfere. Call this method after the render is completed for the final blend to get the proper background color.

vtkm::Id **GetWidth()** const

The width of the image.

vtkm::Id **GetHeight()** const

The height of the image.

const ColorBufferType &**GetColorBuffer()** const

Get the color channels of the image.

ColorBufferType &**GetColorBuffer()**

Get the color channels of the image.

const DepthBufferType &**GetDepthBuffer()** const

Get the depth channel of the image.

DepthBufferType &**GetDepthBuffer()**

Get the depth channel of the image.

vtkm::cont::DataSet **GetDataSet**(const std::string &colorFieldName = "color", const std::string &depthFieldName = "depth") const

Gets the image in this *Canvas* as a `vtkm::cont::DataSet`.

The returned `DataSet` will be a uniform structured 2D grid. The color and depth buffers will be attached as field with the given names. If the name for the color or depth field is empty, then that respective field will not be added.

The arrays of the color and depth buffer are shallow copied. Thus, changes in the *Canvas* may cause unexpected behavior in the `DataSet`.

vtkm::cont::DataSet **GetDataSet**(const char \*colorFieldName, const char \*depthFieldName = "depth") const

Gets the image in this *Canvas* as a `vtkm::cont::DataSet`.

The returned `DataSet` will be a uniform structured 2D grid. The color and depth buffers will be attached as field with the given names. If the name for the color or depth field is empty, then that respective field will not be added.

The arrays of the color and depth buffer are shallow copied. Thus, changes in the *Canvas* may cause unexpected behavior in the `DataSet`.

void **ResizeBuffers**(vtkm::Id width, vtkm::Id height)

Change the size of the image.

const vtkm::rendering::Color &**GetBackgroundColor()** const

Specify the background color.

void **SetBackgroundColor**(const vtkm::rendering::Color &color)

Specify the background color.

const vtkm::rendering::Color &**GetForegroundColor()** const

Specify the foreground color used for annotations.

void **SetForegroundColor**(const vtkm::rendering::Color &color)

Specify the foreground color used for annotations.

virtual void **SaveAs**(const std::string &fileName) const

Save the rendered image.

If the filename ends with “.png”, it will be saved in the portable network graphic format. Otherwise, the file will be saved in Netbpm portable pixmap format.

virtual vtkm::rendering::WorldAnnotator \***CreateWorldAnnotator**() const

Creates a WorldAnnotator of a type that is paired with this *Canvas*.

Other types of world annotators might work, but this provides a default.

The WorldAnnotator is created with the C++ new keyword (so it should be deleted with delete later). A pointer to the created WorldAnnotator is returned.

## 11.3 Mappers

A mapper is a unit that converts data (managed by an *vtkm::rendering::Actor*) and issues commands to the rendering subsystem to generate images. All mappers in VTK-m are a subclass of *vtkm::rendering::Mapper*. Different mappers could render different types of data in different ways. For example, one mapper might render polygonal surfaces whereas another might render polyhedra as a translucent volume.

class **Mapper**

Converts data into commands to a rendering system.

This is the base class for all mapper classes in VTK-m. Different concrete derived classes can provide different representations and rendering techniques.

Subclassed by *vtkm::rendering::MapperConnectivity*, *vtkm::rendering::MapperCylinder*, *vtkm::rendering::MapperGlyphBase*, *vtkm::rendering::MapperPoint*, *vtkm::rendering::MapperQuad*, *vtkm::rendering::MapperRayTracer*, *vtkm::rendering::MapperVolume*, *vtkm::rendering::MapperWireframer*

The following mappers are provided by VTK-m.

class **MapperCylinder** : public *vtkm::rendering::Mapper*

*MapperCylinder* renders edges from a cell set and renders them as cylinders via ray tracing.

### Public Functions

void **UseVariableRadius**(bool useVariableRadius)

render points using a variable radius based on the scalar field.

The default is false.

void **SetRadius**(const *vtkm::Float32* &radius)

Set a base radius for all points.

If a radius is never specified the default heuristic is used.

void **SetRadiusDelta**(const *vtkm::Float32* &delta)

When using a variable radius for all cylinder, the radius delta controls how much larger and smaller radii become based on the scalar field.

If the delta is 0 all points will have the same radius. If the delta is 0.5 then the max/min scalar values would have a radii of base +/- base \* 0.5.

class **MapperGlyphBase** : public vtkm::rendering::Mapper

Base class for glyph mappers.

Glyph mappers place 3D icons at various places in the mesh. The icons are placed based on the location of points or cells in the mesh.

Subclassed by *vtkm::rendering::MapperGlyphScalar*, *vtkm::rendering::MapperGlyphVector*

## Public Functions

virtual vtkm::cont::Field::Association **GetAssociation()** const

Specify the elements the glyphs will be associated with.

The glyph mapper will place glyphs over locations specified by either the points or the cells of a mesh. The glyph may also be oriented by a scalar field with the same association.

virtual void **SetAssociation**(vtkm::cont::Field::Association association)

Specify the elements the glyphs will be associated with.

The glyph mapper will place glyphs over locations specified by either the points or the cells of a mesh. The glyph may also be oriented by a scalar field with the same association.

virtual bool **GetUseCells()** const

Specify the elements the glyphs will be associated with.

The glyph mapper will place glyphs over locations specified by either the points or the cells of a mesh. The glyph may also be oriented by a scalar field with the same association.

virtual void **SetUseCells()**

Specify the elements the glyphs will be associated with.

The glyph mapper will place glyphs over locations specified by either the points or the cells of a mesh. The glyph may also be oriented by a scalar field with the same association.

virtual bool **GetUsePoints()** const

Specify the elements the glyphs will be associated with.

The glyph mapper will place glyphs over locations specified by either the points or the cells of a mesh. The glyph may also be oriented by a scalar field with the same association.

virtual void **SetUsePoints()**

Specify the elements the glyphs will be associated with.

The glyph mapper will place glyphs over locations specified by either the points or the cells of a mesh. The glyph may also be oriented by a scalar field with the same association.

virtual vtkm::Float32 **GetBaseSize()** const

Specify the size of each glyph (before scaling).

If the base size is not set to a positive value, it is automatically sized with a heuristic based off the bounds of the geometry.

virtual void **SetBaseSize**(vtkm::Float32 size)

Specify the size of each glyph (before scaling).

If the base size is not set to a positive value, it is automatically sized with a heuristic based off the bounds of the geometry.

virtual bool **GetScaleByValue**() const

Specify whether to scale the glyphs by a field.

virtual void **SetScaleByValue**(bool on)

Specify whether to scale the glyphs by a field.

virtual vtkm::Float32 **GetScaleDelta**() const

Specify the range of values to scale the glyphs.

When `ScaleByValue` is on, the glyphs will be scaled proportionally to the field magnitude. The `ScaleDelta` determines how big and small they get. For a `ScaleDelta` of one, the smallest field values will have glyphs of zero size and the maximum field values will be twice the base size. A `ScaleDelta` of 0.5 will result in glyphs sized in the range of 0.5 times the base size to 1.5 times the base size. `ScaleDelta` outside the range [0, 1] is undefined.

virtual void **SetScaleDelta**(vtkm::Float32 delta)

Specify the range of values to scale the glyphs.

When `ScaleByValue` is on, the glyphs will be scaled proportionally to the field magnitude. The `ScaleDelta` determines how big and small they get. For a `ScaleDelta` of one, the smallest field values will have glyphs of zero size and the maximum field values will be twice the base size. A `ScaleDelta` of 0.5 will result in glyphs sized in the range of 0.5 times the base size to 1.5 times the base size. `ScaleDelta` outside the range [0, 1] is undefined.

class **MapperGlyphScalar** : public vtkm::rendering::MapperGlyphBase

A mapper that produces unoriented glyphs.

This mapper is meant to be used with scalar fields. The glyphs can be optionally sized based on the field.

### Public Functions

vtkm::rendering::GlyphType **GetGlyphType**() const

Specify the shape of the glyphs.

void **SetGlyphType**(vtkm::rendering::GlyphType glyphType)

Specify the shape of the glyphs.

class **MapperGlyphVector** : public vtkm::rendering::MapperGlyphBase

A mapper that produces oriented glyphs.

This mapper is meant to be used with 3D vector fields. The glyphs are oriented in the direction of the vector field. The glyphs can be optionally sized based on the magnitude of the field.

### Public Functions

vtkm::rendering::GlyphType **GetGlyphType**() const

Specify the shape of the glyphs.

void **SetGlyphType**(vtkm::rendering::GlyphType glyphType)

Specify the shape of the glyphs.

class **MapperPoint** : public vtkm::rendering::Mapper

This mapper renders points from a cell set.

This mapper can natively create points from vertex cell shapes as well as use the points defined by a coordinate system.

## Public Functions

virtual vtkm::cont::Field::Association **GetAssociation**() const

Specify the elements the points will be associated with.

The point mapper will place visible points over locations specified by either the points or the cells of a mesh.

virtual void **SetAssociation**(vtkm::cont::Field::Association association)

Specify the elements the points will be associated with.

The point mapper will place visible points over locations specified by either the points or the cells of a mesh.

virtual bool **GetUseCells**() const

Specify the elements the points will be associated with.

The point mapper will place visible points over locations specified by either the points or the cells of a mesh.

virtual void **SetUseCells**()

Specify the elements the points will be associated with.

The point mapper will place visible points over locations specified by either the points or the cells of a mesh.

virtual bool **GetUsePoints**() const

Specify the elements the points will be associated with.

The point mapper will place visible points over locations specified by either the points or the cells of a mesh.

virtual void **SetUsePoints**()

Specify the elements the points will be associated with.

The point mapper will place visible points over locations specified by either the points or the cells of a mesh.

void **UseVariableRadius**(bool useVariableRadius)

Render points using a variable radius based on the scalar field.

The default is false.

void **SetRadius**(const vtkm::Float32 &radius)

Set a base radius for all points.

If a radius is never specified the default heuristic is used.

void **SetRadiusDelta**(const vtkm::Float32 &delta)

When using a variable radius for all points, the radius delta controls how much larger and smaller radii become based on the scalar field.

If the delta is 0 all points will have the same radius. If the delta is 0.5 then the max/min scalar values would have a radii of base +/- base \* 0.5.

class **MapperQuad** : public vtkm::rendering::Mapper

A mapper that renders quad faces from a cell set via ray tracing.

As opposed to breaking quads into two triangles, scalars are interpolated **using** all 4 points of the quad resulting in more accurate interpolation.

class **MapperRayTracer** : public vtkm::rendering::Mapper

Mapper to render surfaces using ray tracing.

Provides a “standard” data mapper that uses ray tracing to render the surfaces of DataSet objects.

class **MapperVolume** : public vtkm::rendering::Mapper

Mapper that renders a volume as a translucent cloud.

### Public Functions

void **SetSampleDistance**(const vtkm::Float32 distance)

Specify how much space is between samples of rays that traverse the volume.

The volume rendering ray caster finds the entry point of the ray through the volume and then samples the volume along the direction of the ray at regular intervals. This parameter specifies how far these samples occur.

class **MapperWireframer** : public vtkm::rendering::Mapper

Mapper that renders the edges of a mesh.

Each edge in the mesh is rendered as a line, which provides a wireframe representation of the data.

### Public Functions

bool **GetShowInternalZones**() const

Specify whether to show interior edges.

When rendering a 3D volume of data, the *MapperWireframer* can show either the wireframe of the external surface of the data (the default) or render the entire wireframe including the internal edges.

void **SetShowInternalZones**(bool showInternalZones)

Specify whether to show interior edges.

When rendering a 3D volume of data, the *MapperWireframer* can show either the wireframe of the external surface of the data (the default) or render the entire wireframe including the internal edges.

## 11.4 Views

A view is a unit that collects all the structures needed to perform rendering. It contains everything needed to take a `vtkm::rendering::Scene` and use a `vtkm::rendering::Mapper` to render it onto a `vtkm::rendering::Canvas`. The view also annotates the image with spatial and scalar properties.

The base class for all views is `vtkm::rendering::View`, which is an abstract class. You must choose one of the three provided subclasses, `vtkm::rendering::View3D`, `vtkm::rendering::View2D`, and `vtkm::rendering::View1D`, depending on the type of data being presented. All three view classes take a `vtkm::rendering::Scene`, a `vtkm::rendering::Mapper`, and a `vtkm::rendering::Canvas` as arguments to their constructor.

Example 3: Constructing a `vtkm::rendering::View`.

```

1  vtkm::rendering::Actor actor(surfaceData.GetCellSet(),
2                                surfaceData.GetCoordinateSystem(),
3                                surfaceData.GetField("RandomPointScalars"));
4
5  vtkm::rendering::Scene scene;
6  scene.AddActor(actor);
7
8  vtkm::rendering::MapperRayTracer mapper;
9  vtkm::rendering::CanvasRayTracer canvas(1920, 1080);
10
11  vtkm::rendering::View3D view(scene, mapper, canvas);

```

class **View**

The abstract class representing the view of a rendering scene.

Subclassed by `vtkm::rendering::View1D`, `vtkm::rendering::View2D`, `vtkm::rendering::View3D`

### Public Functions

const `vtkm::rendering::Scene` &**GetScene**() const

Specify the scene object holding the objects to render.

`vtkm::rendering::Scene` &**GetScene**()

Specify the scene object holding the objects to render.

void **SetScene**(const `vtkm::rendering::Scene` &scene)

Specify the scene object holding the objects to render.

const `vtkm::rendering::Mapper` &**GetMapper**() const

Specify the mapper object determining how objects are rendered.

`vtkm::rendering::Mapper` &**GetMapper**()

Specify the mapper object determining how objects are rendered.

const `vtkm::rendering::Canvas` &**GetCanvas**() const

Specify the canvas object that holds the buffer to render into.

`vtkm::rendering::Canvas` &**GetCanvas**()

Specify the canvas object that holds the buffer to render into.



```
const vtkm::rendering::Camera &GetCamera() const
```

Specify the perspective from which to render a scene.

```
vtkm::rendering::Camera &GetCamera()
```

Specify the perspective from which to render a scene.

```
void SetCamera(const vtkm::rendering::Camera &camera)
```

Specify the perspective from which to render a scene.

```
const vtkm::rendering::Color &GetBackgroundColor() const
```

Specify the color used where nothing is rendered.

```
void SetBackgroundColor(const vtkm::rendering::Color &color)
```

Specify the color used where nothing is rendered.

```
void SetForegroundColor(const vtkm::rendering::Color &color)
```

Specify the color of foreground elements.

The foreground is typically used for annotation elements. The foreground should contrast well with the background.

```
virtual void Paint() = 0
```

Render a scene and store the result in the canvas' buffers.

```
void SaveAs(const std::string &fileName) const
```

Save the rendered image.

If the filename ends with “.png”, it will be saved in the portable network graphic format. Otherwise, the file will be saved in Netbpm portable pixmap format.

```
class View1D : public vtkm::rendering::View
```

A view for a 1D data set.

1D data are rendered as an X-Y plot with the values shown on the Y axis.

## Public Functions

```
virtual void Paint() override
```

Render a scene and store the result in the canvas' buffers.

```
inline void SetLogX(bool l)
```

Specify whether log scaling should be used on the X axis.

```
inline void SetLogY(bool l)
```

Specify whether log scaling should be used on the Y axis.

```
class View2D : public vtkm::rendering::View
```

A view for a 3D data set.

2D data are rendered directly on the X-Y plane.

## Public Functions

virtual void **Paint**() override

Render a scene and store the result in the canvas' buffers.

class **View3D** : public vtkm::rendering::View

A view for a 3D data set.

## Public Functions

virtual void **Paint**() override

Render a scene and store the result in the canvas' buffers.

The `vtkm::rendering::View` also maintains a background color (the color used in areas where nothing is drawn) and a foreground color (the color used for annotation elements). By default, the `vtkm::rendering::View` has a black background and a white foreground. These can be set in the view's constructor, but it is a bit more readable to set them using the `vtkm::rendering::View::SetBackgroundColor()` and `vtkm::rendering::View::SetForegroundColor()` methods. In either case, the colors are specified using the `vtkm::rendering::Color` helper class, which manages the red, green, and blue color channels as well as an optional alpha channel. These channel values are given as floating point values between 0 and 1.

Example 4: Changing the background and foreground colors of a  
`vtkm::rendering::View`.

```
1 view.SetBackgroundColor(vtkm::rendering::Color(1.0f, 1.0f, 1.0f));  
2 view.SetForegroundColor(vtkm::rendering::Color(0.0f, 0.0f, 0.0f));
```

---

## Common Errors

Although the background and foreground colors are set independently, it will be difficult or impossible to see the annotation if there is not enough contrast between the background and foreground colors. Thus, when changing a `vtkm::rendering::View`'s background color, it is always good practice to also change the foreground color.

---

class **Color**

Representation of a color.

The color is defined as red, green, and blue intensities as well as an alpha representation of transparency (RGBA). The class provides mechanisms to retrieve the color as 8-bit integers or floating point values in the range [0, 1].

## Public Functions

inline **Color**()

Create a black color.

inline **Color**(vtkm::Float32 r\_, vtkm::Float32 g\_, vtkm::Float32 b\_, vtkm::Float32 a\_ = 1.f)

Create a color with specified RGBA values.

The values are floating point and in the range [0, 1].

```
inline Color(const vtkm::Vec4f_32 &components)
```

Create a color with specified RGBA values.

The values are floating point and in the range [0, 1].

```
inline void SetComponentFromByte(vtkm::Int32 i, vtkm::UInt8 v)
```

Set the color value from 8 bit RGBA components.

The components are packed together into a 32-bit (4-byte) values.

Once the `vtkm::rendering::View` is constructed, initialized, and set up, it is ready to render. This is done by calling the `vtkm::rendering::View::Paint()` method.

Example 5: Using `vtkm::rendering::Canvas::Paint()` in a display callback.

```
view.Paint();
```

Putting together Example 3, Example 4, and Example 5, the final render of a view looks like that in Figure 1.

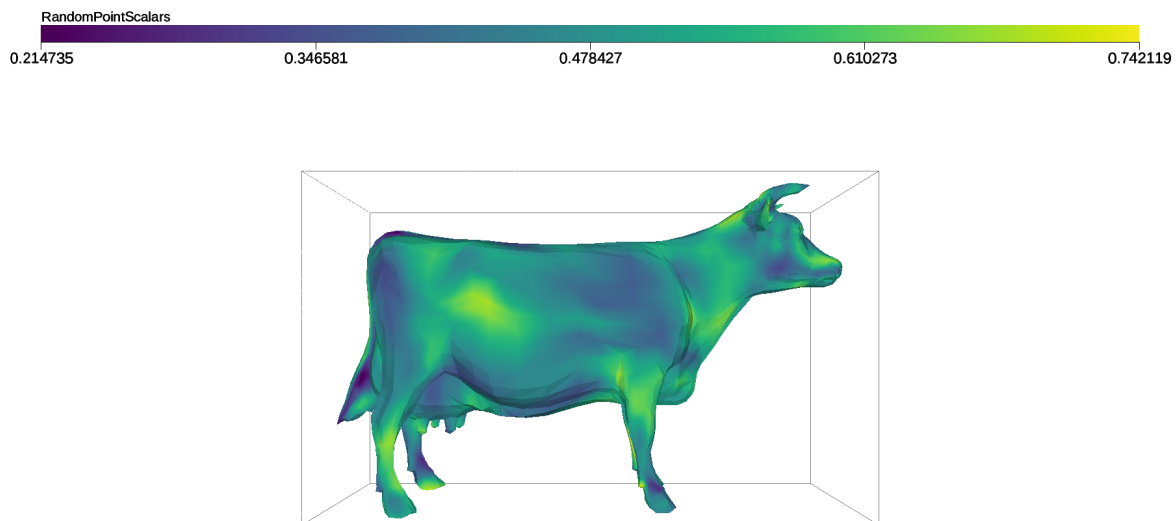


Figure 1: Example output of VTK-m's rendering system.

Of course, the `vtkm::rendering::CanvasRayTracer` created in Example 3 is an offscreen rendering buffer, so you cannot immediately see the image. When doing batch visualization, an easy way to output the image to a file for later viewing is with the `vtkm::rendering::View::SaveAs()` method. This method can save the image in either PNG or in the portable pixmap (PPM) format.

Example 6: Saving the result of a render as an image file.

```
view.SaveAs("BasicRendering.png");
```

We visit doing interactive rendering in a GUI later in Section 11.7 (Interactive Rendering).

## 11.5 Changing Rendering Modes

Example 3 constructs the default mapper for ray tracing, which renders the data as an opaque solid. However, you can change the rendering mode by using one of the other mappers listed in [Section 11.3 \(Mappers\)](#). For example, say you just wanted to see a wireframe representation of your data. You can achieve this by using `vtkm::rendering::MapperWireframer`.

Example 7: Creating a mapper for a wireframe representation.

```
1  vtkm::rendering::MapperWireframer mapper;  
2  vtkm::rendering::View3D view(scene, mapper, canvas);
```

Alternatively, perhaps you wish to render just the points of mesh. `vtkm::rendering::MapperGlyphScalar` renders the points as glyphs and also optionally can scale the glyphs based on field values.

Example 8: Creating a mapper for point representation.

```
1  vtkm::rendering::MapperGlyphScalar mapper;  
2  mapper.SetGlyphType(vtkm::rendering::GlyphType::Cube);  
3  mapper.SetScaleByValue(true);  
4  mapper.SetScaleDelta(10.0f);  
5  
6  vtkm::rendering::View3D view(scene, mapper, canvas);
```

These mappers respectively render the images shown in [Figure 2](#). Other mappers, such as those that can render translucent volumes, are also available.

## 11.6 Manipulating the Camera

The `vtkm::rendering::View` uses an object called `vtkm::rendering::Camera` to describe the vantage point from which to draw the geometry. The camera can be retrieved from the `vtkm::rendering::View::GetCamera()` method. That retrieved camera can be directly manipulated or a new camera can be provided by calling `vtkm::rendering::View::SetCamera()`. In this section we discuss camera setups typical during view set up. Camera movement during interactive rendering is revisited in [Section 11.7.2 \(Camera Movement\)](#).

### class **Camera**

Specifies the viewport for a rendering.

The `vtkm::rendering::View` object holds a `Camera` object to specify from what perspective the rendering should take place.

A `Camera` operates in one of two major modes: 2D mode or 3D mode. 2D mode is designed for looking at flat geometry (or close to flat geometry) that is parallel to the x-y plane. 3D mode provides the freedom to place the camera anywhere in 3D space.

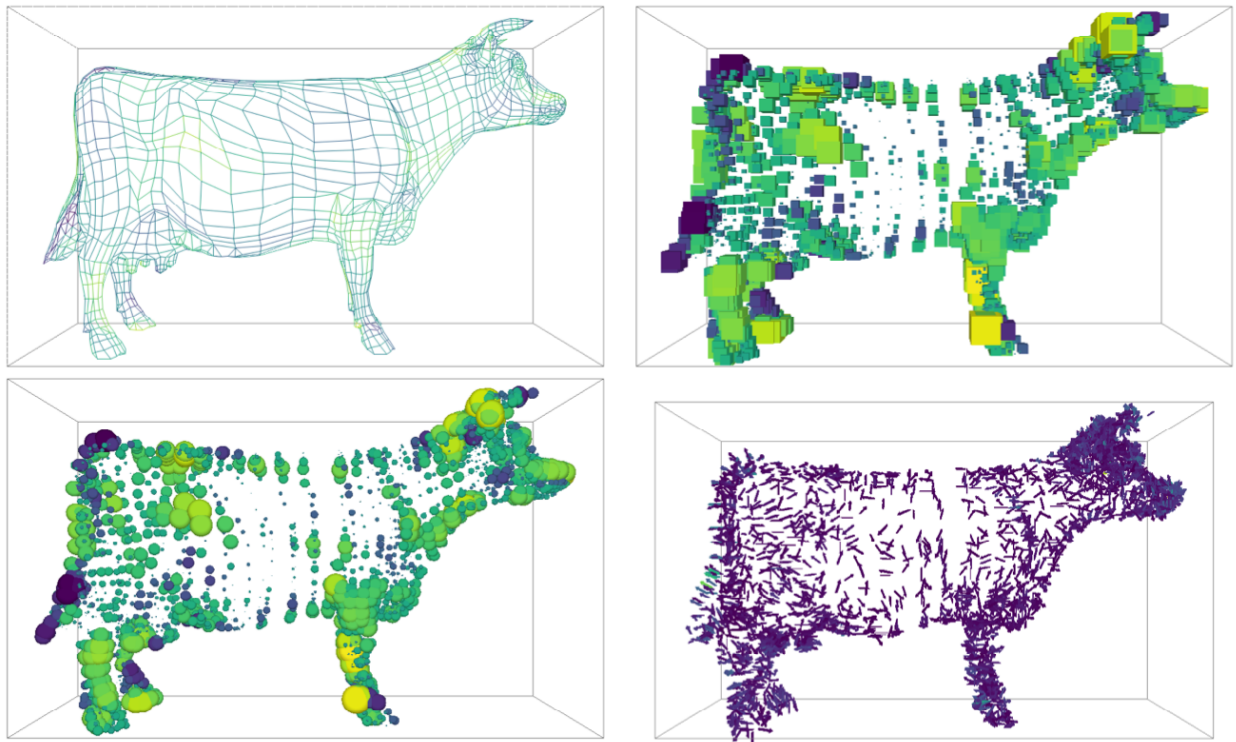


Figure 2: Examples of alternate rendering modes using different mappers. The top left image is rendered with `vtkm::rendering::MapperWireframer`. The top right and bottom left images are rendered with `vtkm::rendering::MapperGlyphScalar`. The bottom right image is rendered with `vtkm::rendering::MapperGlyphVector`.

## Public Functions

inline vtkm::rendering::Camera::Mode **GetMode**() const

The mode of the camera (2D or 3D).

`vtkm::rendering::Camera` can be set to a 2D or 3D mode. 2D mode is used for looking at data in the x-y plane. 3D mode allows the camera to be positioned anywhere and pointing at any place in 3D.

inline void **SetMode**(vtkm::rendering::Camera::Mode mode)

The mode of the camera (2D or 3D).

`vtkm::rendering::Camera` can be set to a 2D or 3D mode. 2D mode is used for looking at data in the x-y plane. 3D mode allows the camera to be positioned anywhere and pointing at any place in 3D.

inline void **SetModeTo3D**()

The mode of the camera (2D or 3D).

`vtkm::rendering::Camera` can be set to a 2D or 3D mode. 2D mode is used for looking at data in the x-y plane. 3D mode allows the camera to be positioned anywhere and pointing at any place in 3D.

inline void **SetModeTo2D**()

The mode of the camera (2D or 3D).

`vtkm::rendering::Camera` can be set to a 2D or 3D mode. 2D mode is used for looking at data in the x-y plane. 3D mode allows the camera to be positioned anywhere and pointing at any place in 3D.

inline vtkm::Range **GetClippingRange**() const

The clipping range of the camera.

The clipping range establishes the near and far clipping planes. These clipping planes are parallel to the viewing plane. The planes are defined by simply specifying the distance from the viewpoint. Renderers can (and usually do) remove any geometry closer than the near plane and further than the far plane.

For precision purposes, it is best to place the near plane as far away as possible (while still being in front of any geometry). The far plane usually has less effect on the depth precision, so can be placed well far behind the geometry.

inline void **SetClippingRange**(vtkm::Float32 nearPlane, vtkm::Float32 farPlane)

The clipping range of the camera.

The clipping range establishes the near and far clipping planes. These clipping planes are parallel to the viewing plane. The planes are defined by simply specifying the distance from the viewpoint. Renderers can (and usually do) remove any geometry closer than the near plane and further than the far plane.

For precision purposes, it is best to place the near plane as far away as possible (while still being in front of any geometry). The far plane usually has less effect on the depth precision, so can be placed well far behind the geometry.

inline void **SetClippingRange**(vtkm::Float64 nearPlane, vtkm::Float64 farPlane)

The clipping range of the camera.

The clipping range establishes the near and far clipping planes. These clipping planes are parallel to the viewing plane. The planes are defined by simply specifying the distance from the viewpoint. Renderers can (and usually do) remove any geometry closer than the near plane and further than the far plane.

For precision purposes, it is best to place the near plane as far away as possible (while still being in front of any geometry). The far plane usually has less effect on the depth precision, so can be placed well far behind the geometry.

```
inline void SetClippingRange(const vtkm::Range &nearFarRange)
```

The clipping range of the camera.

The clipping range establishes the near and far clipping planes. These clipping planes are parallel to the viewing plane. The planes are defined by simply specifying the distance from the viewpoint. Renderers can (and usually do) remove any geometry closer than the near plane and further than the far plane.

For precision purposes, it is best to place the near plane as far away as possible (while still being in front of any geometry). The far plane usually has less effect on the depth precision, so can be placed well far behind the geometry.

```
inline void GetViewport(vtkm::Float32 &left, vtkm::Float32 &right, vtkm::Float32 &bottom,  
                        vtkm::Float32 &top) const
```

The viewport of the projection.

The projection of the camera can be offset to be centered around a subset of the rendered image. This is established with a “viewport,” which is defined by the left/right and bottom/top of this viewport. The values of the viewport are relative to the rendered image’s bounds. The left and bottom of the image are at -1 and the right and top are at 1.

```
inline void GetViewport(vtkm::Float64 &left, vtkm::Float64 &right, vtkm::Float64 &bottom,  
                        vtkm::Float64 &top) const
```

The viewport of the projection.

The projection of the camera can be offset to be centered around a subset of the rendered image. This is established with a “viewport,” which is defined by the left/right and bottom/top of this viewport. The values of the viewport are relative to the rendered image’s bounds. The left and bottom of the image are at -1 and the right and top are at 1.

```
inline vtkm::Bounds GetViewport() const
```

The viewport of the projection.

The projection of the camera can be offset to be centered around a subset of the rendered image. This is established with a “viewport,” which is defined by the left/right and bottom/top of this viewport. The values of the viewport are relative to the rendered image’s bounds. The left and bottom of the image are at -1 and the right and top are at 1.

```
inline void SetViewport(vtkm::Float32 left, vtkm::Float32 right, vtkm::Float32 bottom, vtkm::Float32 top)
```

The viewport of the projection.

The projection of the camera can be offset to be centered around a subset of the rendered image. This is established with a “viewport,” which is defined by the left/right and bottom/top of this viewport. The values of the viewport are relative to the rendered image’s bounds. The left and bottom of the image are at -1 and the right and top are at 1.

```
inline void SetViewport(vtkm::Float64 left, vtkm::Float64 right, vtkm::Float64 bottom, vtkm::Float64 top)
```

The viewport of the projection.

The projection of the camera can be offset to be centered around a subset of the rendered image. This is established with a “viewport,” which is defined by the left/right and bottom/top of this viewport. The values of the viewport are relative to the rendered image’s bounds. The left and bottom of the image are at -1 and the right and top are at 1.

```
inline void SetViewport(const vtkm::Bounds &viewportBounds)
```

The viewport of the projection.

The projection of the camera can be offset to be centered around a subset of the rendered image. This is established with a “viewport,” which is defined by the left/right and bottom/top of this viewport. The values

of the viewport are relative to the rendered image's bounds. The left and bottom of the image are at -1 and the right and top are at 1.

inline const vtkm::Vec3f\_32 &GetLookAt() const

The focal point the camera is looking at in 3D mode.

When in 3D mode, the camera is set up to be facing the LookAt position. If LookAt is set, the mode is changed to 3D mode.

inline void SetLookAt(const vtkm::Vec3f\_32 &lookAt)

The focal point the camera is looking at in 3D mode.

When in 3D mode, the camera is set up to be facing the LookAt position. If LookAt is set, the mode is changed to 3D mode.

inline void SetLookAt(const vtkm::Vec<Float64, 3> &lookAt)

The focal point the camera is looking at in 3D mode.

When in 3D mode, the camera is set up to be facing the LookAt position. If LookAt is set, the mode is changed to 3D mode.

inline const vtkm::Vec3f\_32 &GetPosition() const

The spatial position of the camera in 3D mode.

When in 3D mode, the camera is modeled to be at a particular location. If Position is set, the mode is changed to 3D mode.

inline void SetPosition(const vtkm::Vec3f\_32 &position)

The spatial position of the camera in 3D mode.

When in 3D mode, the camera is modeled to be at a particular location. If Position is set, the mode is changed to 3D mode.

inline void SetPosition(const vtkm::Vec3f\_64 &position)

The spatial position of the camera in 3D mode.

When in 3D mode, the camera is modeled to be at a particular location. If Position is set, the mode is changed to 3D mode.

inline const vtkm::Vec3f\_32 &GetViewUp() const

The up orientation of the camera in 3D mode.

When in 3D mode, the camera is modeled to be at a particular location and looking at a particular spot. The view up vector orients the rotation of the image so that the top of the image is in the direction pointed to by view up. If ViewUp is set, the mode is changed to 3D mode.

inline void SetViewUp(const vtkm::Vec3f\_32 &viewUp)

The up orientation of the camera in 3D mode.

When in 3D mode, the camera is modeled to be at a particular location and looking at a particular spot. The view up vector orients the rotation of the image so that the top of the image is in the direction pointed to by view up. If ViewUp is set, the mode is changed to 3D mode.

inline void SetViewUp(const vtkm::Vec3f\_64 &viewUp)

The up orientation of the camera in 3D mode.

When in 3D mode, the camera is modeled to be at a particular location and looking at a particular spot. The view up vector orients the rotation of the image so that the top of the image is in the direction pointed to by view up. If ViewUp is set, the mode is changed to 3D mode.



inline vtkm::Float32 **GetXScale**() const

The xscale of the camera.

The xscale forces the 2D curves to be full-frame

Setting the xscale changes the mode to 2D.

inline void **SetXScale**(vtkm::Float32 xscale)

The xscale of the camera.

The xscale forces the 2D curves to be full-frame

Setting the xscale changes the mode to 2D.

inline void **SetXScale**(vtkm::Float64 xscale)

The xscale of the camera.

The xscale forces the 2D curves to be full-frame

Setting the xscale changes the mode to 2D.

inline vtkm::Float32 **GetFieldOfView**() const

The field of view angle.

The field of view defines the angle (in degrees) that are visible from the camera position.

Setting the field of view changes the mode to 3D.

inline void **SetFieldOfView**(vtkm::Float32 fov)

The field of view angle.

The field of view defines the angle (in degrees) that are visible from the camera position.

Setting the field of view changes the mode to 3D.

inline void **SetFieldOfView**(vtkm::Float64 fov)

The field of view angle.

The field of view defines the angle (in degrees) that are visible from the camera position.

Setting the field of view changes the mode to 3D.

void **Pan**(vtkm::Float32 dx, vtkm::Float32 dy)

Pans the camera.

Panning the camera shifts the view horizontally and/or vertically with respect to the image plane.

Panning works in either 2D or 3D mode.

inline void **Pan**(vtkm::Float64 dx, vtkm::Float64 dy)

Pans the camera.

Panning the camera shifts the view horizontally and/or vertically with respect to the image plane.

Panning works in either 2D or 3D mode.

inline void **Pan**(vtkm::Vec2f\_32 direction)

Pans the camera.

Panning the camera shifts the view horizontally and/or vertically with respect to the image plane.

Panning works in either 2D or 3D mode.

inline void **Pan**(vtkm::Vec2f\_64 direction)

Pans the camera.

Panning the camera shifts the view horizontally and/or vertically with respect to the image plane.

Panning works in either 2D or 3D mode.

inline vtkm::Vec2f\_32 **GetPan**() const

Pans the camera.

Panning the camera shifts the view horizontally and/or vertically with respect to the image plane.

Panning works in either 2D or 3D mode.

void **Zoom**(vtkm::Float32 zoom)

Zooms the camera in or out.

Zooming the camera scales everything in the image up or down. Positive zoom makes the geometry look bigger or closer. Negative zoom has the opposite effect. A zoom of 0 has no effect.

Zooming works in either 2D or 3D mode.

inline void **Zoom**(vtkm::Float64 zoom)

Zooms the camera in or out.

Zooming the camera scales everything in the image up or down. Positive zoom makes the geometry look bigger or closer. Negative zoom has the opposite effect. A zoom of 0 has no effect.

Zooming works in either 2D or 3D mode.

inline vtkm::Float32 **GetZoom**() const

Zooms the camera in or out.

Zooming the camera scales everything in the image up or down. Positive zoom makes the geometry look bigger or closer. Negative zoom has the opposite effect. A zoom of 0 has no effect.

Zooming works in either 2D or 3D mode.

void **TrackballRotate**(vtkm::Float32 startX, vtkm::Float32 startY, vtkm::Float32 endX, vtkm::Float32 endY)

Moves the camera as if a point was dragged along a sphere.

*TrackballRotate()* takes the normalized screen coordinates (in the range -1 to 1) and rotates the camera around the **LookAt** position. The rotation first projects the points to a sphere around the **LookAt** position. The camera is then rotated as if the start point was dragged to the end point along with the world.

*TrackballRotate()* changes the mode to 3D.

inline void **TrackballRotate**(vtkm::Float64 startX, vtkm::Float64 startY, vtkm::Float64 endX, vtkm::Float64 endY)

Moves the camera as if a point was dragged along a sphere.

*TrackballRotate()* takes the normalized screen coordinates (in the range -1 to 1) and rotates the camera around the **LookAt** position. The rotation first projects the points to a sphere around the **LookAt** position. The camera is then rotated as if the start point was dragged to the end point along with the world.

*TrackballRotate()* changes the mode to 3D.

void **ResetToBounds**(const vtkm::Bounds &dataBounds)

Set up the camera to look at geometry.

*ResetToBounds()* takes a *vtkm::Bounds* structure containing the bounds in 3D space that contain the geometry being rendered. This method sets up the camera so that it is looking at this region in space. The

view direction is preserved. `ResetToBounds()` can also take optional padding that the viewpoint should preserve around the object. Padding is specified as the fraction of the bounds to add as padding.

void **ResetToBounds**(const vtkm::Bounds &dataBounds, vtkm::Float64 dataViewPadding)

Set up the camera to look at geometry.

`ResetToBounds()` takes a `vtkm::Bounds` structure containing the bounds in 3D space that contain the geometry being rendered. This method sets up the camera so that it is looking at this region in space. The view direction is preserved. `ResetToBounds()` can also take optional padding that the viewpoint should preserve around the object. Padding is specified as the fraction of the bounds to add as padding.

void **ResetToBounds**(const vtkm::Bounds &dataBounds, vtkm::Float64 XDataViewPadding, vtkm::Float64 YDataViewPadding, vtkm::Float64 ZDataViewPadding)

Set up the camera to look at geometry.

`ResetToBounds()` takes a `vtkm::Bounds` structure containing the bounds in 3D space that contain the geometry being rendered. This method sets up the camera so that it is looking at this region in space. The view direction is preserved. `ResetToBounds()` can also take optional padding that the viewpoint should preserve around the object. Padding is specified as the fraction of the bounds to add as padding.

void **Roll**(vtkm::Float32 angleDegrees)

Roll the camera.

Rotates the camera around the view direction by the given angle. The angle is given in degrees.

Roll is currently only supported for 3D cameras.

inline void **Roll**(vtkm::Float64 angleDegrees)

Roll the camera.

Rotates the camera around the view direction by the given angle. The angle is given in degrees.

Roll is currently only supported for 3D cameras.

void **Azimuth**(vtkm::Float32 angleDegrees)

Rotate the camera about the view up vector centered at the focal point.

Note that the view up vector is whatever was set via `SetViewUp()`, and is not necessarily perpendicular to the direction of projection. The angle is given in degrees.

`Azimuth()` only makes sense for 3D cameras, so the camera mode will be set to 3D when this method is called.

inline void **Azimuth**(vtkm::Float64 angleDegrees)

Rotate the camera about the view up vector centered at the focal point.

Note that the view up vector is whatever was set via `SetViewUp()`, and is not necessarily perpendicular to the direction of projection. The angle is given in degrees.

`Azimuth()` only makes sense for 3D cameras, so the camera mode will be set to 3D when this method is called.

void **Elevation**(vtkm::Float32 angleDegrees)

Rotate the camera vertically around the focal point.

Specifically, this rotates the camera about the cross product of the negative of the direction of projection and the view up vector, using the focal point (`LookAt`) as the center of rotation. The angle is given in degrees.

`Elevation()` only makes sense for 3D cameras, so the camera mode will be set to 3D when this method is called.

inline void **Elevation**(vtkm::Float64 angleDegrees)

Rotate the camera vertically around the focal point.

Specifically, this rotates the camera about the cross product of the negative of the direction of projection and the view up vector, using the focal point (**LookAt**) as the center of rotation. The angle is given in degrees.

**Elevation()** only makes sense for 3D cameras, so the camera mode will be set to 3D when this method is called.

void **Dolly**(vtkm::Float32 value)

Move the camera toward or away from the focal point.

Specifically, this divides the camera's distance from the focal point (**LookAt**) by the given value. Use a value greater than one to dolly in toward the focal point, and use a value less than one to dolly-out away from the focal point.

**Dolly()** has a similar effect as **Zoom()** since an object will appear larger when the camera is closer. However, because you are moving the camera, **Dolly()** can change the perspective relative to objects such as moving inside an object for an interior perspective whereas **Zoom()** will just change the size of the visible objects.

**Dolly()** only makes sense for 3D cameras, so the camera mode will be set to 3D when this method is called.

inline void **Dolly**(vtkm::Float64 value)

Move the camera toward or away from the focal point.

Specifically, this divides the camera's distance from the focal point (**LookAt**) by the given value. Use a value greater than one to dolly in toward the focal point, and use a value less than one to dolly-out away from the focal point.

**Dolly()** has a similar effect as **Zoom()** since an object will appear larger when the camera is closer. However, because you are moving the camera, **Dolly()** can change the perspective relative to objects such as moving inside an object for an interior perspective whereas **Zoom()** will just change the size of the visible objects.

**Dolly()** only makes sense for 3D cameras, so the camera mode will be set to 3D when this method is called.

inline void **GetViewRange2D**(vtkm::Float32 &left, vtkm::Float32 &right, vtkm::Float32 &bottom, vtkm::Float32 &top) const

The viewable region in the x-y plane.

When the camera is in 2D, it is looking at some region of the x-y plane. The region being looked at is defined by the range in x (determined by the left and right sides) and by the range in y (determined by the bottom and top sides).

**SetViewRange2D()** changes the camera mode to 2D.

inline vtkm::Bounds **GetViewRange2D**() const

The viewable region in the x-y plane.

When the camera is in 2D, it is looking at some region of the x-y plane. The region being looked at is defined by the range in x (determined by the left and right sides) and by the range in y (determined by the bottom and top sides).

**SetViewRange2D()** changes the camera mode to 2D.

inline void **SetViewRange2D**(vtkm::Float32 left, vtkm::Float32 right, vtkm::Float32 bottom, vtkm::Float32 top)

The viewable region in the x-y plane.

When the camera is in 2D, it is looking at some region of the x-y plane. The region being looked at is defined by the range in x (determined by the left and right sides) and by the range in y (determined by the bottom and top sides).

`SetViewRange2D()` changes the camera mode to 2D.

```
inline void SetViewRange2D(vtkm::Float64 left, vtkm::Float64 right, vtkm::Float64 bottom, vtkm::Float64
                           top)
```

The viewable region in the x-y plane.

When the camera is in 2D, it is looking at some region of the x-y plane. The region being looked at is defined by the range in x (determined by the left and right sides) and by the range in y (determined by the bottom and top sides).

`SetViewRange2D()` changes the camera mode to 2D.

```
inline void SetViewRange2D(const vtkm::Range &xRange, const vtkm::Range &yRange)
```

The viewable region in the x-y plane.

When the camera is in 2D, it is looking at some region of the x-y plane. The region being looked at is defined by the range in x (determined by the left and right sides) and by the range in y (determined by the bottom and top sides).

`SetViewRange2D()` changes the camera mode to 2D.

A `vtkm::rendering::Camera` operates in one of two major modes: 2D mode or 3D mode. 2D mode is designed for looking at flat geometry (or close to flat geometry) that is parallel to the x-y plane. 3D mode provides the freedom to place the camera anywhere in 3D space. The different modes can be set with `vtkm::rendering::Camera::SetModeTo2D()` and `vtkm::rendering::Camera::SetModeTo3D()`, respectively. The interaction with the camera in these two modes is very different.

### 11.6.1 Common Camera Controls

Some camera controls operate relative to the rendered image and are common among the 2D and 3D camera modes.

#### Pan

A camera pan moves the viewpoint left, right, up, or down. A camera pan is performed by calling the `vtkm::cont::Camera::Pan()` method. `vtkm::cont::Camera::Pan()` takes two arguments: the amount to pan in x and the amount to pan in y.

The pan is given with respect to the projected space. So a pan of 1 in the x direction moves the camera to focus on the right edge of the image whereas a pan of  $-1$  in the x direction moves the camera to focus on the left edge of the image.

Example 9: Panning the camera.

```
view.GetCamera().Pan(deltaX, deltaY);
```

## Zoom

A camera zoom draws the geometry larger or smaller. A camera zoom is performed by calling the `vtkm::rendering::Camera::Zoom()` method. `vtkm::rendering::Camera::Zoom()` takes a single argument specifying the zoom factor. A positive number draws the geometry larger (zoom in), and larger zoom factor results in larger geometry. Likewise, a negative number draws the geometry smaller (zoom out). A zoom factor of 0 has no effect.

Example 10: Zooming the camera.

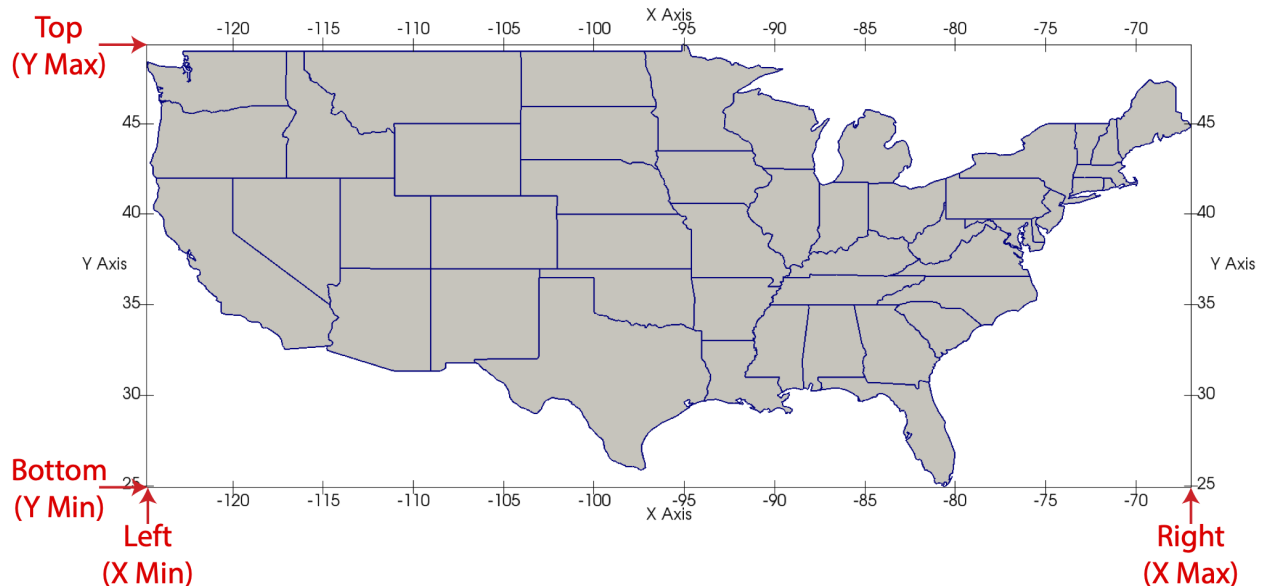
```
view.GetCamera().Zoom(zoomFactor);
```

## 11.6.2 2D Camera Mode

The 2D camera is restricted to looking at some region of the x-y plane.

### View Range

The vantage point of a 2D camera can be specified by simply giving the region in the x-y plane to look at. This region is specified by calling `vtkm::rendering::Camera::SetViewRange2D()`. This method takes the left, right, bottom, and top of the region to view. Typically these are set to the range of the geometry in world space as shown in Figure 3.

Figure 3: The view range bounds to give a `vtkm::rendering::Camera`.

### 11.6.3 3D Camera Mode

The 3D camera is a free-form camera that can be placed anywhere in 3D space and can look in any direction. The projection of the 3D camera is based on the pinhole camera model in which all viewing rays intersect a single point. This single point is the camera's position.

#### Position and Orientation

The position of the camera, which is the point where the observer is viewing the scene, can be set with the `vtkm::rendering::Camera::SetPosition()` method. The direction the camera is facing is specified by giving a position to focus on. This is called either the “look at” point or the focal point and is specified with the `vtkm::rendering::Camera::SetLookAt()` method. Figure 4 shows the relationship between the position and look at points.

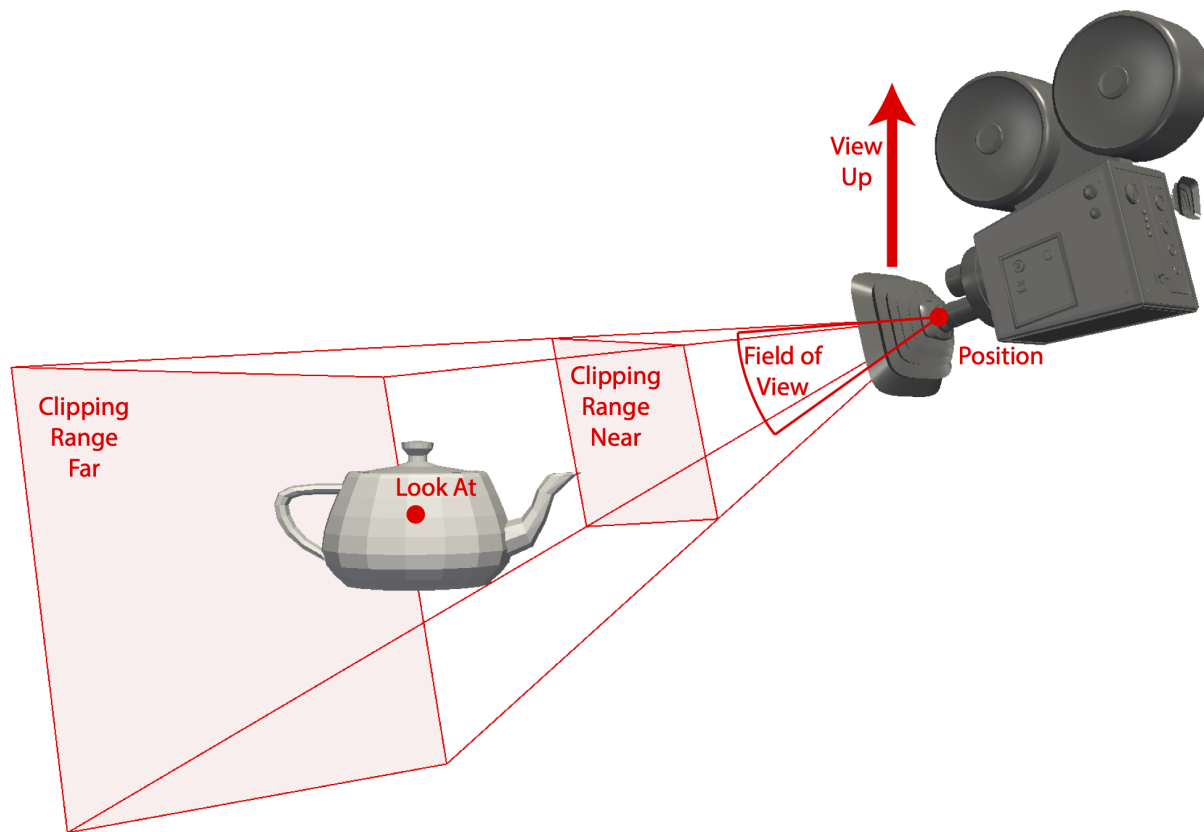


Figure 4: The position and orientation parameters for a `vtkm::rendering::Camera`.

In addition to specifying the direction to point the camera, the camera must also know which direction is considered “up.” This is specified with the view up vector using the `vtkm::rendering::Camera::SetViewUp()` method. The view up vector points from the camera position (in the center of the image) to the top of the image. The view up vector in relation to the camera position and orientation is shown in Figure 4.

Another important parameter for the camera is its field of view. The field of view specifies how wide of a region the camera can see. It is specified by giving the angle in degrees of the cone of visible region emanating from the pinhole of the camera to the `vtkm::rendering::Camera::SetFieldOfView()` method. The field of view angle in relation to the camera orientation is shown in Figure 4. A field of view angle of  $60^\circ$  usually works well.

Finally, the camera must specify a clipping region that defines the valid range of depths for the object. This is a pair of planes parallel to the image that all visible data must lie in. Each of these planes is defined simply by their distance to the camera position. The near clip plane is closer to the camera and must be in front of all geometry. The far clip plane is further from the camera and must be behind all geometry. The distance to both the near and far planes are specified with the `vtkm::rendering::Camera::SetClippingRange()` method. Figure 4 shows the clipping planes in relationship to the camera position and orientation.

Example 11: Directly setting `vtkm::rendering::Camera` position and orientation.

```
1 camera.SetPosition(vtkm::make_Vec(10.0, 6.0, 6.0));
2 camera.SetLookAt(vtkm::make_Vec(0.0, 0.0, 0.0));
3 camera.SetViewUp(vtkm::make_Vec(0.0, 1.0, 0.0));
4 camera.SetFieldOfView(60.0);
5 camera.SetClippingRange(0.1, 100.0);
```

## Movement

In addition to specifically setting the position and orientation of the camera, `vtkm::rendering::Camera` contains several convenience methods that move the camera relative to its position and look at point.

Two such methods are elevation and azimuth, which move the camera around the sphere centered at the look at point. `vtkm::rendering::Camera::Elevation()` raises or lowers the camera. Positive values raise the camera up (in the direction of the view up vector) whereas negative values lower the camera down. `vtkm::rendering::Camera::Azimuth()` moves the camera around the look at point to the left or right. Positive values move the camera to the right whereas negative values move the camera to the left. Both `vtkm::rendering::Camera::Elevation()` and `vtkm::rendering::Camera::Azimuth()` specify the amount of rotation in terms of degrees. Figure 5 shows the relative movements of `vtkm::rendering::Camera::Elevation()` and `vtkm::rendering::Camera::Azimuth()`.

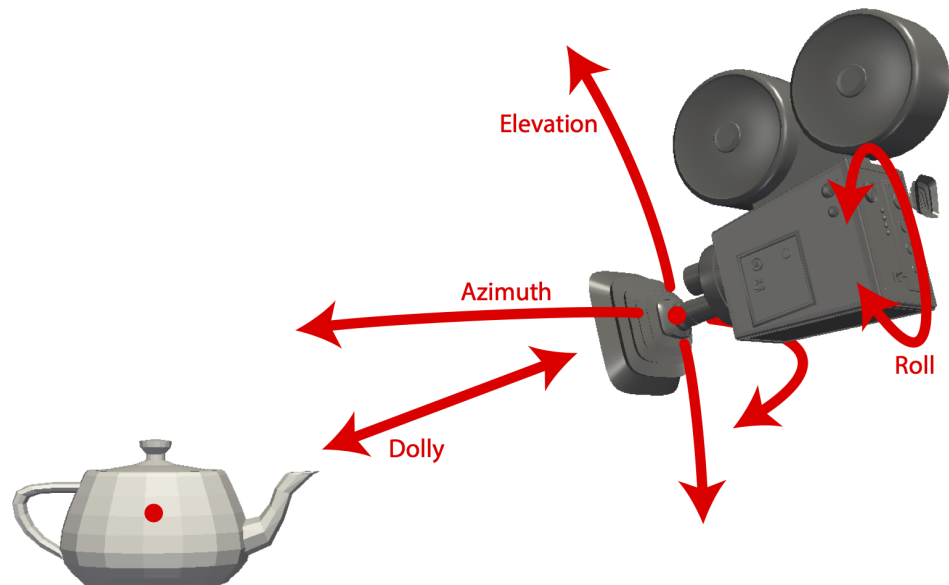


Figure 5: `vtkm::rendering::Camera` movement functions relative to position and orientation.



Example 12: Moving the camera around the look at point.

```

1 view.GetCamera().Azimuth(45.0);
2 view.GetCamera().Elevation(45.0);

```

### Common Errors

The `vtkm::rendering::Camera::Elevation()` and `vtkm::rendering::Camera::Azimuth()` methods change the position of the camera, but not the view up vector. This can cause some wild camera orientation changes when the direction of the camera view is near parallel to the view up vector, which often happens when the elevation is raised or lowered by about 90 degrees.

In addition to rotating the camera around the look at point, you can move the camera closer or further from the look at point. This is done with the `vtkm::rendering::Camera::Dolly()` method. The `vtkm::rendering::Camera::Dolly()` method takes a single value that is the factor to scale the distance between camera and look at point. Values greater than one move the camera away, values less than one move the camera closer. The direction of dolly movement is shown in Figure 5.

Finally, the `vtkm::rendering::Camera::Roll()` method rotates the camera around the viewing direction. It has the effect of rotating the rendered image. The `vtkm::rendering::Camera::Roll()` method takes a single value that is the angle to rotate in degrees. The direction of roll movement is shown in Figure 5.

### Reset

Setting a specific camera position and orientation can be frustrating, particularly when the size, shape, and location of the geometry is not known a priori. Typically this involves querying the data and finding a good camera orientation.

To make this process simpler, the `vtkm::rendering::Camera::ResetToBounds()` convenience method automatically positions the camera based on the spatial bounds of the geometry. The most expedient method to find the spatial bounds of the geometry being rendered is to get the `vtkm::rendering::Scene` object and call `vtkm::rendering::Scene::GetSpatialBounds()`. The `vtkm::rendering::Scene` object can be retrieved from the `vtkm::rendering::View`, which, as described in Section 11.4 (Views), is the central object for managing rendering.

Example 13: Resetting a `vtkm::rendering::Camera` to view geometry.

```

1 void ResetCamera(vtkm::rendering::View& view)
2 {
3     vtkm::Bounds bounds = view.GetScene().GetSpatialBounds();
4     view.GetCamera().ResetToBounds(bounds);
5 }

```

The `vtkm::rendering::Camera::ResetToBounds()` method operates by placing the look at point in the center of the bounds and then placing the position of the camera relative to that look at point. The position is such that the view direction is the same as before the call to `vtkm::rendering::Camera::ResetToBounds()` and the distance between the camera position and look at point has the bounds roughly fill the rendered image. This behavior is a convenient way to update the camera to make the geometry most visible while still preserving the viewing position. If you want to reset the camera to a new viewing angle, it is best to set the camera to be pointing in the right direction and then calling `vtkm::rendering::Camera::ResetToBounds()` to adjust the position.

Example 14: Resetting a `vtkm::rendering::Camera` to be axis aligned.

```
1 view.GetCamera().SetPosition(vtkm::make_Vec(0.0, 0.0, 0.0));
2 view.GetCamera().SetLookAt(vtkm::make_Vec(0.0, 0.0, -1.0));
3 view.GetCamera().SetViewUp(vtkm::make_Vec(0.0, 1.0, 0.0));
4 vtkm::Bounds bounds = view.GetScene().GetSpatialBounds();
5 view.GetCamera().ResetToBounds(bounds);
```

## 11.7 Interactive Rendering

So far in our description of VTK-m's rendering capabilities we have talked about doing rendering of fixed scenes. However, an important use case of scientific visualization is to provide an interactive rendering system to explore data. In this case, you want to render into a GUI application that lets the user interact manipulate the view. The full design of a 3D visualization application is well outside the scope of this book, but we discuss in general terms what you need to plug VTK-m's rendering into such a system.

In this section we discuss two important concepts regarding interactive rendering. First, we need to write images into a GUI while they are being rendered. Second, we want to translate user interaction to camera movement.

### 11.7.1 Rendering Into a GUI

Before being able to show rendering to a user, we need a system rendering context in which to push the images. In this section we demonstrate the display of images using the OpenGL rendering system, which is common for scientific visualization applications. That said, you could also use other rendering systems like DirectX or even paste images into a blank widget.

Creating an OpenGL context varies depending on the OS platform you are using. If you do not already have an application you want to integrate with VTK-m's rendering, you may wish to start with graphics utility API such as GLUT or GLFW. The process of initializing an OpenGL context is not discussed here.

The process of rendering into an OpenGL context is straightforward. First call `vtkm::rendering::View::Paint()` on the `vtkm::rendering::View` object to do the actual rendering. Second, get the image color data out of the `vtkm::rendering::View`'s `vtkm::rendering::Canvas` object. This is done by calling `vtkm::rendering::Canvas::GetColorBuffer()`. This will return a `vtkm::cont::ArrayHandle` object containing the image's pixel color data. (`vtkm::cont::ArrayHandle` is discussed in detail in [Chapter 17 \(Basic Array Handles\)](#) and subsequent chapters.) A raw pointer can be pulled out of this `vtkm::cont::ArrayHandle` by casting it to the `vtkm::cont::ArrayHandleBase` subclass and calling the `vtkm::cont::ArrayHandleBase::GetReadPointer()` method on that. Third, the pixel color data are pasted into the OpenGL render context. There are multiple ways to do so, but the most straightforward way is to use the `glDrawPixels` function provided by OpenGL. Fourth, swap the OpenGL buffers. The method to swap OpenGL buffers varies by OS platform. The aforementioned graphics libraries GLUT and GLFW each provide a function for doing so.

Example 15: Rendering a `vtkm::rendering::View` and pasting the result to an active OpenGL context.

```
1 view.Paint();
2
3 // Get the color buffer containing the rendered image.
4 vtkm::cont::ArrayHandle<vtkm::Vec4f_32> colorBuffer =
```

(continues on next page)

(continued from previous page)

```

5     view.GetCanvas().GetColorBuffer();
6
7     // Pull the C array out of the arrayhandle.
8     const void* colorArray =
9         vtkm::cont::ArrayHandleBasic<vtkm::Vec4f_32>(colorBuffer).GetReadPointer();
10
11    // Write the C array to an OpenGL buffer.
12    glDrawPixels((GLint)view.GetCanvas().GetWidth(),
13                (GLint)view.GetCanvas().GetHeight(),
14                GL_RGBA,
15                GL_FLOAT,
16                colorArray);
17
18    // Swap the OpenGL buffers (system dependent).

```

## 11.7.2 Camera Movement

When interactively manipulating the camera in a windowing system, the camera is usually moved in response to mouse movements. Typically, mouse movements are detected through callbacks from the windowing system back to your application. Once again, the details on how this works depend on your windowing system. The assumption made in this section is that through the windowing system you will be able to track the x-y pixel location of the mouse cursor at the beginning of the movement and the end of the movement. Using these two pixel coordinates, as well as the current width and height of the render space, we can make several typical camera movements.

### Common Errors

Pixel coordinates in VTK-m's rendering system originate in the lower-left corner of the image. However, windowing systems generally report mouse coordinates with the origin in the *upper*-left corner. The upshot is that the y coordinates will have to be reversed when translating mouse coordinates to VTK-m image coordinates. This inverting is present in all the following examples.

### Interactive Rotate

A common and important mode of interaction with 3D views is to allow the user to rotate the object under inspection by dragging the mouse. To facilitate this type of interactive rotation, `vtkm::rendering::Camera` provides a convenience method named `vtkm::rendering::Camera::TrackballRotate()`. It takes a start and end position of the mouse on the image and rotates viewpoint as if the user grabbed a point on a sphere centered in the image at the start position and moved under the end position.

The `vtkm::rendering::Camera::TrackballRotate()` method is typically called from within a mouse movement callback. The callback must record the pixel position from the last event and the new pixel position of the mouse. Those pixel positions must be normalized to the range -1 to 1 where the position (-1,-1) refers to the lower left of the image and (1,1) refers to the upper right of the image. The following example demonstrates the typical operations used to establish rotations when dragging the mouse.

Example 16: Interactive rotations through mouse dragging with  
`vtkm::rendering::Camera::TrackballRotate()`.

```

1 void DoMouseRotate(vtkm::rendering::View& view,
2                   vtkm::Id mouseStartX,

```

(continues on next page)

(continued from previous page)

```

3         vtkm::Id mouseStartY,
4         vtkm::Id mouseEndX,
5         vtkm::Id mouseEndY)
6     {
7         vtkm::Id screenWidth = view.GetCanvas().GetWidth();
8         vtkm::Id screenHeight = view.GetCanvas().GetHeight();
9
10        // Convert the mouse position coordinates, given in pixels from 0 to
11        // width/height, to normalized screen coordinates from -1 to 1. Note that y
12        // screen coordinates are usually given from the top down whereas our
13        // geometry transforms are given from bottom up, so you have to reverse the y
14        // coordiantes.
15        vtkm::Float32 startX = (2.0f * mouseStartX) / screenWidth - 1.0f;
16        vtkm::Float32 startY = -((2.0f * mouseStartY) / screenHeight - 1.0f);
17        vtkm::Float32 endX = (2.0f * mouseEndX) / screenWidth - 1.0f;
18        vtkm::Float32 endY = -((2.0f * mouseEndY) / screenHeight - 1.0f);
19
20        view.GetCamera().TrackballRotate(startX, startY, endX, endY);
21    }

```

## Interactive Pan

Panning can be performed by calling `vtkm::rendering::Camera::Pan()` with the translation relative to the width and height of the canvas. For the translation to track the movement of the mouse cursor, simply scale the pixels the mouse has traveled by the width and height of the image.

Example 17: Pan the view based on mouse movements.

```

1 void DoMousePan(vtkm::rendering::View& view,
2                 vtkm::Id mouseStartX,
3                 vtkm::Id mouseStartY,
4                 vtkm::Id mouseEndX,
5                 vtkm::Id mouseEndY)
6 {
7     vtkm::Id screenWidth = view.GetCanvas().GetWidth();
8     vtkm::Id screenHeight = view.GetCanvas().GetHeight();
9
10    // Convert the mouse position coordinates, given in pixels from 0 to
11    // width/height, to normalized screen coordinates from -1 to 1. Note that y
12    // screen coordinates are usually given from the top down whereas our
13    // geometry transforms are given from bottom up, so you have to reverse the y
14    // coordiantes.
15    vtkm::Float32 startX = (2.0f * mouseStartX) / screenWidth - 1.0f;
16    vtkm::Float32 startY = -((2.0f * mouseStartY) / screenHeight - 1.0f);
17    vtkm::Float32 endX = (2.0f * mouseEndX) / screenWidth - 1.0f;
18    vtkm::Float32 endY = -((2.0f * mouseEndY) / screenHeight - 1.0f);
19
20    vtkm::Float32 deltaX = endX - startX;
21    vtkm::Float32 deltaY = endY - startY;
22
23    view.GetCamera().Pan(deltaX, deltaY);

```

(continues on next page)

(continued from previous page)

24 }  
}

## Interactive Zoom

Zooming can be performed by calling `vtkm::rendering::Camera::Zoom()` with a positive or negative zoom factor. When using `vtkm::rendering::Camera::Zoom()` to respond to mouse movements, a natural zoom will divide the distance traveled by the mouse pointer by the width or height of the screen as demonstrated in the following example.

Example 18: Zoom the view based on mouse movements.

```

1 void DoMouseZoom(vtkm::rendering::View& view, vtkm::Id mouseStartY, vtkm::Id mouseEndY)
2 {
3     vtkm::Id screenHeight = view.GetCanvas().GetHeight();
4
5     // Convert the mouse position coordinates, given in pixels from 0 to height,
6     // to normalized screen coordinates from -1 to 1. Note that y screen
7     // coordinates are usually given from the top down whereas our geometry
8     // transforms are given from bottom up, so you have to reverse the y
9     // coordinates.
10    vtkm::Float32 startY = -((2.0f * mouseStartY) / screenHeight - 1.0f);
11    vtkm::Float32 endY = -((2.0f * mouseEndY) / screenHeight - 1.0f);
12
13    vtkm::Float32 zoomFactor = endY - startY;
14
15    view.GetCamera().Zoom(zoomFactor);
16 }

```

## 11.8 Color Tables

An important feature of VTK-m's rendering units is the ability to pseudocolor objects based on scalar data. This technique maps each scalar to a potentially unique color. This mapping from scalars to colors is defined by a `vtkm::cont::ColorTable` object. A `vtkm::cont::ColorTable` can be specified as an optional argument when constructing a `vtkm::rendering::Actor`. (Use of `vtkm::rendering::Actor` is discussed in Section 11.1 (Scenes and Actors).)

Example 19: Specifying a `vtkm::cont::ColorTable` for a `vtkm::rendering::Actor`.

```

1 vtkm::rendering::Actor actor(surfaceData.GetCellSet(),
2                             surfaceData.GetCoordinateSystem(),
3                             surfaceData.GetField("RandomPointScalars"),
4                             vtkm::cont::ColorTable("inferno"));

```

class **ColorTable** : public `vtkm::cont::ExecutionObjectBase`

Color Table for coloring arbitrary fields.

The `vtkm::cont::ColorTable` allows for color mapping in RGB or HSV space and uses a piecewise hermite functions to allow opacity interpolation that can be piecewise constant, piecewise linear, or somewhere in-between (a modified piecewise hermite function that squishes the function according to a sharpness parameter).

For colors interpolation is handled using a piecewise linear function.

For opacity we define a piecewise function mapping. This mapping allows the addition of control points, and allows the user to control the function between the control points. A piecewise hermite curve is used between control points, based on the sharpness and midpoint parameters. A sharpness of 0 yields a piecewise linear function and a sharpness of 1 yields a piecewise constant function. The midpoint is the normalized distance between control points at which the curve reaches the median Y value. The midpoint and sharpness values specified when adding a node are used to control the transition to the next node with the last node's values being ignored.

When adding opacity nodes without an explicit midpoint and sharpness we will default to to Midpoint = 0.5 (halfway between the control points) and Sharpness = 0.0 (linear).

*ColorTable* also contains which ColorSpace should be used for interpolation. The color space is selected with the `vtkm::ColorSpace` enumeration. Currently the valid ColorSpaces are:

- RGB
- HSV
- HSVWrap
- Lab
- Diverging

In HSVWrap mode, it will take the shortest path in Hue (going back through 0 if that is the shortest way around the hue circle) whereas HSV will not go through 0 (in order to match the current functionality of `vtkLookupTable`). In Lab mode, it will take the shortest path in the Lab color space with respect to the CIE Delta E 2000 color distance measure. Diverging is a special mode where colors will pass through white when interpolating between two saturated colors.

To map a field from a `vtkm::cont::DataSet` through the color and opacity transfer functions and into a RGB or RGBA array you should use `vtkm::filter::FieldToColor`.

Note that modifications of `vtkm::cont::ColorTable` are not thread safe. You should not modify a *ColorTable* simultaneously in 2 or more threads. Also, you should not modify a *ColorTable* that might be used in the execution environment. However, the *ColorTable* can be used in multiple threads and on multiple devices as long as no modifications are made.

## Public Functions

**ColorTable**(`vtkm::cont::ColorTable::Preset` preset = `vtkm::cont::ColorTable::Preset::Default`)

Construct a color table from a preset.

Constructs a color table from a given preset, which might include a NaN color. The alpha table will have 2 entries of alpha = 1.0 with linear interpolation

Note: these are a select set of the presets you can get by providing a string identifier.

explicit **ColorTable**(const std::string &name)

Construct a color table from a preset color table.

Constructs a color table from a given preset, which might include a NaN color. The alpha table will have 2 entries of alpha = 1.0 with linear interpolation

Note: Names are case insensitive Currently supports the following color tables:

“Default” “Cool to Warm” “Cool to Warm Extended” “Viridis” “Inferno” “Plasma” “Black-Body Radiation” “X Ray” “Green” “Black - Blue - White” “Blue to Orange” “Gray to Red” “Cold and Hot” “Blue - Green - Orange” “Yellow - Gray - Blue” “Rainbow Uniform” “Jet” “Rainbow Desaturated”

explicit **ColorTable**(vtkm::ColorSpace space)

Construct a color table with a zero positions, and an invalid range.

Note: The color table will have 0 entries Note: The alpha table will have 0 entries

**ColorTable**(const vtkm::Range &range, vtkm::ColorSpace space = vtkm::ColorSpace::Lab)

Construct a color table with a 2 positions.

Note: The color table will have 2 entries of rgb = {1.0,1.0,1.0} Note: The alpha table will have 2 entries of alpha = 1.0 with linear interpolation

**ColorTable**(const vtkm::Range &range, const vtkm::Vec3f\_32 &rgb1, const vtkm::Vec3f\_32 &rgb2, vtkm::ColorSpace space = vtkm::ColorSpace::Lab)

Construct a color table with 2 positions.

Note: The alpha table will have 2 entries of alpha = 1.0 with linear interpolation

**ColorTable**(const vtkm::Range &range, const vtkm::Vec4f\_32 &rgba1, const vtkm::Vec4f\_32 &rgba2, vtkm::ColorSpace space = vtkm::ColorSpace::Lab)

Construct color and alpha and table with 2 positions.

Note: The alpha table will use linear interpolation

**ColorTable**(const std::string &name, vtkm::ColorSpace colorSpace, const vtkm::Vec3f\_64 &nanColor, const std::vector<vtkm::Float64> &rgbPoints, const std::vector<vtkm::Float64> &alphaPoints = {0.0, 1.0, 0.5, 0.0, 1.0, 1.0, 0.5, 0.0})

Construct a color table with a list of colors and alphas.

For this version you must also specify a name.

This constructor is mostly used for presets.

bool **LoadPreset**(const std::string &name)

Load a preset color table.

Removes all existing all values in both color and alpha tables, and will reset the NaN Color if the color table has that information. Will not modify clamping, below, and above range state.

Note: Names are case insensitive

Currently supports the following color tables: “Default” “Cool to Warm” “Cool to Warm Extended” “Viridis” “Inferno” “Plasma” “Black-Body Radiation” “X Ray” “Green” “Black - Blue - White” “Blue to Orange” “Gray to Red” “Cold and Hot” “Blue - Green - Orange” “Yellow - Gray - Blue” “Rainbow Uniform” “Jet” “Rainbow Desaturated”

*ColorTable* **MakeDeepCopy**()

Make a deep copy of the current color table.

The *ColorTable* is implemented so that all stack based copies are ‘shallow’ copies. This means that they all alter the same internal instance. But sometimes you need to make an actual fully independent copy.

inline void **SetClampingOn**()

If clamping is disabled values that lay out side the color table range are colored based on Below and Above settings.

By default clamping is enabled

void **SetBelowRangeColor**(const vtkm::Vec3f\_32 &c)

Color to use when clamping is disabled for any value that is below the given range.

Default value is {0,0,0}



void **SetAboveRangeColor**(const vtkm::Vec3f\_32 &c)  
 Color to use when clamping is disabled for any value that is above the given range.  
 Default value is {0,0,0}

void **Clear**()  
 Remove all existing values in both color and alpha tables.  
 Does not remove the clamping, below, and above range state or colors.

void **ClearColors**()  
 Remove only color table values.

void **ClearAlpha**()  
 Remove only alpha table values.

void **ReverseColors**()  
 Reverse the rgb values inside the color table.

void **ReverseAlpha**()  
 Reverse the alpha, mid, and sharp values inside the opacity table.  
 Note: To keep the shape correct the mid and sharp values of the last node are not included in the reversal

const vtkm::Range &**GetRange**() const  
 Returns min and max position of all function points.

void **RescaleToRange**(const vtkm::Range &range)  
 Rescale the color and opacity transfer functions to match the input range.

vtkm::Int32 **AddPoint**(vtkm::Float64 x, const vtkm::Vec3f\_32 &rgb)  
 Adds a point to the color function.  
 If the point already exists, it will be updated to the new value.  
 Note: rgb values need to be between 0 and 1.0 (inclusive). Return the index of the point (0 based), or -1 on error.

vtkm::Int32 **AddPointHSV**(vtkm::Float64 x, const vtkm::Vec3f\_32 &hsv)  
 Adds a point to the color function.  
 If the point already exists, it will be updated to the new value.  
 Note: hsv values need to be between 0 and 1.0 (inclusive). Return the index of the point (0 based), or -1 on error.

vtkm::Int32 **AddSegment**(vtkm::Float64 x1, const vtkm::Vec3f\_32 &rgb1, vtkm::Float64 x2, const vtkm::Vec3f\_32 &rgb2)  
 Add a line segment to the color function.  
 All points which lay between x1 and x2 (inclusive) are removed from the function.  
 Note: rgb1, and rgb2 values need to be between 0 and 1.0 (inclusive). Return the index of the point x1 (0 based), or -1 on error.

vtkm::Int32 **AddSegmentHSV**(vtkm::Float64 x1, const vtkm::Vec3f\_32 &hsv1, vtkm::Float64 x2, const vtkm::Vec3f\_32 &hsv2)  
 Add a line segment to the color function.  
 All points which lay between x1 and x2 (inclusive) are removed from the function.  
 Note: hsv1, and hsv2 values need to be between 0 and 1.0 (inclusive) Return the index of the point x1 (0 based), or -1 on error



bool **GetPoint**(vtkm::Int32 index, vtkm::Vec4f\_64&) const

Get the location, and rgb information for an existing point in the opacity function.

Note: components 1-3 are rgb and will have values between 0 and 1.0 (inclusive) Return the index of the point (0 based), or -1 on error.

vtkm::Int32 **UpdatePoint**(vtkm::Int32 index, const vtkm::Vec4f\_64&)

Update the location, and rgb information for an existing point in the color function.

If the location value for the index is modified the point is removed from the function and re-inserted in the proper sorted location.

Note: components 1-3 are rgb and must have values between 0 and 1.0 (inclusive). Return the new index of the updated point (0 based), or -1 on error.

bool **RemovePoint**(vtkm::Float64 x)

Remove the Color function point that exists at exactly x.

Return true if the point x exists and has been removed

bool **RemovePoint**(vtkm::Int32 index)

Remove the Color function point n.

Return true if n >= 0 && n < GetNumberOfPoints

vtkm::Int32 **GetNumberOfPoints**() const

Returns the number of points in the color function.

inline vtkm::Int32 **AddPointAlpha**(vtkm::Float64 x, vtkm::Float32 alpha)

Adds a point to the opacity function.

If the point already exists, it will be updated to the new value. Uses a midpoint of 0.5 (halfway between the control points) and sharpness of 0.0 (linear).

Note: alpha needs to be a value between 0 and 1.0 (inclusive). Return the index of the point (0 based), or -1 on error.

vtkm::Int32 **AddPointAlpha**(vtkm::Float64 x, vtkm::Float32 alpha, vtkm::Float32 midpoint, vtkm::Float32 sharpness)

Adds a point to the opacity function.

If the point already exists, it will be updated to the new value.

Note: alpha, midpoint, and sharpness values need to be between 0 and 1.0 (inclusive) Return the index of the point (0 based), or -1 on error.

inline vtkm::Int32 **AddSegmentAlpha**(vtkm::Float64 x1, vtkm::Float32 alpha1, vtkm::Float64 x2, vtkm::Float32 alpha2)

Add a line segment to the opacity function.

All points which lay between x1 and x2 (inclusive) are removed from the function. Uses a midpoint of 0.5 (halfway between the control points) and sharpness of 0.0 (linear).

Note: alpha values need to be between 0 and 1.0 (inclusive) Return the index of the point x1 (0 based), or -1 on error

vtkm::Int32 **AddSegmentAlpha**(vtkm::Float64 x1, vtkm::Float32 alpha1, vtkm::Float64 x2, vtkm::Float32 alpha2, const vtkm::Vec2f\_32 &mid\_sharp1, const vtkm::Vec2f\_32 &mid\_sharp2)

Add a line segment to the opacity function.

All points which lay between x1 and x2 (inclusive) are removed from the function.

Note: alpha, midpoint, and sharpness values need to be between 0 and 1.0 (inclusive) Return the index of the point x1 (0 based), or -1 on error

bool **GetPointAlpha**(vtkm::Int32 index, vtkm::Vec4f\_64&) const

Get the location, alpha, midpoint and sharpness information for an existing point in the opacity function.

Note: alpha, midpoint, and sharpness values all will be between 0 and 1.0 (inclusive) Return the index of the point (0 based), or -1 on error.

vtkm::Int32 **UpdatePointAlpha**(vtkm::Int32 index, const vtkm::Vec4f\_64&)

Update the location, alpha, midpoint and sharpness information for an existing point in the opacity function.

If the location value for the index is modified the point is removed from the function and re-inserted in the proper sorted location

Note: alpha, midpoint, and sharpness values need to be between 0 and 1.0 (inclusive) Return the new index of the updated point (0 based), or -1 on error.

bool **RemovePointAlpha**(vtkm::Float64 x)

Remove the Opacity function point that exists at exactly x.

Return true if the point x exists and has been removed

bool **RemovePointAlpha**(vtkm::Int32 index)

Remove the Opacity function point n.

Return true if  $n \geq 0$  &&  $n < \text{GetNumberOfPointsAlpha}$

vtkm::Int32 **GetNumberOfPointsAlpha**() const

Returns the number of points in the alpha function.

bool **FillColorTableFromDataPointer**(vtkm::Int32 n, const vtkm::Float64 \*ptr)

Fill the Color table from a *vtkm::Float64* pointer.

The *vtkm::Float64* pointer is required to have the layout out of [X1, R1, G1, B1, X2, R2, G2, B2, ..., Xn, Rn, Gn, Bn] where n is the number of nodes. This will remove any existing color control points.

Note: n represents the length of the array, so (  $n/4 ==$  number of control points )

Note: This is provided as a interoperability method with VTK Will return false and not modify anything if n is  $\leq 0$  or ptr == nullptr

bool **FillColorTableFromDataPointer**(vtkm::Int32 n, const vtkm::Float32 \*ptr)

Fill the Color table from a *vtkm::Float32* pointer.

The *vtkm::Float64* pointer is required to have the layout out of [X1, R1, G1, B1, X2, R2, G2, B2, ..., Xn, Rn, Gn, Bn] where n is the number of nodes. This will remove any existing color control points.

Note: n represents the length of the array, so (  $n/4 ==$  number of control points )

Note: This is provided as a interoperability method with VTK Will return false and not modify anything if n is  $\leq 0$  or ptr == nullptr

bool **FillOpacityTableFromDataPointer**(vtkm::Int32 n, const vtkm::Float64 \*ptr)

Fill the Opacity table from a *vtkm::Float64* pointer.

The *vtkm::Float64* pointer is required to have the layout out of [X1, A1, M1, S1, X2, A2, M2, S2, ..., Xn, An, Mn, Sn] where n is the number of nodes. The Xi values represent the value to map, the Ai values represent alpha (opacity) value, the Mi values represent midpoints, and the Si values represent sharpness. Use 0.5 for midpoint and 0.0 for sharpness to have linear interpolation of the alpha.

This will remove any existing opacity control points.

Note:  $n$  represents the length of the array, so ( $n/4 ==$  number of control points )

Will return false and not modify anything if  $n$  is  $\leq 0$  or  $ptr == nullptr$

bool **FillOpacityTableFromDataPointer**(*vtkm::Int32* n, const *vtkm::Float32* \*ptr)

Fill the Opacity table from a *vtkm::Float32* pointer.

The *vtkm::Float32* pointer is required to have the layout out of [X1, A1, M1, S1, X2, A2, M2, S2, ..., Xn, An, Mn, Sn] where  $n$  is the number of nodes. The  $X_i$  values represent the value to map, the  $A_i$  values represent alpha (opacity) value, the  $M_i$  values represent midpoints, and the  $S_i$  values represent sharpness. Use 0.5 for midpoint and 0.0 for sharpness to have linear interpolation of the alpha.

This will remove any existing opacity control points.

Note:  $n$  represents the length of the array, so ( $n/4 ==$  number of control points )

Will return false and not modify anything if  $n$  is  $\leq 0$  or  $ptr == nullptr$

bool **Sample**(*vtkm::Int32* numSamples, *vtkm::cont::ColorTableSamplesRGBA* &samples, *vtkm::Float64* tolerance = 0.002) const

generate RGB colors using regular spaced samples along the range.

Will use the current range of the color table to generate evenly spaced values using either *vtkm::Float32* or *vtkm::Float64* space. Will use *vtkm::Float32* space when the difference between the *vtkm::Float32* and *vtkm::Float64* values when the range is within *vtkm::Float32* space and the following are within a tolerance:

- $(\text{max-min}) / \text{numSamples}$
- $((\text{max-min}) / \text{numSamples}) * \text{numSamples}$

Note: This will return false if the number of samples is less than 2

bool **Sample**(*vtkm::Int32* numSamples, *vtkm::cont::ColorTableSamplesRGB* &samples, *vtkm::Float64* tolerance = 0.002) const

generate a sample lookup table using regular spaced samples along the range.

Will use the current range of the color table to generate evenly spaced values using either *vtkm::Float32* or *vtkm::Float64* space. Will use *vtkm::Float32* space when the difference between the *vtkm::Float32* and *vtkm::Float64* values when the range is within *vtkm::Float32* space and the following are within a tolerance:

- $(\text{max-min}) / \text{numSamples}$
- $((\text{max-min}) / \text{numSamples}) * \text{numSamples}$

Note: This will return false if the number of samples is less than 2

bool **Sample**(*vtkm::Int32* numSamples, *vtkm::cont::ArrayHandle*<*vtkm::Vec4ui\_8*> &colors, *vtkm::Float64* tolerance = 0.002) const

generate RGBA colors using regular spaced samples along the range.

Will use the current range of the color table to generate evenly spaced values using either *vtkm::Float32* or *vtkm::Float64* space. Will use *vtkm::Float32* space when the difference between the *vtkm::Float32* and *vtkm::Float64* values when the range is within *vtkm::Float32* space and the following are within a tolerance:

- $(\text{max-min}) / \text{numSamples}$
- $((\text{max-min}) / \text{numSamples}) * \text{numSamples}$

Note: This will return false if the number of samples is less than 2

bool **Sample**(vtkm::Int32 numSamples, vtkm::cont::ArrayHandle<vtkm::Vec3ui\_8> &colors, vtkm::Float64 tolerance = 0.002) const

generate RGB colors using regular spaced samples along the range.

Will use the current range of the color table to generate evenly spaced values using either *vtkm::Float32* or *vtkm::Float64* space. Will use *vtkm::Float32* space when the difference between the *vtkm::Float32* and *vtkm::Float64* values when the range is within *vtkm::Float32* space and the following are within a tolerance:

- $(\text{max} - \text{min}) / \text{numSamples}$
- $((\text{max} - \text{min}) / \text{numSamples}) * \text{numSamples}$

Note: This will return false if the number of samples is less than 2

vtkm::exec::ColorTable **PrepareForExecution**(vtkm::cont::DeviceAdapterId deviceId, vtkm::cont::Token &token) const

returns a virtual object pointer of the exec color table

This pointer is only valid as long as the *ColorTable* is unmodified

vtkm::Id **GetModifiedCount**() const

Returns the modified count for changes of the color table.

The *ModifiedCount* of the color table starts at 1 and gets incremented every time a change is made to the color table. The modified count allows consumers of a shared color table to keep track if the color table has been modified since the last time they used it. This is important for consumers that need to sample the color table. You only want to resample the color table if changes have been made.







## Public Static Functions












static std::set<std::string> **GetPresets**()

Returns the name of all preset color tables.

This list will include all presets defined in *vtkm::cont::ColorTable::Preset* and could include extras as well.

The easiest way to create a *vtkm::cont::ColorTable* is to provide the name of one of the many predefined sets of color provided by VTK-m. A list of all available predefined color tables is provided below.

-  Viridis Matplotlib Viridis, which is designed to have perceptual uniformity, accessibility to color blind viewers, and good conversion to black and white. This is the default color map.
-  Cool to Warm A color table designed to be perceptually even, to work well on shaded 3D surfaces, and to generally perform well across many uses.
-  Cool to Warm Extended This colormap is an expansion on cool to warm that moves through a wider range of hue and saturation. Useful if you are looking for a greater level of detail, but the darker colors at the end might interfere with 3D surfaces.
-  Inferno Matplotlib Inferno, which is designed to have perceptual uniformity, accessibility to color blind viewers, and good conversion to black and white.
-  Plasma Matplotlib Plasma, which is designed to have perceptual uniformity, accessibility to color blind viewers, and good conversion to black and white.
-  Black Body Radiation The colors are inspired by the wavelengths of light from black body radiation. The actual colors used are designed to be perceptually uniform.

-  X Ray Greyscale colormap useful for making volume renderings similar to what you would expect in an x-ray.
-  Green A sequential color map of green varied by saturation.
-  Black - Blue - White A sequential color map from black to blue to white.
-  Blue to Orange A double-ended (diverging) color table that goes from dark blues to a neutral white and then a dark orange at the other end.
-  Gray to Red A double-ended (diverging) color table with black/gray at the low end and orange/red at the high end.
-  Cold and Hot A double-ended color map with a black middle color and diverging values to either side. Colors go from red to yellow on the positive side and through blue on the negative side.
-  Blue - Green - Orange A three-part color map with blue at the low end, green in the middle, and orange at the high end.
-  Yellow - Gray - Blue A three-part color map with yellow at the low end, gray in the middle, and blue at the high end.
-  Rainbow Uniform A color table that spans the hues of a rainbow. This color table modifies the hues to make them more perceptually uniform than the raw color wavelengths.
-  Jet A rainbow color table that adds some darkness for greater perceptual resolution.
-  Rainbow Desaturated Basic rainbow colors with periodic dark points to increase the local discriminability.



## ERROR HANDLING

VTK-m contains several mechanisms for checking and reporting error conditions.

### 12.1 Runtime Error Exceptions

VTK-m uses exceptions to report errors. All exceptions thrown by VTK-m will be a subclass of `vtkm::cont::Error`. For simple error reporting, it is possible to simply catch a `vtkm::cont::Error` and report the error message string reported by the `vtkm::cont::Error::GetMessage()` method.

Example 1: Simple error reporting.

```
1 int main(int argc, char** argv)
2 {
3     try
4     {
5         // Do something cool with VTK-m
6         // ...
7     }
8     catch (const vtkm::cont::Error& error)
9     {
10        std::cout << error.GetMessage() << std::endl;
11        return 1;
12    }
13    return 0;
14 }
```

class **Error** : public std::exception

The superclass of all exceptions thrown by any VTKm function or method.

Subclassed by `vtkm::cont::ErrorBadAllocation`, `vtkm::cont::ErrorBadDevice`, `vtkm::cont::ErrorBadType`,  
`vtkm::cont::ErrorBadValue`, `vtkm::cont::ErrorExecution`, `vtkm::cont::ErrorFilterExecution`,  
`vtkm::cont::ErrorInternal`, `vtkm::cont::ErrorUserAbort`, `vtkm::cont::cuda::ErrorCuda`, `vtkm::io::ErrorIO`

## Public Functions

inline const std::string &**GetMessage**() const

Returns a message describing what caused the error.

inline const std::string &**GetStackTrace**() const

Provides a stack trace to the location where this error was thrown.

inline const char \***what**() const noexcept override

Returns the message for the error and the stack trace for it.

This method is provided for `std::exception` compatibility.

inline bool **GetIsDeviceIndependent**() const

Returns true if this exception is device independent.

For exceptions that are not device independent, `vtkm::TryExecute`, for example, may try executing the code on other available devices.

There are several subclasses to `vtkm::cont::Error`. The specific subclass gives an indication of the type of error that occurred when the exception was thrown. Catching one of these subclasses may help a program better recover from errors.

class **ErrorBadAllocation** : public vtkm::cont::Error

This class is thrown when VTK-m attempts to manipulate memory that it should not.

class **ErrorBadDevice** : public vtkm::cont::Error

This class is thrown when VTK-m performs an operation that is not supported on the current device.

class **ErrorBadType** : public vtkm::cont::Error

This class is thrown when VTK-m encounters data of a type that is incompatible with the current operation.

class **ErrorBadValue** : public vtkm::cont::Error

This class is thrown when a VTKm function or method encounters an invalid value that inhibits progress.

class **ErrorExecution** : public vtkm::cont::Error

This class is thrown in the control environment whenever an error occurs in the execution environment.

class **ErrorFilterExecution** : public vtkm::cont::Error

This class is primarily intended to filters to throw in the control environment to indicate an execution failure due to misconfiguration e.g.

incorrect parameters, etc. This is a device independent exception i.e. when thrown, unlike most other exceptions, VTK-m will not try to re-execute the filter on another available device.

class **ErrorInternal** : public vtkm::cont::Error

This class is thrown when VTKm detects an internal state that should never be reached.

This error usually indicates a bug in vtkm or, at best, VTKm failed to detect an invalid input it should have.

class **ErrorUserAbort** : public vtkm::cont::Error

This class is thrown when vtk-m detects a request for aborting execution in the current thread.



class **ErrorIO** : public vtkm::cont::Error

This class is thrown when VTK-m encounters an error with the file system.

This can happen if there is a problem with reading or writing a file such as a bad filename.

## 12.2 Asserting Conditions

In addition to the aforementioned error signaling, the `vtkm/Assert.h` header file defines a macro named `VTKM_ASSERT`. This macro behaves the same as the POSIX `assert` macro. It takes a single argument that is a condition that is expected to be true. If it is not true, the program is halted and a message is printed. Asserts are useful debugging tools to ensure that software is behaving and being used as expected.

**VTKM\_ASSERT**(condition)

Asserts that *condition* resolves to true.

If *condition* is false, then a diagnostic message is outputted and execution is terminated. The behavior is essentially the same as the POSIX `assert` macro, but is wrapped for added portability.

Like the POSIX `assert` macro, the check will be removed when compiling in non-debug mode (specifically when `NDEBUG` is defined), so be prepared for the possibility that the condition is never evaluated.

The `VTKM_NO_ASSERT` cmake and preprocessor option allows debugging builds to remove assertions for performance reasons.

Example 2: Using `VTKM_ASSERT`.

```

1 template<typename T>
2 VTkmCont T GetArrayValue(vtkm::cont::ArrayHandle<T> arrayHandle, vtkm::Id index)
3 {
4     VTkmAssert(index >= 0);
5     VTkmAssert(index < arrayHandle.GetNumberOfValues());

```

### Did You Know?

Like the POSIX `assert`, if the `NDEBUG` macro is defined, then `VTKM_ASSERT` will become an empty expression. Typically `NDEBUG` is defined with a compiler flag (like `-DNDEBUG`) for release builds to better optimize the code. CMake will automatically add this flag for release builds.

### Common Errors

A helpful warning provided by many compilers alerts you of unused variables. (This warning is commonly enabled on VTK-m regression test nightly builds.) If a function argument is used only in a `VTKM_ASSERT`, then it will be required for debug builds and be unused in release builds. To get around this problem, add a statement to the function of the form `(void)variableName;`. This statement will have no effect on the code generated but will suppress the warning for release builds.

## 12.3 Compile Time Checks

Because VTK-m makes heavy use of C++ templates, it is possible that these templates could be used with inappropriate types in the arguments. Using an unexpected type in a template can lead to very confusing errors, so it is better to catch such problems as early as possible. The `VTM_STATIC_ASSERT` macro, defined in `vtkm/StaticAssert.h` makes this possible. This macro takes a constant expression that can be evaluated at compile time and verifies that the result is true.

In the following example, `VTM_STATIC_ASSERT` and its sister macro `VTM_STATIC_ASSERT_MSG`, which allows you to give a descriptive message for the failure, are used to implement checks on a templated function that is designed to work on any scalar type that is represented by 32 or more bits.

Example 3: Using `VTM_STATIC_ASSERT`.

```
1 template<typename T>
2 VTKM_EXEC_CONT void MyMathFunction(T& value)
3 {
4     VTKM_STATIC_ASSERT((std::is_same<typename vtkm::TypeTraits<T>::DimensionalityTag,
5                             vtkm::TypeTraitsScalarTag>::value));
6
7     VTKM_STATIC_ASSERT_MSG(sizeof(T) >= 4,
8                             "MyMathFunction needs types with at least 32 bits.");
```

---

### Did You Know?

In addition to the several trait template classes provided by VTK-m to introspect C++ types, the C++ standard `type_traits` header file contains several helpful templates for general queries on types. [Example 3](#) demonstrates the use of one such template: `std::is_same`.

---

---

### Common Errors

Many templates used to introspect types resolve to the tags `std::true_type` and `std::false_type` rather than the constant values `true` and `false` that `VTM_STATIC_ASSERT` expects. The `std::true_type` and `std::false_type` tags can be converted to the Boolean literal by adding `::value` to the end of them. Failing to do so will cause `VTM_STATIC_ASSERT` to behave incorrectly. [Example 3](#) demonstrates getting the Boolean literal from the result of `std::is_same`.

---

## MANAGING DEVICES

Multiple vendors vie to provide accelerator-type processors. VTK-m endeavors to support as many such architectures as possible. Each device and device technology requires some level of code specialization, and that specialization is encapsulated in a unit called a device adapter.

So far in [Part II \(Using VTK-m\)](#) we have been writing code that runs on a local serial CPU. In those examples where we run a filter, VTK-m is launching parallel execution in the execution environment. Internally VTK-m uses a device adapter to manage this execution.

A build of VTK-m generally supports multiple device adapters. In this chapter we describe how to represent and manage devices.

### 13.1 Device Adapter Tag

A device adapter is identified by a *device adapter tag*. This tag, which is simply an empty struct type, is used as the template parameter for several classes in the VTK-m control environment and causes these classes to direct their work to a particular device. The following device adapter tags are available in VTK-m.

```
struct DeviceAdapterTagSerial : public vtkm::cont::DeviceAdapterId
```

Tag for a device adapter that performs all computation on the same single thread as the control environment.

This device is useful for debugging. This device is always available. This tag is defined in `vtkm/cont/DeviceAdapterSerial.h`.

```
struct DeviceAdapterTagCuda : public vtkm::cont::DeviceAdapterId
```

Tag for a device adapter that uses a CUDA capable GPU device.

For this device to work, VTK-m must be configured to use CUDA and the code must be compiled by the CUDA nvcc compiler. This tag is defined in `vtkm/cont/cuda/DeviceAdapterCuda.h`.

```
struct DeviceAdapterTagOpenMP : public vtkm::cont::DeviceAdapterId
```

Tag for a device adapter that uses OpenMP compiler extensions to run algorithms on multiple threads.

For this device to work, VTK-m must be configured to use OpenMP and the code must be compiled with a compiler that supports OpenMP pragmas. This tag is defined in `vtkm/cont/openmp/DeviceAdapterOpenMP.h`.

```
struct DeviceAdapterTagTBB : public vtkm::cont::DeviceAdapterId
```

Tag for a device adapter that uses the Intel Threading Building Blocks library to run algorithms on multiple threads.

For this device to work, VTK-m must be configured to use TBB and the executable must be linked to the TBB library. This tag is defined in `vtkm/cont/tbb/DeviceAdapterTBB.h`.

struct **DeviceAdapterTagKokkos** : public `vtkm::cont::DeviceAdapterId`

Tag for a device adapter that uses the Kokkos library to run algorithms in parallel.

For this device to work, VTK-m must be configured to use Kokkos and the executable must be linked to the Kokkos libraries. VTK-m will use the default execution space of the provided kokkos library build. This tag is defined in `vtkm/cont/kokkos/DeviceAdapterKokkos.h`.

The following example uses the tag for the Kokkos device adapter to specify a specific device for VTK-m to use. (Details on specifying devices in VTK-m is provided in [Section 13.4 \(Specifying Devices\)](#).)

Example 1: Specifying a device using a device adapter tag.

```
1 vtkm::cont::ScopedRuntimeDeviceTracker(vtkm::cont::DeviceAdapterTagKokkos{});
```

For classes and methods that have a template argument that is expected to be a device adapter tag, the tag type can be checked with the `VTKM_IS_DEVICE_ADAPTER_TAG` macro to verify the type is a valid device adapter tag. It is good practice to check unknown types with this macro to prevent further unexpected errors.

## 13.2 Device Adapter Id

Using a device adapter tag directly means that the type of device needs to be known at compile time. To store a device adapter type at run time, one can instead use `vtkm::cont::DeviceAdapterId`. `vtkm::cont::DeviceAdapterId` is a superclass to all the device adapter tags, and any device adapter tag can be “stored” in a `vtkm::cont::DeviceAdapterId`. Thus, it is more common for functions and classes to use `vtkm::cont::DeviceAdapterId` then to try to track a specific device with templated code.

struct **DeviceAdapterId**

An object used to specify a device.

`vtkm::cont::DeviceAdapterId` can be used to specify a device to use when executing some code. Each `DeviceAdapterTag` object inherits from `vtkm::cont::DeviceAdapterId`. Functions can accept a `vtkm::cont::DeviceAdapterId` object rather than a templated tag to select a device adapter at runtime.

Subclassed by `vtkm::cont::DeviceAdapterTagAny`, `vtkm::cont::DeviceAdapterTagCuda`,  
`vtkm::cont::DeviceAdapterTagKokkos`, `vtkm::cont::DeviceAdapterTagOpenMP`,  
`vtkm::cont::DeviceAdapterTagSerial`, `vtkm::cont::DeviceAdapterTagTBB`, `vtkm::cont::DeviceAdapterTagUndefined`

### Public Functions

inline constexpr bool **IsValid()** const

Return whether this object represents a valid type of device.

This method will return true if the id represents a specific, valid device. It will return true even if the device is disabled in by the runtime tracker or if the device is not supported by the VTK-m build configuration.

It should be noted that this method return false for tags that are not specific devices. This includes `vtkm::cont::DeviceAdapterTagAny` and `vtkm::cont::DeviceAdapterTagUndefined`.

inline constexpr `vtkm::Int8` **GetValue()** const

Returns the numeric value of the index.

DeviceAdapterNameType **GetName()** const

Return a name representing the device.

The string returned from this method is stored in a type named `vtkm::cont::DeviceAdapterNameType`, which is currently aliased to `std::string`. The device adapter name is useful for printing information about a device being used.

---

### Did You Know?

As a cheat, all device adapter tags actually inherit from the `vtkm::cont::DeviceAdapterId` class. Thus, all of these methods can be called directly on a device adapter tag.

---

---

### Common Errors

Just because the `vtkm::cont::DeviceAdapterId::IsValueValid()` returns true that does not necessarily mean that this device is available to be run on. It simply means that the device is implemented in VTK-m. However, that device might not be compiled, or that device might not be available on the current running system, or that device might not be enabled. Use the device runtime tracker described in [Section 13.3 \(Runtime Device Tracker\)](#) to determine if a particular device can actually be used.

---

In addition to the provided device adapter tags listed previously, a `vtkm::cont::DeviceAdapterId` can store some special device adapter tags that do not directly specify a specific device.

```
struct DeviceAdapterTagAny : public vtkm::cont::DeviceAdapterId
```

Tag for a device adapter used to specify that any device may be used for an operation.

In practice this is limited to devices that are currently available.

```
struct DeviceAdapterTagUndefined : public vtkm::cont::DeviceAdapterId
```

Tag for a device adapter used to avoid specifying a device.

Useful as a placeholder when a device can be specified but none is given.

---

### Did You Know?

Any device adapter tag can be used where a device adapter id is expected. Thus, you can use a device adapter tag whenever you want to specify a particular device and pass that to any method expecting a device id. Likewise, it is usually more convenient for classes and methods to manage device adapter ids rather than device adapter tag.

---

## 13.3 Runtime Device Tracker

It is often the case that you are agnostic about what device VTK-m algorithms run so long as they complete correctly and as fast as possible. Thus, rather than directly specify a device adapter, you would like VTK-m to try using the best available device, and if that does not work try a different device. Because of this, there are many features in VTK-m that behave this way. For example, you may have noticed that running filters, as in the examples of [Chapter 9 \(Running Filters\)](#), you do not need to specify a device; they choose a device for you.

However, even though we often would like VTK-m to choose a device for us, we still need a way to manage device preferences. VTK-m also needs a mechanism to record runtime information about what devices are available so that it does not have to continually try (and fail) to use devices that are not available at runtime. These needs are met

with the `vtkm::cont::RuntimeDeviceTracker` class. `vtkm::cont::RuntimeDeviceTracker` maintains information about which devices can and should be run on. VTK-m maintains a `vtkm::cont::RuntimeDeviceTracker` for each thread your code is operating on. To get the runtime device for the current thread, use the `vtkm::cont::GetRuntimeDeviceTracker()` method.

```
vtkm::cont::RuntimeDeviceTracker &vtkm::cont::GetRuntimeDeviceTracker()
```

Get the `RuntimeDeviceTracker` for the current thread.

Many features in VTK-m will attempt to run algorithms on the “best available device.” This often is determined at runtime as failures in one device are recorded and that device is disabled. To prevent having to check over and over again, VTK-m uses per thread runtime device tracker so that these choices are marked and shared.

class **RuntimeDeviceTracker**

`RuntimeDeviceTracker` is the central location for determining which device adapter will be active for algorithm execution.

Many features in VTK-m will attempt to run algorithms on the “best available device.” This generally is determined at runtime as some backends require specific hardware, or failures in one device are recorded and that device is disabled.

While `vtkm::cont::RuntimeDeviceInformation` reports on the existence of a device being supported, this tracks on a per-thread basis when worklets fail, why the fail, and will update the list of valid runtime devices based on that information.

Subclassed by `vtkm::cont::ScopedRuntimeDeviceTracker`

## Public Functions

```
bool CanRunOn(DeviceAdapterId deviceId) const
```

Returns true if the given device adapter is supported on the current machine.

```
inline void ReportAllocationFailure(vtkm::cont::DeviceAdapterId deviceId, const  
                                   vtkm::cont::ErrorBadAllocation&)
```

Report a failure to allocate memory on a device, this will flag the device as being unusable for all future invocations.

```
inline void ReportBadDeviceFailure(vtkm::cont::DeviceAdapterId deviceId, const  
                                   vtkm::cont::ErrorBadDevice&)
```

Report a `ErrorBadDevice` failure and flag the device as unusable.

```
void ResetDevice(vtkm::cont::DeviceAdapterId deviceId)
```

Reset the tracker for the given device.

This will discard any updates caused by reported failures. Passing `DeviceAdapterTagAny` to this will reset all devices (same as `Reset()`).

```
void Reset()
```

Reset the tracker to its default state for default devices.

Will discard any updates caused by reported failures.

void **DisableDevice**(*DeviceAdapterId* deviceId)

Disable the given device.

The main intention of *RuntimeDeviceTracker* is to keep track of what devices are working for VTK-m. However, it can also be used to turn devices on and off. Use this method to disable (turn off) a given device. Use *ResetDevice()* to turn the device back on (if it is supported).

Passing *DeviceAdapterTagAny* to this will disable all devices.

void **ForceDevice**(*DeviceAdapterId* deviceId)

Disable all devices except the specified one.

The main intention of *RuntimeDeviceTracker* is to keep track of what devices are working for VTK-m. However, it can also be used to turn devices on and off. Use this method to disable all devices except one to effectively force VTK-m to use that device. Either pass the *DeviceAdapterTagAny* to this function or call *Reset()* to restore all devices to their default state.

This method will throw a *vtkm::cont::ErrorBadValue* if the given device does not exist on the system.

bool **GetThreadFriendlyMemAlloc**() const

Get/Set use of thread-friendly memory allocation for a device.

void **SetThreadFriendlyMemAlloc**(bool state)

Get/Set use of thread-friendly memory allocation for a device.

void **CopyStateFrom**(const vtkm::cont::RuntimeDeviceTracker &tracker)

Copies the state from the given device.

This is a convenient way to allow the *RuntimeDeviceTracker* on one thread copy the behavior from another thread.

void **SetAbortChecker**(const std::function<bool()> &func)

Set/Clear the abort checker functor.

If set the abort checker functor is called by *vtkm::cont::TryExecute()* before scheduling a task on a device from the associated the thread. If the functor returns *true*, an exception is thrown.

void **ClearAbortChecker**()

Set/Clear the abort checker functor.

If set the abort checker functor is called by *vtkm::cont::TryExecute()* before scheduling a task on a device from the associated the thread. If the functor returns *true*, an exception is thrown.

void **PrintSummary**(std::ostream &out) const

Produce a human-readable report on the state of the runtime device tracker.

## 13.4 Specifying Devices

A *vtkm::cont::RuntimeDeviceTracker* can be used to specify which devices to consider for a particular operation. However, a better way to specify devices is to use the *vtkm::cont::ScopedRuntimeDeviceTracker* class. When a *vtkm::cont::ScopedRuntimeDeviceTracker* is constructed, it specifies a new set of devices for VTK-m to use. When the *vtkm::cont::ScopedRuntimeDeviceTracker* is destroyed as it leaves scope, it restores VTK-m's devices to those that existed when it was created.

```
class ScopedRuntimeDeviceTracker : public vtkm::cont::RuntimeDeviceTracker
```

A class to create a scoped runtime device tracker object.

This object captures the state of the per-thread device tracker and will revert any changes applied during its lifetime on destruction.

## Unnamed Group

**ScopedRuntimeDeviceTracker**(const vtkm::cont::RuntimeDeviceTracker &tracker =  
GetRuntimeDeviceTracker())

Construct a *ScopedRuntimeDeviceTracker* associated with the thread, associated with the provided tracker (defaults to current thread's tracker).

Any modifications to the *ScopedRuntimeDeviceTracker* will effect what ever thread the *tracker* is associated with, which might not be the thread on which the *ScopedRuntimeDeviceTracker* was constructed.

Constructors are not thread safe

**ScopedRuntimeDeviceTracker**(vtkm::cont::DeviceAdapterId device, RuntimeDeviceTrackerMode mode =  
RuntimeDeviceTrackerMode::Force, const  
vtkm::cont::RuntimeDeviceTracker &tracker =  
GetRuntimeDeviceTracker())

Use this constructor to modify the state of the device adapters associated with the provided tracker.

Use mode with device as follows:

'Force' (default)

- Force-Enable the provided single device adapter
- Force-Enable all device adapters when using vtkm::cont::DeviceAdaterTagAny 'Enable'
- Enable the provided single device adapter if it was previously disabled
- Enable all device adapters that are currently disabled when using vtkm::cont::DeviceAdaterTagAny 'Disable'
- Disable the provided single device adapter
- Disable all device adapters when using vtkm::cont::DeviceAdaterTagAny

**ScopedRuntimeDeviceTracker**(const std::function<bool()> &abortChecker, const  
vtkm::cont::RuntimeDeviceTracker &tracker =  
GetRuntimeDeviceTracker())

Use this constructor to set the abort checker functor for the provided tracker.

**~ScopedRuntimeDeviceTracker()**

Destructor is not thread safe.

The following example demonstrates how the *vtkm::cont::ScopedRuntimeDeviceTracker* is used to force the VTK-m operations that happen within a function to operate exclusively with the Kokkos device.

Example 2: Restricting which devices VTK-m uses per thread.

```
1 void ChangeDefaultRuntime()
2 {
3     std::cout << "Checking changing default runtime." << std::endl;
4
5     vtkm::cont::ScopedRuntimeDeviceTracker(vtkm::cont::DeviceAdapterTagKokkos{});
6
7     // VTK-m operations limited to Kokkos devices here...
8 }
```

(continues on next page)



(continued from previous page)

```

9  // Devices restored as we leave scope.
10 }
```

In the previous example we forced VTK-m to use the Kokkos device. This is the default behavior of `vtkm::cont::ScopedRuntimeDeviceTracker`, but the constructor takes an optional second argument that is a value in the `vtkm::cont::RuntimeDeviceTrackerMode` to specify how modify the current device adapter list.

enum class `vtkm::cont::RuntimeDeviceTrackerMode`

Identifier used to specify whether to enable or disable a particular device.

*Values:*

enumerator **Force**

Replaces the current list of devices to try with the device specified.

This has the effect of forcing VTK-m to use the provided device. This is the default behavior for `vtkm::cont::ScopedRuntimeDeviceTracker`.

enumerator **Enable**

Adds the provided device adapter to the list of devices to try.

enumerator **Disable**

Removes the provided device adapter from the list of devices to try.

As a motivating example, let us say that we want to perform a deep copy of an array (described in [Section 17.2 \(Deep Array Copies\)](#)). However, we do not want to do the copy on a Kokkos device because we happen to know the data is not on that device and we do not want to spend the time to transfer the data to that device. We can use a `vtkm::cont::ScopedRuntimeDeviceTracker` to temporarily disable the Kokkos device for this operation.

Example 3: Disabling a device with  
`vtkm::cont::RuntimeDeviceTracker`.

```

1  vtkm::cont::ScopedRuntimeDeviceTracker tracker(
2      vtkm::cont::DeviceAdapterTagKokkos(), vtkm::cont::RuntimeDeviceTrackerMode::Disable);
3
4  vtkm::cont::ArrayCopy(srcArray, destArray);
```



## TIMERS

It is often the case that you need to measure the time it takes for an operation to happen. This could be for performing measurements for algorithm study or it could be to dynamically adjust scheduling.

Performing timing in a multi-threaded environment can be tricky because operations happen asynchronously. To ensure that accurate timings can be made, VTK-m provides a `vtkm::cont::Timer` class to provide an accurate measurement of operations that happen on devices that VTK-m can use. By default, `vtkm::cont::Timer` will time operations on all possible devices.

The timer is started by calling the `vtkm::cont::Timer::Start()` method. The timer can subsequently be stopped by calling `vtkm::cont::Timer::Stop()`. The time elapsed between calls to `vtkm::cont::Timer::Start()` and `vtkm::cont::Timer::Stop()` (or the current time if `vtkm::cont::Timer::Stop()` was not called) can be retrieved with a call to the `vtkm::cont::Timer::GetElapsedTime()` method. Subsequently calling `vtkm::cont::Timer::Start()` again will restart the timer.

Example 1: Using `vtkm::cont::Timer`.

```
1  vtkm::filter::field_transform::PointElevation elevationFilter;
2  elevationFilter.SetUseCoordinateSystemAsField(true);
3  elevationFilter.SetOutputFieldName("elevation");
4
5  vtkm::cont::Timer timer;
6
7  timer.Start();
8
9  vtkm::cont::DataSet result = elevationFilter.Execute(dataSet);
10
11  // This code makes sure data is pulled back to the host in a host/device
12  // architecture.
13  vtkm::cont::ArrayHandle<vtkm::Float64> outArray;
14  result.GetField("elevation").GetData().AsArrayHandle(outArray);
15  outArray.SyncControlArray();
16
17  timer.Stop();
18
19  vtkm::Float64 elapsedTime = timer.GetElapsedTime();
20
21  std::cout << "Time to run: " << elapsedTime << std::endl;
```

---

### Common Errors

Some device require data to be copied between the host CPU and the device. In this case you might want to measure

the time to copy data back to the host. This can be done by “touching” the data on the host by getting a control portal.

---

The VTK-m `vtkm::cont::Timer` does its best to capture the time it takes for all parallel operations run between calls to `vtkm::cont::Timer::Start()` and `vtkm::cont::Timer::Stop()` to complete. It does so by synchronizing to concurrent execution on devices that might be in use.

---

### Common Errors

Because `vtkm::cont::Timer` synchronizes with devices (essentially waiting for the device to finish executing), that can have an effect on how your program runs. Be aware that using a `vtkm::cont::Timer` can itself change the performance of your code. In particular, starting and stopping the timer many times to measure the parts of a sequence of operations can potentially make the whole operation run slower.

---

By default, `vtkm::cont::Timer` will synchronize with all active devices. However, if you want to measure the time for a specific device, then you can pass the device adapter tag or id to `vtkm::cont::Timer`'s constructor. You can also change the device being used by passing a device adapter tag or id to the `vtkm::cont::Timer::Reset()` method. A device can also be specified through an optional argument to the `vtkm::cont::Timer::GetElapsedTime()` method.

---

### class **Timer**

A class that can be used to time operations in VTK-m that might be occurring in parallel.

Users are recommended to provide a device adapter at construction time which matches the one being used to execute algorithms to ensure that thread synchronization is correct and accurate. If no device adapter is provided at construction time, the maximum elapsed time of all enabled devices will be returned. Normally cuda is expected to have the longest execution time if enabled. Per device adapter time query is also supported. It's useful when users want to reuse the same timer to measure the cuda kernel call as well as the cuda device execution. It is also possible to change the device adapter after construction by calling the form of the Reset method with a new `DeviceAdapterId`.

There is no guaranteed resolution of the time but should generally be good to about a millisecond.

### Public Functions

#### void **Reset()**

Restores the initial state of the `class: vtkm::cont::Timer`.

All previous recorded time is erased. `Reset()` optionally takes a device adapter tag or id that specifies on which device to time and synchronize.

#### void **Reset**(`vtkm::cont::DeviceAdapterId` device)

Resets the timer and changes the device to time on.

#### void **Start()**

Causes the `Timer` to begin timing.

The elapsed time will record an interval beginning when this method is called.

#### void **Stop()**

Causes the `Timer()` to finish timing.

The elapsed time will record an interval ending when this method is called. It is invalid to stop the timer if `Started()` is not true.

bool **Started()** const

Returns true if [Start\(\)](#) has been called.

It is invalid to try to get the elapsed time if [Started\(\)](#) is not true.

bool **Stopped()** const

Returns true if [Timer::Stop\(\)](#) has been called.

If [Stopped\(\)](#) is true, then the elapsed time will no longer increase. If [Stopped\(\)](#) is false and [Started\(\)](#) is true, then the timer is still running.

bool **Ready()** const

Used to check if [Timer](#) has finished the synchronization to get the result from the device.

vtkm::Float64 **GetElapsedTime()** const

Returns the amount of time that has elapsed between calling [Start\(\)](#) and [Stop\(\)](#).

If [Stop\(\)](#) was not called, then the amount of time between calling [Start\(\)](#) and [GetElapsedTime\(\)](#) is returned. [GetElapsedTime\(\)](#) can optionally take a device adapter tag or id to specify for which device to return the elapsed time. Returns the device for which this timer is synchronized. If the device adapter has the same id as [vtkm::cont::DeviceAdapterTagAny](#), then the timer will synchronize all devices.

inline vtkm::cont::DeviceAdapterId **GetDevice()** const

Returns the id of the device adapter for which this timer is synchronized.

If the device adapter has the same id as [vtkm::cont::DeviceAdapterTagAny](#) (the default), then the timer will synchronize on all devices.

void **Synchronize()** const

Synchronize the device(s) that this timer is monitoring without starting or stopping the timer.

This is useful for ensuring that external events are synchronized to this timer.

Note that this method will always block until the device(s) finish even if the [Start/Stop](#) methods do not actually block. For example, the timer for CUDA does not actually wait for asynchronous operations to finish. Rather, it inserts a fence and records the time as fences are encountered. But regardless, this [Synchronize](#) method will block for the CUDA device.



## IMPLICIT FUNCTIONS

VTK-m's implicit functions are objects that are constructed with values representing 3D spatial coordinates that often describe a shape. Each implicit function is typically defined by the surface formed where the value of the function is equal to 0. All implicit functions implement `Value()` and `Gradient()` methods that describe the orientation of a provided point with respect to the implicit function's shape.

The `Value()` method for an implicit function takes a `vtkm::Vec3f` and returns a `vtkm::FloatDefault` representing the orientation of the point with respect to the implicit function's shape. Negative scalar values represent vector points inside of the implicit function's shape. Positive scalar values represent vector points outside the implicit function's shape. Zero values represent vector points that lie on the surface of the implicit function.

The `Gradient()` method for an implicit function takes a `vtkm::Vec3f` and returns a `vtkm::Vec3f` representing the pointing direction from the implicit function's shape. Gradient calculations are more object shape specific. It is advised to look at the individual shape implementations for specific implicit functions.

Implicit functions are useful when trying to clip regions from a dataset. For example, it is possible to use `vtkm::filter::contour::ClipWithImplicitFunction` to remove a region in a provided dataset according to the shape of an implicit function. See [Section 10.3.4 \(Clip with Implicit Function\)](#) for more information on clipping with implicit functions.

VTK-m has implementations of various implicit functions provided by the following subclasses.

### 15.1 Plane

`vtkm::Plane` defines an infinite plane. The plane is defined by a pair of `vtkm::Vec3f` values that represent the origin, which is any point on the plane, and a normal, which is a vector that is tangent to the plane. These are set with the `vtkm::Plane::SetOrigin()` and `vtkm::Plane::SetNormal()` methods, respectively. Planes extend infinitely from the origin point in the direction perpendicular from the Normal. An example `vtkm::Plane` is shown in [Figure 1](#).

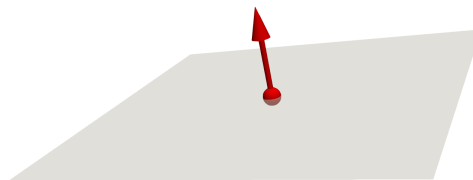


Figure 1: Visual Representation of an Implicit Plane. The red dot and arrow represent the origin and normal of the plane, respectively. For demonstrative purposes the plane as shown with limited area, but in actuality the plane extends infinitely.

```
template<typename CoordType = vtkm::FloatDefault>
```

class **Plane** : public vtkm::internal::ImplicitFunctionBase<*Plane*>

Represent a plane with a base point (origin) and normal vector.

Implicit function for a plane.

A plane is defined by a point in the plane and a normal to the plane. The normal does not have to be a unit vector. The implicit function will still evaluate to 0 at the plane, but the values outside the plane (and the gradient) will be scaled by the length of the normal vector.

## Public Functions

**Plane**()

Construct a default plane whose base point is the origin and whose normal is (0,0,1)

**Plane**(const Vector &origin, const Vector &normal, *CoordType* tol2 = static\_cast<*CoordType*>(1e-8f))

Construct a plane with the given *origin* and *normal*.

inline bool **IsValid**() const

Return true if the plane's normal is well-defined to within the given tolerance.

*CoordType* **DistanceTo**(const Vector &point) const

Return the **signed** distance from the plane to the point.

Vector **ClosestPoint**(const Vector &point) const

Return the closest point in the plane to the given point.

template<bool **IsTwoSided**>

bool **Intersect**(const Ray<*CoordType*, 3, *IsTwoSided*> &ray, *CoordType* &parameter, Vector &point, bool &lineInPlane, *CoordType* tol = *CoordType*(1e-6f)) const

Intersect this plane with the ray (or line if the ray is two-sided).

Returns true if there is a non-degenerate intersection (i.e., an isolated point of intersection). Returns false if there is no intersection *or* if the intersection is degenerate (i.e., the entire ray/line lies in the plane). In the latter case, *lineInPlane* will be true upon exit.

If this method returns true, then *parameter* will be set to a number indicating where along the ray/line the plane hits and *point* will be set to that location. If the input is a ray, the *parameter* will be non-negative.

bool **Intersect**(const LineSegment<*CoordType*> &segment, *CoordType* &parameter, bool &lineInPlane) const

Intersect this plane with the line *segment*.

Returns true if there is a non-degenerate intersection (i.e., an isolated point of intersection). Returns false if there is no intersection *or* if the intersection is degenerate (i.e., the entire line segment lies in the plane). In the latter case, *lineInPlane* will be true upon exit.

If this method returns true, then *parameter* will be set to a number in [0,1] indicating where along the line segment the plane hits.

bool **Intersect**(const LineSegment<*CoordType*> &segment, *CoordType* &parameter, Vector &point, bool &lineInPlane) const

Intersect this plane with the line *segment*.

Returns true if there is a non-degenerate intersection (i.e., an isolated point of intersection). Returns false if there is no intersection *or* if the intersection is degenerate (i.e., the entire line segment lies in the plane). In the latter case, *lineInPlane* will be true upon exit.



If this method returns true, then *parameter* will be set to a number in [0,1] indicating where along the line segment the plane hits and *point* will be set to that location.

```
bool Intersect(const Plane<CoordType> &other, Ray<CoordType, 3, true> &ray, bool &coincident,
               CoordType tol2 = static_cast<CoordType>(1e-6f)) const
```

Intersect this plane with another plane.

Returns true if there is a non-degenerate intersection (i.e., a line of intersection). Returns false if there is no intersection *or* if the intersection is degenerate (i.e., the planes are coincident). In the latter case, *coincident* will be true upon exit and *segment* will be unmodified.

If this method returns true, then the resulting *segment* will have its base point on the line of intersection and its second point will be a unit length away in the direction of the cross product of the input plane normals (this plane crossed with the *other*).

The tolerance *tol* is the minimum squared length of the cross-product of the two plane normals. It is also compared to the squared distance of the base point of *other* away from *this* plane when considering whether the planes are coincident.

```
inline explicit Plane(const Vector &normal = {0, 0, 1})
```

Construct a plane through the origin with the given normal.

```
inline Plane(const Vector &origin, const Vector &normal)
```

Construct a plane through the given point with the given normal.

```
inline void SetOrigin(const Vector &origin)
```

Specify the origin of the plane.

The origin can be any point on the plane.

```
inline void SetNormal(const Vector &normal)
```

Specify the normal vector to the plane.

The magnitude of the plane does not matter (so long as it is more than zero) in terms of the location of the plane where the implicit function equals 0. However, if offsets away from the plane matter then the magnitude determines the scale of the value away from the plane.

```
inline const Vector &GetOrigin() const
```

Specify the origin of the plane.

The origin can be any point on the plane.

```
inline const Vector &GetNormal() const
```

Specify the normal vector to the plane.

The magnitude of the plane does not matter (so long as it is more than zero) in terms of the location of the plane where the implicit function equals 0. However, if offsets away from the plane matter then the magnitude determines the scale of the value away from the plane.

```
inline Scalar Value(const Vector &point) const
```

Evaluate the value of the implicit function.

The *Value()* method for an implicit function takes a *vtkm::Vec3f* and returns a *vtkm::FloatDefault* representing the orientation of the point with respect to the implicit function's shape. Negative scalar values represent vector points inside of the implicit function's shape. Positive scalar values represent vector points outside the implicit function's shape. Zero values represent vector points that lie on the surface of the implicit function.

```
inline Vector Gradient(const Vector&) const
```

Evaluate the gradient of the implicit function.

The `Gradient()` method for an implicit function takes a `vtkm::Vec3f` and returns a `vtkm::Vec3f` representing the pointing direction from the implicit function's shape. Gradient calculations are more object shape specific. It is advised to look at the individual shape implementations for specific implicit functions.

## 15.2 Sphere

`vtkm::Sphere` defines a sphere. The `vtkm::Sphere` is defined by a center location and a radius, which are set with the `vtkm::Sphere::SetCenter()` and `vtkm::Sphere::SetRadius()` methods, respectively. An example `vtkm::Sphere` is shown in Figure 2.

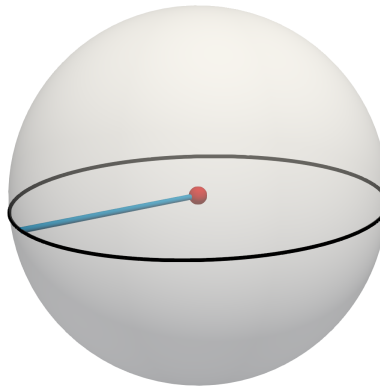


Figure 2: Visual Representation of an Implicit Sphere. The red dot represents the center of the sphere. The radius is the length of any line (like the blue one shown here) that extends from the center in any direction to the surface.

```
template<typename CoordType = vtkm::FloatDefault, int Dim = 3>
```

```
class Sphere : public vtkm::internal::ImplicitFunctionBase<Sphere>
```

Represent a sphere of the given *Dimension*.

Implicit function for a sphere.

If a constructor is given an invalid specification, then the Radius of the resulting sphere will be -1.

A sphere is defined by its center and a radius.

The value of the sphere implicit function is the square of the distance from the center biased by the radius (so the surface of the sphere is at value 0).

### Public Functions

#### **Sphere()**

Construct a default sphere (unit radius at the origin).

#### **Sphere**(const Vector &center, *CoordType* radius)

Construct a sphere from a center point and radius.

```
inline bool IsValid() const
```

Return true if the sphere is valid (i.e., has a strictly positive radius).

bool **Contains**(const Vector &point, *CoordType* tol2 = 0.f) const  
 Return whether the point lies strictly inside the sphere.

int **Classify**(const Vector &point, *CoordType* tol2 = 0.f) const  
 Classify a point as inside (-1), on (0), or outside (+1) of the sphere.  
 The tolerance *tol2* is the maximum allowable difference in squared magnitude between the squared radius and the squared distance between the *point* and Center.

inline explicit **Sphere**(Scalar radius = 0.5)  
 Construct a sphere with center at (0,0,0) and the given radius.

inline **Sphere**(Vector center, Scalar radius)  
 Construct a sphere with the given center and radius.

inline void **SetRadius**(Scalar radius)  
 Specify the radius of the sphere.

inline void **SetCenter**(const Vector &center)  
 Specify the center of the sphere.

inline Scalar **GetRadius**() const  
 Specify the radius of the sphere.

inline const Vector &**GetCenter**() const  
 Specify the center of the sphere.

inline Scalar **Value**(const Vector &point) const  
 Evaluate the value of the implicit function.  
 The *Value()* method for an implicit function takes a *vtkm::Vec3f* and returns a *vtkm::FloatDefault* representing the orientation of the point with respect to the implicit function's shape. Negative scalar values represent vector points inside of the implicit function's shape. Positive scalar values represent vector points outside the implicit function's shape. Zero values represent vector points that lie on the surface of the implicit function.

inline Vector **Gradient**(const Vector &point) const  
 Evaluate the gradient of the implicit function.  
 The *Gradient()* method for an implicit function takes a *vtkm::Vec3f* and returns a *vtkm::Vec3f* representing the pointing direction from the implicit function's shape. Gradient calculations are more object shape specific. It is advised to look at the individual shape implementations for specific implicit functions.

## 15.3 Cylinder

*vtkm::Cylinder* defines a cylinder that extends infinitely along its axis. The cylinder is defined with a center point, a direction of the center axis, and a radius, which are set with *vtkm::Cylinder::SetCenter()*, *vtkm::Cylinder::SetAxis()*, and *vtkm::Cylinder::SetRadius()*, respectively. An example *vtkm::Cylinder* is shown in Figure 3 with set origin, radius, and axis values.

```
class Cylinder : public vtkm::internal::ImplicitFunctionBase<Cylinder>
{
public:
  Implicit function for a cylinder.
```

*Cylinder* computes the implicit function and function gradient for a cylinder using  $F(r)=r^2-Radius^2$ . By default the *Cylinder* is centered at the origin and the axis of rotation is along the y-axis. You can redefine the center and axis of rotation by setting the Center and Axis data members.

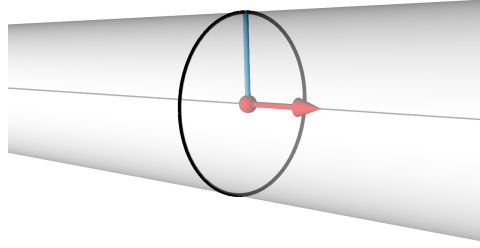


Figure 3: Visual Representation of an Implicit Cylinder. The red dot represents the center value, and the red arrow represents the vector that points in the direction of the axis. The radius is the length of any line (like the blue one shown here) that extends perpendicular from the axis to the surface.

Note that the cylinder is infinite in extent.

### Public Functions

inline **Cylinder**()

Construct cylinder radius of 0.5; centered at origin with axis along y coordinate axis.

inline **Cylinder**(const Vector &axis, Scalar radius)

Construct a cylinder with the given axis and radius.

The cylinder is centered at the origin.

inline **Cylinder**(const Vector &center, const Vector &axis, Scalar radius)

Construct a cylinder at the given center, axis, and radius.

inline void **SetCenter**(const Vector &center)

Specify the center of the cylinder.

The axis of the cylinder goes through the center.

inline void **SetAxis**(const Vector &axis)

Specify the direction of the axis of the cylinder.

inline void **SetRadius**(Scalar radius)

Specify the radius of the cylinder.

inline Scalar **Value**(const Vector &point) const

Evaluate the value of the implicit function.

The `Value()` method for an implicit function takes a `vtkm::Vec3f` and returns a `vtkm::FloatDefault` representing the orientation of the point with respect to the implicit function's shape. Negative scalar values represent vector points inside of the implicit function's shape. Positive scalar values represent vector points outside the implicit function's shape. Zero values represent vector points that lie on the surface of the implicit function.

inline Vector **Gradient**(const Vector &point) const

Evaluate the gradient of the implicit function.

The `Gradient()` method for an implicit function takes a `vtkm::Vec3f` and returns a `vtkm::Vec3f` representing the pointing direction from the implicit function's shape. Gradient calculations are more object shape specific. It is advised to look at the individual shape implementations for specific implicit functions.

## 15.4 Box

`vtkm::Box` defines an axis-aligned box. The box is defined with a pair of `vtkm::Vec3f` values that represent the minimum point coordinates and maximum point coordinates, which are set with `vtkm::Box::SetMinPoint()` and `vtkm::Box::SetMaxPoint()`, respectively. The `vtkm::Box` is the shape enclosed by intersecting axis-parallel lines drawn from each point. Alternately, the `vtkm::Box` can be specified with a `vtkm::Bounds` object using the `vtkm::Box::SetBounds()` method. An example `vtkm::Box` is shown in Figure 4.

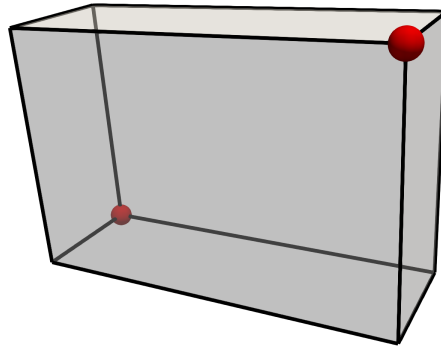


Figure 4: Visual Representation of an Implicit Box. The red dots represent the minimum and maximum points.

```
class Box : public vtkm::internal::ImplicitFunctionBase<Box>
```

Implicit function for a box.

`Box` computes the implicit function and/or gradient for a axis-aligned bounding box. Each side of the box is orthogonal to all other sides meeting along shared edges and all faces are orthogonal to the x-y-z coordinate axes.

### Public Functions

```
inline Box()
```

Construct box with center at (0,0,0) and each side of length 1.0.

```
inline Box(const Vector &minPoint, const Vector &maxPoint)
```

Construct a box with the specified minimum and maximum point.

```
inline Box(Scalar xmin, Scalar xmax, Scalar ymin, Scalar ymax, Scalar zmin, Scalar zmax)
```

Construct a box with the specified minimum and maximum point.

```
inline Box(const vtkm::Bounds &bounds)
```

Construct a box that encompasses the given bounds.

```
inline void SetMinPoint(const Vector &point)
```

Specify the minimum coordinate of the box.

```
inline void SetMaxPoint(const Vector &point)
```

Specify the maximum coordinate of the box.

```
inline const Vector &GetMinPoint() const
```

Specify the minimum coordinate of the box.

```
inline const Vector &GetMaxPoint() const
```

Specify the maximum coordinate of the box.

```
inline void SetBounds(const vtkm::Bounds &bounds)
```

Specify the size and location of the box by the bounds it encompasses.

```
inline vtkm::Bounds GetBounds() const
```

Specify the size and location of the box by the bounds it encompasses.

```
inline Scalar Value(const Vector &point) const
```

Evaluate the value of the implicit function.

The `Value()` method for an implicit function takes a `vtkm::Vec3f` and returns a `vtkm::FloatDefault` representing the orientation of the point with respect to the implicit function's shape. Negative scalar values represent vector points inside of the implicit function's shape. Positive scalar values represent vector points outside the implicit function's shape. Zero values represent vector points that lie on the surface of the implicit function.

```
inline Vector Gradient(const Vector &point) const
```

Evaluate the gradient of the implicit function.

The `Gradient()` method for an implicit function takes a `vtkm::Vec3f` and returns a `vtkm::Vec3f` representing the pointing direction from the implicit function's shape. Gradient calculations are more object shape specific. It is advised to look at the individual shape implementations for specific implicit functions.

## 15.5 Frustum

`vtkm::Frustum` defines a hexahedral region with potentially oblique faces. A `vtkm::Frustum` is typically used to define the tapered region of space visible in a perspective camera projection. The frustum is defined by the 6 planes that make up its 6 faces. Each plane is defined by a point and a normal vector, which are set with `vtkm::Frustum::SetPlane()` and `vtkm::Frustum::SetNormal()`, respectively. Parameters for all 6 planes can be set at once using the `vtkm::Frustum::SetPlanes()` and `vtkm::Frustum::SetNormals()` methods. Alternately, the `vtkm::Frustum` can be defined by the 8 points at the vertices of the enclosing hexahedron using the `vtkm::Frustum::CreateFromPoints()` method. The points given to `vtkm::Frustum::CreateFromPoints()` must be in hex-cell order where the first four points are assumed to be a plane, and the last four points are assumed to be a plane. An example `vtkm::Frustum` is shown in Figure 5.

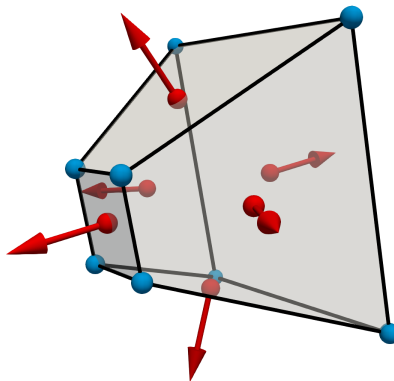


Figure 5: Visual Representation of an Implicit Frustum. The red dots and arrows represent the points and normals defining each enclosing plane. The blue dots represent the 8 vertices, which can also be used to define the frustum.

```
class Frustum : public vtkm::internal::ImplicitFunctionBase<Frustum>
    Implicit function for a frustum.
```

## 15.6 General Implicit Functions

It is often the case when creating code that uses an implicit function that you do not know which implicit function will be desired. For example, the `vtkm::filter::contour::ClipWithImplicitFunction` filter can be used with any of the implicit functions described here (`vtkm::Plane`, `vtkm::Sphere`, etc.).

To handle conditions where you want to support multiple implicit functions simultaneously, VTK-m provides `vtkm::ImplicitFunctionGeneral`. Any of the implicit functions described in this chapter can be copied to a `vtkm::ImplicitFunctionGeneral`, which will behave like the specified function. The following example shows passing a `vtkm::Sphere` to `vtkm::filter::contour::ClipWithImplicitFunction`, which internally uses `vtkm::ImplicitFunctionGeneral` to manage the implicit function types.

Example 1: Passing an implicit function to a filter.

```
1 // Parameters needed for implicit function
2 vtkm::Sphere implicitFunction(vtkm::make_Vec(1, 0, 1), 0.5);
3
4 // Create an instance of a clip filter with this implicit function.
5 vtkm::filter::contour::ClipWithImplicitFunction clip;
6 clip.SetImplicitFunction(implicitFunction);
```

```
class ImplicitFunctionGeneral : public vtkm::ImplicitFunctionMultiplexer<vtkm::Box, vtkm::Cylinder,
    vtkm::Frustum, vtkm::Plane, vtkm::Sphere, vtkm::MultiPlane<3>>
```

Implicit function that can switch among known implicit function types.

`ImplicitFunctionGeneral` can behave as any of the predefined implicit functions provided by VTK-m. This is helpful when the type of implicit function is not known at compile time. For example, say you want a filter that can operate on an implicit function. Rather than compile separate versions of the filter, one for each type of implicit function, you can compile the filter once for `ImplicitFunctionGeneral` and then set the desired implicit function at runtime.

To use `ImplicitFunctionGeneral`, simply create the actual implicit function that you want to use, and then set the `ImplicitFunctionGeneral` to that concrete implicit function object.

`ImplicitFunctionGeneral` currently supports `vtkm::Box`, `vtkm::Cylinder`, `vtkm::Frustum`, `vtkm::Plane`, and `vtkm::Sphere`.





# **Part III**

## **Developing Algorithms**



## GENERAL APPROACH

VTK-m is designed to provide a pervasive parallelism throughout all its visualization algorithms, meaning that the algorithm is designed to operate with independent concurrency at the finest possible level throughout. VTK-m provides this pervasive parallelism by providing a programming construct called a worklet, which operates on a very fine granularity of data. The worklets are designed as serial components, and VTK-m handles whatever layers of concurrency are necessary, thereby removing the onus from the visualization algorithm developer. Worklet operation is then wrapped into filter, which provide a simplified interface to end users.

A worklet is essentially a functor or kernel designed to operate on a small element of data. (The name “worklet” means work on a small amount of data.) The worklet is constrained to contain a serial and stateless function. These constraints form three critical purposes. First, the constraints on the worklets allow VTK-m to schedule worklet invocations on a great many independent concurrent threads and thereby making the algorithm pervasively parallel. Second, the constraints allow VTK-m to provide thread safety. By controlling the memory access the toolkit can insure that no worklet will have any memory collisions, false sharing, or other parallel programming pitfalls. Third, the constraints encourage good programming practices. The worklet model provides a natural approach to visualization algorithm design that also has good general performance characteristics.

VTK-m allows developers to design algorithms that are run on massive amounts of threads. However, VTK-m also allows developers to interface to applications, define data, and invoke algorithms that they have written or are provided otherwise. These two modes represent significantly different operations on the data. The operating code of an algorithm in a worklet is constrained to access only a small portion of data that is provided by the framework. Conversely, code that is building the data structures needs to manage the data in its entirety, but has little reason to perform computations on any particular element.

Consequently, VTK-m is divided into two environments that handle each of these use cases. Each environment has its own API, and direct interaction between the environments is disallowed. The environments are as follows.

- **Execution Environment** This is the environment in which the computational portion of algorithms are executed. The API for this environment provides work for one element with convenient access to information such as connectivity and neighborhood as needed by typical visualization algorithms. Code for the execution environment is designed to always execute on a very large number of threads.
- **Control Environment** This is the environment that is used to interface with applications, interface with I/O devices, and schedule parallel execution of the algorithms. The associated API is designed for users that want to use VTK-m to analyze their data using provided or supplied filters. Code for the control environment is designed to run on a single thread (or one single thread per process in an MPI job).

These dual programming environments are partially a convenience to isolate the application from the execution of the worklets and are partially a necessity to support GPU languages with host and device environments. The control and execution environments are logically equivalent to the host and device environments, respectively, in CUDA and other associated GPU languages.

Figure 1 displays the relationship between the control and execution environment. The typical workflow when using VTK-m is that first the control thread establishes a data set in the control environment and then invokes a parallel operation on the data using a filter. From there the data is logically divided into its constituent elements, which are sent

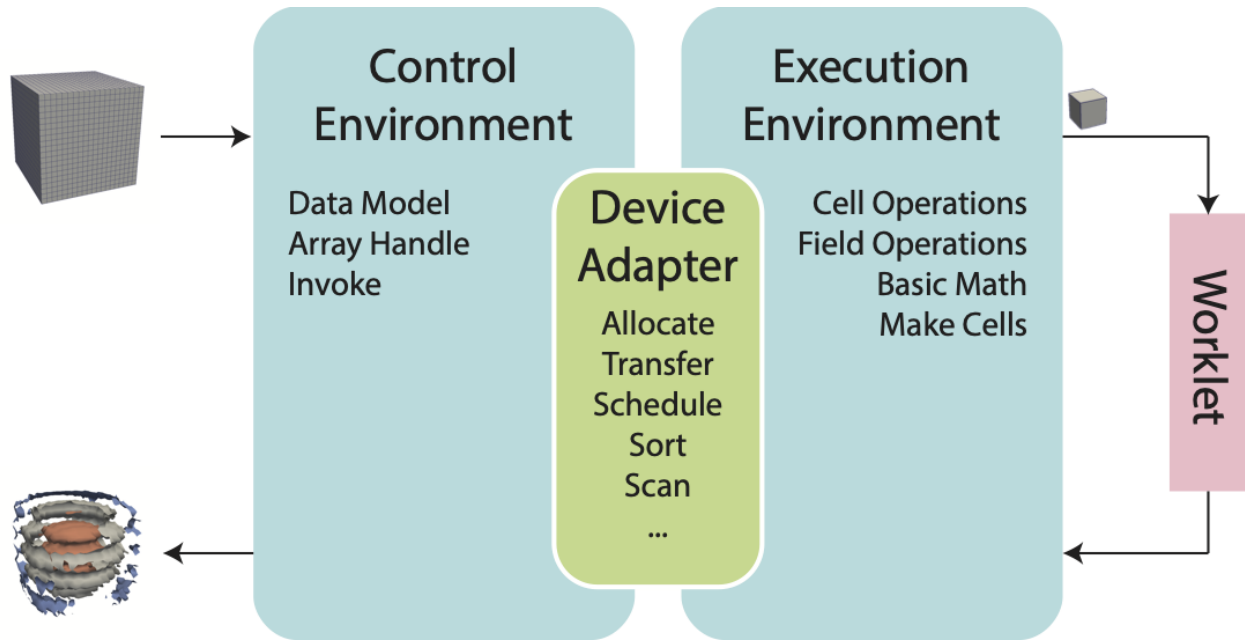


Figure 1: Diagram of the VTK-m framework.

to independent invocations of a worklet. The worklet invocations, being independent, are run on as many concurrent threads as are supported by the device. On completion the results of the worklet invocations are collected to a single data structure and a handle is returned back to the control environment.

---

#### Did You Know?

Are you only planning to use filters in VTK-m that already exist? If so, then everything you work with will be in the control environment. The execution environment is only used when implementing algorithms for filters.

---

## 16.1 Package Structure

VTK-m is organized in a hierarchy of nested packages. VTK-m places definitions in namespaces that correspond to the package (with the exception that one package may specialize a template defined in a different namespace).

The base package is named `vtkm`. All classes within VTK-m are placed either directly in the `vtkm` package or in a package beneath it. This helps prevent name collisions between VTK-m and any other library.

As described at the beginning of this chapter, the VTK-m API is divided into two distinct environments: the control environment and the execution environment. The API for these two environments are located in the `vtkm::cont` and `vtkmexec` packages, respectively. Items located in the base `vtkm` namespace are available in both environments.

---

#### Did You Know?

Although it is conventional to spell out names in identifiers (as outlined in <https://gitlab.kitware.com/vtk/vtk-m/blob/master/docs/CodingConventions.md>) there is an exception to abbreviate control and execution to `cont` and `exec`, respectively. This is because it is also part of the coding convention to declare the entire namespace when using an identifier that is part of the corresponding package. The shorter names make the identifiers easier to read, faster to type, and more feasible to pack lines in terminal displays. These abbreviations are also used instead of more common

abbreviations (e.g. ctrl for control) because, as part of actual English words, they are easier to type.

Further functionality in VTK-m is built on top of the base `vtkm`, `vtkm::cont`, and `vtkm::exec` packages. Support classes for building worklets, introduced in Chapter [Chapter 18 \(Simple Worklets\)](#), are contained in the `vtkm::worklet` package. Other facilities in VTK-m are provided in their own packages such as `vtkm::io`, `vtkm::filter`, and `vtkm::rendering`. These packages are described in [Part II \(Using VTK-m\)](#).

VTK-m contains code that uses specialized compiler features, such as those with CUDA, or libraries, such as Kokkos, that will not be available on all machines. Code for these features are encapsulated in their own packages under the `vtkm::cont` namespace: `vtkm::cont::cuda` and `vtkm::cont::kokkos`.

By convention all classes will be defined in a file with the same name as the class name (with a `.h` extension) located in a directory corresponding to the package name. For example, the `vtkm::cont::DataSet` class is found in the `vtkm/cont/DataSet.h` header. There are, however, exceptions to this rule. Some smaller classes and types are grouped together for convenience. These exceptions will be noted as necessary.

Within each namespace there may also be `internal` and `detail` sub-namespaces. The `internal` namespaces contain features that are used internally and may change without notice. The `detail` namespaces contain features that are used by a particular class but must be declared outside of that class. Users should generally ignore classes in these namespaces.

## 16.2 Function and Method Environment Modifiers

Any function or method defined by VTK-m must come with a modifier that determines in which environments the function may be run. These modifiers are C macros that VTK-m uses to instruct the compiler for which architectures to compile each method. Most user code outside of VTK-m need not use these macros with the important exception of any classes passed to VTK-m. This occurs when defining new worklets, array storage, and device adapters.

VTK-m provides three modifier macros, `VTKM_CONT`, `VTKM_EXEC`, and `VTKM_EXEC_CONT`, which are used to declare functions and methods that can run in the control environment, execution environment, and both environments, respectively. These macros get defined by including just about any VTK-m header file, but including `vtkm/Types.h` will ensure they are defined.

The modifier macro is placed after the template declaration, if there is one, and before the return type for the function. Here is a simple example of a function that will square a value. Since most types you would use this function on have operators in both the control and execution environments, the function is declared for both places.

Example 1: Usage of an environment modifier macro on a function.

```

1 template<typename ValueType>
2 VTKM_EXEC_CONT ValueType Square(const ValueType& inValue)
3 {
4     return inValue * inValue;
5 }

```

The primary function of the modifier macros is to inject compiler-specific keywords that specify what architecture to compile code for. For example, when compiling with CUDA, the control modifiers have `__host__` in them and execution modifiers have `__device__` in them.

It is sometimes the case that a function declared as `VTKM_EXEC_CONT` has to call a method declared as `VTKM_EXEC` or `VTKM_CONT`. Generally functions should not call other functions with incompatible control/execution modifiers, but sometimes a generic `VTKM_EXEC_CONT` function calls another function determined by the template parameters, and the valid environments of this subfunction may be inconsistent. For cases like this, you can use the `VTKM_SUPPRESS_EXEC_WARNINGS` to tell the compiler to ignore the inconsistency when resolving the template. When applied to a templated function or method, `VTKM_SUPPRESS_EXEC_WARNINGS` is placed before the `template` keyword.

When applied to a non-templated method in a templated class, `VTKM_SUPPRESS_EXEC_WARNINGS` is placed before the environment modifier macro.

Example 2: Suppressing warnings about functions from mixed environments.

```
1 VTKM_SUPPRESS_EXEC_WARNINGS
2 template<typename Functor>
3 VTKM_EXEC_CONT void OverlyComplicatedForLoop(Functor& functor, vtkm::Id numIterations)
4 {
5     for (vtkm::Id index = 0; index < numIterations; index++)
6     {
7         functor();
8     }
9 }
```

## BASIC ARRAY HANDLES

Chapter 7 (Data Sets) describes the basic data sets used by VTK-m. This chapter dives deeper into how VTK-m represents data. Ultimately, data structures like `vtkm::cont::DataSet` can be broken down into arrays of numbers. Arrays in VTK-m are managed by a unit called an *array handle*.

An array handle, which is implemented with the `vtkm::cont::ArrayHandle` class, manages an array of data that can be accessed or manipulated by VTK-m algorithms. It is typical to construct an array handle in the control environment to pass data to an algorithm running in the execution environment. It is also typical for an algorithm running in the execution environment to populate an array handle, which can then be read back in the control environment. It is also possible for an array handle to manage data created by one VTK-m algorithm and passed to another, remaining in the execution environment the whole time and never copied to the control environment.

---

### Did You Know?

The array handle may have multiple copies of the array, one for the control environment and one for each device. However, depending on the device and how the array is being used, the array handle will only have one copy when possible. Copies between the environments are implicit and lazy. They are copied only when an operation needs data in an environment where the data are not.

---

`vtkm::cont::ArrayHandle` behaves like a shared smart pointer in that when the C++ object is copied, each copy holds a reference to the same array. These copies are reference counted so that when all copies of the `vtkm::cont::ArrayHandle` are destroyed, any allocated memory is released.

```
template<typename T, typename StorageTag_ = ::vtkm::cont::StorageTagBasic>
```

```
class ArrayHandle : public vtkm::cont::internal::ArrayHandleBase
```

Manages an array-worth of data.

`ArrayHandle` manages an array of data that can be manipulated by VTKm algorithms. The `ArrayHandle` may have up to two copies of the array, one for the control environment and one for the execution environment, although depending on the device and how the array is being used, the `ArrayHandle` will only have one copy when possible.

An `ArrayHandle` is often constructed by instantiating one of the `ArrayHandle` subclasses. Several basic `ArrayHandle` types can also be constructed directly and then allocated. The `ArrayHandleBasic` subclass provides mechanisms for importing user arrays into an `ArrayHandle`.

`ArrayHandle` behaves like a shared smart pointer in that when it is copied each copy holds a reference to the same array. These copies are reference counted so that when all copies of the `ArrayHandle` are destroyed, any allocated memory is released.

```
Subclassed          by          vtkm::cont::ArrayHandleImplicit<          detail::PhiloxFunctor          >,
vtkm::cont::ArrayHandleRuntimeVec<          vtkm::Float32          >,          vtkm::cont::ArrayHandleTransform<
vtkm::cont::ArrayHandleRandomUniformBits,          detail::CanonicalFunctor<          vtkm::Float64          >          >,>
```

```
vtkm::cont::ArrayHandleTransform< vtkm::cont::ArrayHandleZip< vtkm::cont::ArrayHandleRandomUniformReal<
vtkm::Float64 >, vtkm::cont::ArrayHandleRandomUniformReal< vtkm::Float64 > >, detail::BoxMuller
>, vtkm::cont::ArrayHandleCartesianProduct< FirstHandleType, SecondHandleType, ThirdHandleType >,
vtkm::cont::ArrayHandleDecorator< DecoratorImplT, ArrayTs >, vtkm::cont::ArrayHandleDiscard< Value-
Type_ >, vtkm::cont::ArrayHandleImplicit< FunctorType >, vtkm::cont::ArrayHandleZip< FirstHandleType,
SecondHandleType >
```

## Public Functions

inline **ArrayHandle**()

Constructs an empty *ArrayHandle*.

inline **ArrayHandle**(const vtkm::cont::ArrayHandle<ValueType, StorageTag> &src)

Copy constructor.

Implemented so that it is defined exclusively in the control environment. If there is a separate device for the execution environment (for example, with CUDA), then the automatically generated copy constructor could be created for all devices, and it would not be valid for all devices.

inline **ArrayHandle**(vtkm::cont::ArrayHandle<ValueType, StorageTag> &&src) noexcept

Move constructor.

Implemented so that it is defined exclusively in the control environment. If there is a separate device for the execution environment (for example, with CUDA), then the automatically generated move constructor could be created for all devices, and it would not be valid for all devices.

inline explicit **ArrayHandle**(const std::vector<vtkm::cont::internal::Buffer> &buffers)

Special constructor for subclass specializations that need to set the initial state array.

Used when pulling data from other sources.

inline explicit **ArrayHandle**(std::vector<vtkm::cont::internal::Buffer> &&buffers) noexcept

Special constructor for subclass specializations that need to set the initial state array.

Used when pulling data from other sources.

inline **~ArrayHandle**()

Destructs an empty *ArrayHandle*.

Implemented so that it is defined exclusively in the control environment. If there is a separate device for the execution environment (for example, with CUDA), then the automatically generated destructor could be created for all devices, and it would not be valid for all devices.

inline vtkm::cont::ArrayHandle<ValueType, StorageTag> &**operator**=(const  
vtkm::cont::ArrayHandle<ValueType,  
StorageTag> &src)

Shallow copies an *ArrayHandle*.

inline vtkm::cont::ArrayHandle<ValueType, StorageTag> &**operator**=(vtkm::cont::ArrayHandle<ValueType,  
StorageTag> &&src) noexcept

Move and Assignment of an *ArrayHandle*.

inline bool **operator**==(const ArrayHandle<ValueType, StorageTag> &rhs) const

Like a pointer, two *ArrayHandles* are considered equal if they point to the same location in memory.

inline StorageType **GetStorage**() const

Get the storage.



inline ReadPortalType **ReadPortal**() const

Get an array portal that can be used in the control environment.

The returned array can be used in the control environment to read values from the array. (It is not possible to write to the returned portal. That is `Get` will work on the portal, but `Set` will not.)

**Note:** The returned portal cannot be used in the execution environment. This is because the portal will not work on some devices like GPUs. To get a portal that will work in the execution environment, use `PrepareForInput`.

inline WritePortalType **WritePortal**() const

Get an array portal that can be used in the control environment.

The returned array can be used in the control environment to read and write values to the array.

**Note:** The returned portal cannot be used in the execution environment. This is because the portal will not work on some devices like GPUs. To get a portal that will work in the execution environment, use `PrepareForInput`.

inline WritePortalType **WritePortal**(vtkm::cont::Token &token) const

Get an array portal that can be used in the control environment.

The returned array can be used in the control environment to read and write values to the array.

**Note:** The returned portal cannot be used in the execution environment. This is because the portal will not work on some devices like GPUs. To get a portal that will work in the execution environment, use `PrepareForInput`.

inline vtkm::Id **GetNumberOfValues**() const

Returns the number of entries in the array.

inline vtkm::IdComponent **GetNumberOfComponentsFlat**() const

Returns the total number of components for each value in the array.

If the array holds `vtkm::Vec` objects, this will return the total number of components in each value assuming the object is flattened out to one level of `Vec` objects. If the array holds a basic C type (such as `float`), this will return 1. If the array holds a simple `Vec` (such as `vtkm::Vec3f`), this will return the number of components (in this case 3). If the array holds a hierarchy of `Vecs` (such as `vtkm::Vec<vtkm::Vec3f, 2>`), this will return the total number of vecs (in this case 6).

If this object is holding an array where the number of components can be selected at runtime (for example, `vtkm::cont::ArrayHandleRuntimeVec`), this method will still return the correct number of components. However, if each value in the array can be a `Vec` of a different size (such as `vtkm::cont::ArrayHandleGroupVecVariable`), this method will return 0 (because there is no consistent answer).

inline void **Allocate**(vtkm::Id numberOfValues, vtkm::CopyFlag preserve, vtkm::cont::Token &token) const

Allocates an array large enough to hold the given number of values.

The allocation may be done on an already existing array. If so, then the data are preserved as best as possible if the preserve flag is set to `vtkm::CopyFlag::On`. If the preserve flag is set to `vtkm::CopyFlag::Off` (the default), any existing data could be wiped out.

This method can throw `vtkm::cont::ErrorBadAllocation` if the array cannot be allocated or `vtkm::cont::ErrorBadValue` if the allocation is not feasible (for example, the array storage is read-only).

inline void **Allocate**(vtkm::Id numberOfValues, vtkm::CopyFlag preserve = vtkm::CopyFlag::Off) const

Allocates an array large enough to hold the given number of values.

The allocation may be done on an already existing array. If so, then the data are preserved as best as possible if the preserve flag is set to `vtkm::CopyFlag::On`. If the preserve flag is set to `vtkm::CopyFlag::Off` (the default), any existing data could be wiped out.

This method can throw `vtkm::cont::ErrorBadAllocation` if the array cannot be allocated or `vtkm::cont::ErrorBadValue` if the allocation is not feasible (for example, the array storage is read-only).

```
inline void AllocateAndFill(vtkm::Id numberOfValues, const ValueType &fillValue, vtkm::CopyFlag  
                             preserve, vtkm::cont::Token &token) const
```

Allocates an array and fills it with an initial value.

`AllocateAndFill` behaves similar to `Allocate` except that after allocation it fills the array with a given `fillValue`. This method is convenient when you wish to initialize the array.

If the preserve flag is `vtkm::CopyFlag::On`, then any data that existed before the call to `AllocateAndFill` will remain after the call (assuming the new array size is large enough). If the array size is expanded, then the new values at the end will be filled.

If the preserve flag is `vtkm::CopyFlag::Off` (the default), the entire array is filled with the given `fillValue`.

```
inline void AllocateAndFill(vtkm::Id numberOfValues, const ValueType &fillValue, vtkm::CopyFlag  
                             preserve = vtkm::CopyFlag::Off) const
```

Allocates an array and fills it with an initial value.

`AllocateAndFill` behaves similar to `Allocate` except that after allocation it fills the array with a given `fillValue`. This method is convenient when you wish to initialize the array.

If the preserve flag is `vtkm::CopyFlag::On`, then any data that existed before the call to `AllocateAndFill` will remain after the call (assuming the new array size is large enough). If the array size is expanded, then the new values at the end will be filled.

If the preserve flag is `vtkm::CopyFlag::Off` (the default), the entire array is filled with the given `fillValue`.

```
inline void Fill(const ValueType &fillValue, vtkm::Id startIndex, vtkm::Id endIndex, vtkm::cont::Token  
                 &token) const
```

Fills the array with a given value.

After calling this method, every entry in the array from `startIndex` (inclusive) to `endIndex` (exclusive) of the array is set to `fillValue`. If `startIndex` or `endIndex` is not specified, then the fill happens from the beginning or end, respectively.

```
inline void Fill(const ValueType &fillValue, vtkm::Id startIndex, vtkm::Id endIndex) const
```

Fills the array with a given value.

After calling this method, every entry in the array from `startIndex` (inclusive) to `endIndex` (exclusive) of the array is set to `fillValue`. If `startIndex` or `endIndex` is not specified, then the fill happens from the beginning or end, respectively.

```
inline void Fill(const ValueType &fillValue, vtkm::Id startIndex = 0) const
```

Fills the array with a given value.

After calling this method, every entry in the array from `startIndex` (inclusive) to `endIndex` (exclusive) of the array is set to `fillValue`. If `startIndex` or `endIndex` is not specified, then the fill happens from the beginning or end, respectively.

inline void **ReleaseResourcesExecution()** const

Releases any resources being used in the execution environment (that are not being shared by the control environment).

inline void **ReleaseResources()** const

Releases all resources in both the control and execution environments.

inline ReadPortalType **PrepareForInput**(vtkm::cont::DeviceAdapterId device, vtkm::cont::Token &token) const

Prepares this array to be used as an input to an operation in the execution environment.

If necessary, copies data to the execution environment. Can throw an exception if this array does not yet contain any data. Returns a portal that can be used in code running in the execution environment.

The Token object provided will be attached to this [ArrayHandle](#). The returned portal is guaranteed to be valid while the Token is still attached and in scope. Other operations on this [ArrayHandle](#) that would invalidate the returned portal will block until the Token is released. Likewise, this method will block if another Token is already attached. This can potentially lead to deadlocks.

inline WritePortalType **PrepareForInPlace**(vtkm::cont::DeviceAdapterId device, vtkm::cont::Token &token) const

Prepares this array to be used in an in-place operation (both as input and output) in the execution environment.

If necessary, copies data to the execution environment. Can throw an exception if this array does not yet contain any data. Returns a portal that can be used in code running in the execution environment.

The Token object provided will be attached to this [ArrayHandle](#). The returned portal is guaranteed to be valid while the Token is still attached and in scope. Other operations on this [ArrayHandle](#) that would invalidate the returned portal will block until the Token is released. Likewise, this method will block if another Token is already attached. This can potentially lead to deadlocks.

inline WritePortalType **PrepareForOutput**(vtkm::Id numberOfValues, vtkm::cont::DeviceAdapterId device, vtkm::cont::Token &token) const

Prepares (allocates) this array to be used as an output from an operation in the execution environment.

The internal state of this class is set to have valid data in the execution array with the assumption that the array will be filled soon (i.e. before any other methods of this object are called). Returns a portal that can be used in code running in the execution environment.

The Token object provided will be attached to this [ArrayHandle](#). The returned portal is guaranteed to be valid while the Token is still attached and in scope. Other operations on this [ArrayHandle](#) that would invalidate the returned portal will block until the Token is released. Likewise, this method will block if another Token is already attached. This can potentially lead to deadlocks.

inline bool **IsOnDevice**(vtkm::cont::DeviceAdapterId device) const

Returns true if the [ArrayHandle](#)'s data is on the given device.

If the data are on the given device, then preparing for that device should not require any data movement.

inline bool **IsOnHost**() const

Returns true if the [ArrayHandle](#)'s data is on the host.

If the data are on the given device, then calling ReadPortal or WritePortal should not require any data movement.

inline void **SyncControlArray**() const

Synchronizes the control array with the execution array.

If either the user array or control array is already valid, this method does nothing (because the data is already available in the control environment). Although the internal state of this class can change, the method is declared const because logically the data does not.

inline void **Enqueue**(const vtkm::cont::Token &token) const

Enqueue a token for access to this *ArrayHandle*.

This method places the given Token into the queue of Tokens waiting for access to this *ArrayHandle* and then returns immediately. When this token is later used to get data from this *ArrayHandle* (for example, in a call to *PrepareForInput*), it will use this place in the queue while waiting for access.

This method is to be used to ensure that a set of accesses to an *ArrayHandle* that happen on multiple threads occur in a specified order. For example, if you spawn of a job to modify data in an *ArrayHandle* and then spawn off a job that reads that same data, you need to make sure that the first job gets access to the *ArrayHandle* before the second. If they both just attempt to call their respective *Prepare* methods, there is no guarantee which order they will occur. Having the spawning thread first call this method will ensure the order.

**Warning:** After calling this method it is required to subsequently call a method like one of the *Prepare* methods that attaches the token to this *ArrayHandle*. Otherwise, the enqueued token will block any subsequent access to the *ArrayHandle*, even if the Token is destroyed.

inline void **DeepCopyFrom**(const vtkm::cont::ArrayHandle<ValueType, StorageTag> &source) const

Deep copies the data in the array.

Takes the data that is in *source* and copies that data into this array.

inline const std::vector<vtkm::cont::internal::Buffer> &**GetBuffers**() const

Returns the internal Buffer structures that hold the data.

Note that great care should be taken when modifying buffers outside of the *ArrayHandle*.

## 17.1 Creating Array Handles

*vtkm::cont::ArrayHandle* is templated on the type of values being stored in the array. There are multiple ways to create and populate an array handle. The default *vtkm::cont::ArrayHandle* constructor will create an empty array with nothing allocated in either the control or execution environment. This is convenient for creating arrays used as the output for algorithms.

Example 1: Creating an *vtkm::cont::ArrayHandle* for output data.

```
1 vtkm::cont::ArrayHandle<vtkm::Float32> outputArray;
```

Chapter ref{chap:AccessingAllocatingArrays} describes in detail how to allocate memory and access data in an *vtkm::cont::ArrayHandle*. However, you can use the *vtkm::cont::make\_ArrayHandle()* function for a simplified way to create an *vtkm::cont::ArrayHandle* with data.

*vtkm::cont::make\_ArrayHandle()* has many forms. An easy form to use takes an initializer list and creates a basic *vtkm::cont::ArrayHandle* with it. This allows you to create a short *vtkm::cont::ArrayHandle* from literals.

template<typename T>

vtkm::cont::ArrayHandleBasic<T> vtkm::cont::make\_ArrayHandle(std::initializer\_list<T> &&values)

Create an *ArrayHandle* directly from an initializer list of values.

Example 2: Creating an `vtkm::cont::ArrayHandle` from initially specified values.

```
1 auto fibonacciArray = vtkm::cont::make_ArrayHandle({ 0, 1, 1, 2, 3, 5, 8, 13 });
```

One problem with creating an array from an initializer list like this is that it can be tricky to specify the exact value type of the `vtkm::cont::ArrayHandle`. The value type of the `vtkm::cont::ArrayHandle` will be the same types as the literals in the initializer list, but that might not match the type you actually need. This is particularly true for types like `vtkm::Id` and `vtkm::FloatDefault`, which can change depending on compile options. To specify the exact value type to use, give that type as a template argument to the `vtkm::cont::make_ArrayHandle()` function.

Example 3: Creating a typed `vtkm::cont::ArrayHandle` from initially specified values.

```
1 vtkm::cont::ArrayHandle<vtkm::FloatDefault> inputArray =
2   vtkm::cont::make_ArrayHandle<vtkm::FloatDefault>({ 1.4142f, 2.7183f, 3.1416f });
```

Constructing an `vtkm::cont::ArrayHandle` that points to a provided C array is also straightforward. To do this, call `vtkm::cont::make_ArrayHandle()` with the array pointer, the number of values in the C array, and a `vtkm::CopyFlag`. This last argument can be either `vtkm::CopyFlag::On` to copy the array or `vtkm::CopyFlag::Off` to share the provided buffer.

```
template<typename T>
vtkm::cont::ArrayHandleBasic<T> vtkm::cont::make_ArrayHandle(const T *array, vtkm::Id numberOfValues,
                                                             vtkm::CopyFlag copy)
```

A convenience function for creating an `ArrayHandle` from a standard C array.

enum class `vtkm::CopyFlag`

Identifier used to specify whether a function should deep copy data.

Values:

enumerator `Off`

enumerator `On`

Example 4: Creating an `vtkm::cont::ArrayHandle` that points to a provided C array.

```
1 vtkm::Float32 dataBuffer[50];
2 // Populate dataBuffer with meaningful data. Perhaps read data from a file.
3
4 vtkm::cont::ArrayHandle<vtkm::Float32> inputArray =
5   vtkm::cont::make_ArrayHandle(dataBuffer, 50, vtkm::CopyFlag::On);
```

Likewise, you can use `vtkm::cont::make_ArrayHandle()` to transfer data from a `std::vector` to an `vtkm::cont::ArrayHandle`. This form of `vtkm::cont::make_ArrayHandle()` takes the `std::vector` as the first argument and a `vtkm::CopyFlag` as the second argument.

```
template<typename T, typename Allocator>
vtkm::cont::ArrayHandleBasic<T> vtkm::cont::make_ArrayHandle(const std::vector<T, Allocator> &array,
                                                             vtkm::CopyFlag copy)
```

A convenience function for creating an `ArrayHandle` from an `std::vector`.

Example 5: Creating an `vtkm::cont::ArrayHandle` that points to a provided `std::vector`.

```
1  std::vector<vtkm::Float32> dataBuffer;
2  // Populate dataBuffer with meaningful data. Perhaps read data from a file.
3
4  vtkm::cont::ArrayHandle<vtkm::Float32> inputArray =
5  vtkm::cont::make_ArrayHandle(dataBuffer, vtkm::CopyFlag::On);
```

As hinted at earlier, it is possible to send `vtkm::CopyFlag::On` to `vtkm::cont::make_ArrayHandle()` to wrap an `vtkm::cont::ArrayHandle` around an existing C array or `std::vector`. Doing so allows you to send the data to the `vtkm::cont::ArrayHandle` without copying it. It also provides a mechanism for VTK-m to write directly into your array. However, *be aware* that if you change or delete the data provided, the internal state of `vtkm::cont::ArrayHandle` becomes invalid and undefined behavior can ensue. A common manifestation of this error happens when a `std::vector` goes out of scope. This subtle interaction will cause the `vtkm::cont::ArrayHandle` to point to an unallocated portion of the memory heap. The following example provides an erroneous use of `vtkm::cont::ArrayHandle` and some ways to fix it.

Example 6: Invalidating an `vtkm::cont::ArrayHandle` by letting the source `std::vector` leave scope.

```
1  VTKM_CONT vtkm::cont::ArrayHandle<vtkm::Float32> BadDataLoad()
2  {
3      std::vector<vtkm::Float32> dataBuffer;
4      // Populate dataBuffer with meaningful data. Perhaps read data from a file.
5
6      vtkm::cont::ArrayHandle<vtkm::Float32> inputArray =
7      vtkm::cont::make_ArrayHandle(dataBuffer, vtkm::CopyFlag::Off);
8
9      return inputArray;
10     // THIS IS WRONG! At this point dataBuffer goes out of scope and deletes its
11     // memory. However, inputArray has a pointer to that memory, which becomes an
12     // invalid pointer in the returned object. Bad things will happen when the
13     // ArrayHandle is used.
14 }
15
16 VTKM_CONT vtkm::cont::ArrayHandle<vtkm::Float32> SafeDataLoad1()
17 {
18     std::vector<vtkm::Float32> dataBuffer;
19     // Populate dataBuffer with meaningful data. Perhaps read data from a file.
20
21     vtkm::cont::ArrayHandle<vtkm::Float32> inputArray =
22     vtkm::cont::make_ArrayHandle(dataBuffer, vtkm::CopyFlag::On);
23
24     return inputArray;
25     // This is safe.
26 }
27
28 VTKM_CONT vtkm::cont::ArrayHandle<vtkm::Float32> SafeDataLoad2()
29 {
30     std::vector<vtkm::Float32> dataBuffer;
31     // Populate dataBuffer with meaningful data. Perhaps read data from a file.
32 }
```

(continues on next page)



(continued from previous page)

```

33   vtkm::cont::ArrayHandle<vtkm::Float32> inputArray =
34       vtkm::cont::make_ArrayHandleMove(std::move(dataBuffer));
35
36   return inputArray;
37   // This is safe.
38 }

```

An easy way around the problem of having an `vtkm::cont::ArrayHandle`'s data going out of scope is to copy the data into the `vtkm::cont::ArrayHandle`. Simply make the `vtkm::CopyFlag` argument be `vtkm::CopyFlag::On` to copy the data. This solution is shown in Example 6, line 22.

What if you have a `std::vector` that you want to pass to an `vtkm::cont::ArrayHandle` and then want to only use in the `vtkm::cont::ArrayHandle`? In this case, it is wasteful to have to copy the data, but you also do not want to be responsible for keeping the `std::vector` in scope. To handle this, there is a special `vtkm::cont::make_ArrayHandleMove()` that will move the memory out of the `std::vector` and into the `vtkm::cont::ArrayHandle`. `vtkm::cont::make_ArrayHandleMove()` takes an “rvalue” version of a `std::vector`. To create an “rvalue”, use the `std::move` function provided by C++. Once `vtkm::cont::make_ArrayHandleMove()` is called, the provided `std::vector` becomes invalid and any further access to it is undefined. This solution is shown in `exlineref:ex:ArrayOutOfScope:MoveVector``.

```

template<typename T, typename Allocator>
vtkm::cont::ArrayHandleBasic<T> vtkm::cont::make_ArrayHandleMove(std::vector<T, Allocator> &&array)

```

Move an `std::vector` into an `ArrayHandle`.

```

template<typename T, typename Allocator>
vtkm::cont::ArrayHandleBasic<T> vtkm::cont::make_ArrayHandle(std::vector<T, Allocator> &&array,
                                                             vtkm::CopyFlag)

```

Move an `std::vector` into an `ArrayHandle`.

## 17.2 Deep Array Copies

As stated previously, an `vtkm::cont::ArrayHandle` object behaves as a smart pointer that copies references to the data without copying the data itself. This is clearly faster and more memory efficient than making copies of the data itself and usually the behavior desired. However, it is sometimes the case that you need to make a separate copy of the data.

The easiest way to copy an `vtkm::cont::ArrayHandle` is to use the `vtkm::cont::ArrayHandle::DeepCopyFrom()` method.

Example 7: Deep copy a `vtkm::cont::ArrayHandle` of the same type.

```

1   destArray.DeepCopyFrom(srcArray);

```

However, the `vtkm::cont::ArrayHandle::DeepCopyFrom()` method only works if the two `vtkm::cont::ArrayHandle` objects are the exact same type. To simplify copying the data between `vtkm::cont::ArrayHandle` objects of different types, VTK-m comes with the `vtkm::cont::ArrayCopy()` convenience function defined in `vtkm/cont/ArrayCopy.h`. `vtkm::cont::ArrayCopy()` takes the array to copy from (the source) as its first argument and the array to copy to (the destination) as its second argument. The destination array will be properly reallocated to the correct size.

Example 8: Using `vtkm::cont::ArrayCopy()`.

```
1  vtkm::cont::ArrayCopy(srcArray, destArray);
```

```
template<typename SourceArrayType, typename DestArrayType>  
inline void vtkm::cont::ArrayCopy(const SourceArrayType &source, DestArrayType &destination)
```

Does a deep copy from one array to another array.

Given a source `ArrayHandle` and a destination `ArrayHandle`, this function allocates the destination `ArrayHandle` to the correct size and deeply copies all the values from the source to the destination.

This method will attempt to copy the data using the device that the input data is already valid on. If the input data is only valid in the control environment, the runtime device tracker is used to try to find another device.

This should work on some non-writable array handles as well, as long as both *source* and *destination* are the same type.

This version of array copy uses a precompiled version of copy that is efficient for most standard memory layouts. However, there are some types of fancy `ArrayHandle` that cannot be handled directly, and the fallback for these arrays can be slow. If you see a warning in the log about an inefficient memory copy when extracting a component, pay heed and look for a different way to copy the data (perhaps using `ArrayCopyDevice`).

```
template<typename SourceArrayType>  
inline void vtkm::cont::ArrayCopy(const SourceArrayType &source, vtkm::cont::UnknownArrayHandle  
                                &destination)
```

Does a deep copy from one array to another array.

Given a source `ArrayHandle` and a destination `ArrayHandle`, this function allocates the destination `ArrayHandle` to the correct size and deeply copies all the values from the source to the destination.

This method will attempt to copy the data using the device that the input data is already valid on. If the input data is only valid in the control environment, the runtime device tracker is used to try to find another device.

This should work on some non-writable array handles as well, as long as both *source* and *destination* are the same type.

This version of array copy uses a precompiled version of copy that is efficient for most standard memory layouts. However, there are some types of fancy `ArrayHandle` that cannot be handled directly, and the fallback for these arrays can be slow. If you see a warning in the log about an inefficient memory copy when extracting a component, pay heed and look for a different way to copy the data (perhaps using `ArrayCopyDevice`).

## 17.3 The Hidden Second Template Parameter

We have already seen that `vtkm::cont::ArrayHandle` is a templated class with the template parameter indicating the type of values stored in the array. However, `vtkm::cont::ArrayHandle` has a second hidden parameter that indicates the `_storage_` of the array. We have so far been able to ignore this second template parameter because VTK-m will assign a default storage for us that will store the data in a basic array.

Changing the storage of an `vtkm::cont::ArrayHandle` lets us do many weird and wonderful things. We will explore these options in later chapters, but for now we can ignore this second storage template parameter. However, there are a couple of things to note concerning the storage.

First, if the compiler gives an error concerning your use of `vtkm::cont::ArrayHandle`, the compiler will report the `vtkm::cont::ArrayHandle` type with not one but two template parameters. A second template parameter of `vtkm::cont::StorageTagBasic` can be ignored.



Second, if you write a function, method, or class that is templated based on an `vtkm::cont::ArrayHandle` type, it is good practice to accept an `vtkm::cont::ArrayHandle` with a non-default storage type. There are two ways to do this. The first way is to template both the value type and the storage type.

Example 9: Templating a function on an `vtkm::cont::ArrayHandle`'s parameters.

```
1 template<typename T, typename Storage>
2 void Foo(const vtkm::cont::ArrayHandle<T, Storage>& array)
3 {
```

The second way is to template the whole array type rather than the sub types. If you create a template where you expect one of the parameters to be an `vtkm::cont::ArrayHandle`, you should use the `VTKM_IS_ARRAY_HANDLE` macro to verify that the type is indeed an `vtkm::cont::ArrayHandle`.

#### `VTKM_IS_ARRAY_HANDLE(T)`

Checks that the given type is a `vtkm::cont::ArrayHandle`.

If the type is not a `vtkm::cont::ArrayHandle` or a subclass, a static assert will cause a compile exception. This is a good way to ensure that a template argument that is assumed to be an array handle type actually is.

Example 10: A template parameter that should be an `vtkm::cont::ArrayHandle`.

```
1 template<typename ArrayType>
2 void Bar(const ArrayType& array)
3 {
4     VTKM_IS_ARRAY_HANDLE(ArrayType);
```

## 17.4 Mutability

One subtle feature of `vtkm::cont::ArrayHandle` is that the class is, in principle, a pointer to an array pointer. This means that the data in an `vtkm::cont::ArrayHandle` is always mutable even if the class is declared `const`. You can change the contents of “constant” arrays via methods like `vtkm::cont::ArrayHandle::WritePortal()` and `vtkm::cont::ArrayHandle::PrepareForOutput()`. It is even possible to change the underlying array allocation with methods like `vtkm::cont::ArrayHandle::Allocate()` and `vtkm::cont::ArrayHandle::ReleaseResources()`. The upshot is that you can (sometimes) pass output arrays as constant `vtkm::cont::ArrayHandle` references.

So if a constant `vtkm::cont::ArrayHandle` can have its contents modified, what is the difference between a constant reference and a non-constant reference? The difference is that the constant reference can change the array's content, but not the array itself. Basically, this means that you cannot perform shallow copies into a `const vtkm::cont::ArrayHandle`. This can be a pretty big limitation, and many of VTK-m's internal device algorithms still require non-constant references for outputs.



## SIMPLE WORKLETS

The simplest way to implement an algorithm in VTK-m is to create a *worklet*. A worklet is fundamentally a functor that operates on an element of data. Thus, it is a `class` or `struct` that has an overloaded parenthesis operator (which must be declared `const` for thread safety). However, worklets are also embedded with a significant amount of metadata on how the data should be managed and how the execution should be structured.

Example 1: A simple worklet.

```
1 struct PoundsPerSquareInchToNewtonsPerSquareMeterWorklet : vtkm::worklet::WorkletMapField
2 {
3     using ControlSignature = void(FieldIn psi, FieldOut nsm);
4     using ExecutionSignature = void(_1, _2);
5     using InputDomain = _1;
6
7     template<typename T>
8     VTKM_EXEC void operator()(const T& psi, T& nsm) const
9     {
10         // 1 psi = 6894.76 N/m^2
11         nsm = T(6894.76f) * psi;
12     }
13 };
```

As can be seen in [Example 1](#), a worklet is created by implementing a `class` or `struct` with the following features.

1. The class must publicly inherit from a base worklet class that specifies the type of operation being performed ([Example 1](#), line 1).
2. The class must contain a functional type named `ControlSignature` ([Example 1](#), line 3), which specifies what arguments are expected when invoking the class in the control environment.
3. The class must contain a functional type named `ExecutionSignature` ([Example 1](#), line 4), which specifies how the data gets passed from the arguments in the control environment to the worklet running in the execution environment.
4. The class specifies an `InputDomain` ([Example 1](#), line 5), which identifies which input parameter defines the input domain of the data.
5. The class must contain an implementation of the parenthesis operator, which is the method that is executed in the execution environment (lines 7–12). The parenthesis operator must be declared `const`.

## 18.1 Control Signature

The control signature of a worklet is a functional type named `ControlSignature`. The function prototype matches what data are provided when the worklet is invoked (as described in [Section 18.5 \(Invoking a Worklet\)](#)).

Example 2: A `ControlSignature`.

```
using ControlSignature = void(FieldIn psi, FieldOut nsm);
```

---

### Did You Know?

If the code in [Example 2](#) looks strange, you may be unfamiliar with function types. In C++, functions have types just as variables and classes do. A function with a prototype like

```
void functionName(int arg1, float arg2);
```

has the type `void(int, float)`. VTK-m uses function types like this as a signature that defines the structure of a function call.

---

The return type of the function prototype is always `void`. The parameters of the function prototype are *tags* that identify the type of data that is expected to be passed to invoke. `ControlSignature` tags are defined by the worklet type and the various tags are documented more fully in [Chapter 22 \(Worklet Types\)](#). In the case of [Example 2](#), the two tags `FieldIn` and `FieldOut` represent input and output data, respectively.

By convention, `ControlSignature` tag names start with the base concept (e.g. `Field` or `Topology`) followed by the domain (e.g. `Point` or `Cell`) followed by `In` or `Out`. For example, `FieldPointIn` would specify values for a field on the points of a mesh that are used as input (read only). Although they should be there in most cases, some tag names might leave out the domain or in/out parts if they are obvious or ambiguous.

## 18.2 Execution Signature

Like the control signature, the execution signature of a worklet is a functional type named `ExecutionSignature`. The function prototype must match the parenthesis operator (described in [Section 18.4 \(Worklet Operator\)](#)) in terms of arity and argument semantics.

Example 3: An `ExecutionSignature`.

```
using ExecutionSignature = void(_1, _2);
```

The arguments of the `ExecutionSignature`'s function prototype are tags that define where the data come from. The most common tags are an underscore followed by a number, such as `_1`, `_2`, etc. These numbers refer back to the corresponding argument in the `ControlSignature`. For example, `_1` means data from the first control signature argument, `_2` means data from the second control signature argument, etc.

Unlike the control signature, the execution signature optionally can declare a return type if the parenthesis operator returns a value. If this is the case, the return value should be one of the numeric tags (i.e. `_1`, `_2`, etc.) to refer to one of the data structures of the control signature. If the parenthesis operator does not return a value, then `ExecutionSignature` should declare the return type as `void`.

In addition to the numeric tags, there are other execution signature tags to represent other types of data. For example, the `WorkIndex` tag identifies the instance of the worklet invocation. Each call to the worklet function will have a unique `WorkIndex`. Other such tags exist and are described in the following section on worklet types where appropriate.

## 18.3 Input Domain

All worklets represent data parallel operations that are executed over independent elements in some domain. The type of domain is inherent from the worklet type, but the size of the domain is dependent on the data being operated on.

A worklet identifies the argument specifying the domain with a type alias named `InputDomain`. The `InputDomain` must be aliased to one of the execution signature numeric tags (i.e. `_1`, `_2`, etc.). By default, the `InputDomain` points to the first argument, but a worklet can override that to point to any argument.

Example 4: An `InputDomain` declaration.

```
1 using InputDomain = _1;
```

Different types of worklets can have different types of domain. For example a simple field map worklet has a `FieldIn` argument as its input domain, and the size of the input domain is taken from the size of the associated field array. Likewise, a worklet that maps topology has a `CellSetIn` argument as its input domain, and the size of the input domain is taken from the cell set.

Specifying the `InputDomain` is optional. If it is not specified, the first argument is assumed to be the input domain.

## 18.4 Worklet Operator

A worklet is fundamentally a functor that operates on an element of data. Thus, the algorithm that the worklet represents is contained in or called from the parenthesis operator method.

Example 5: An overloaded parenthesis operator of a worklet.

```
1 template<typename T>
2 VTKM_EXEC void operator()(const T& psi, T& nsm) const
3 {
```

There are some constraints on the parenthesis operator. First, it must have the same arity as the `ExecutionSignature`, and the types of the parameters and return must be compatible. Second, because it runs in the execution environment, it must be declared with the `VTKM_EXEC` (or `VTKM_EXEC_CONT`) modifier. Third, the method must be declared `const` to help preserve thread safety.

## 18.5 Invoking a Worklet

Previously in this chapter we discussed creating a simple worklet. In this section we describe how to run the worklet in parallel.

A worklet is run using the `vtkm::cont::Invoker` class.

Example 6: Invoking a worklet.

```
1 vtkm::cont::ArrayHandle<vtkm::FloatDefault> psiArray;
2 // Fill psiArray with values...
3
4 vtkm::cont::Invoker invoke;
5
6 vtkm::cont::ArrayHandle<vtkm::FloatDefault> nsmArray;
7 invoke(PoundsPerSquareInchToNewtonsPerSquareMeterWorklet{}, psiArray, nsmArray);
```

Using an `vtkm::cont::Invoker` is simple. First, an `vtkm::cont::Invoker` can be simply constructed with no arguments (Example 6, line 4). Next, the `vtkm::cont::Invoker` is called as if it were a function (Example 6, line 7).

The first argument to the invoke is always an instance of the worklet. The remaining arguments are data that are passed (indirectly) to the worklet. Each of these arguments (after the worklet) match a corresponding argument listed in the `ControlSignature`. So in the invocation in Example 6, line 7, the second and third arguments correspond the the two `ControlSignature` arguments given in Example 2. `psiArray` corresponds to the `FieldIn` argument and `nmsArray` corresponds to the `FieldOut` argument.

#### struct **Invoker**

Allows launching any worklet without a dispatcher.

`Invoker` is a generalized `Dispatcher` that is able to automatically determine how to properly launch/invoke any worklet that is passed to it. When an `Invoker` is constructed it is provided the desired device adapter that all worklets invoked by it should be launched on.

`Invoker` is designed to not only reduce the verbosity of constructing multiple dispatchers inside a block of logic, but also makes it easier to make sure all worklets execute on the same device.

#### Public Functions

inline explicit **Invoker**()

Constructs an `Invoker` that will try to launch worklets on any device that is enabled.

inline explicit **Invoker**(vtkm::cont::DeviceAdapterId device)

Constructs an `Invoker` that will try to launch worklets only on the provided device adapter.

```
template<typename Worklet, typename T, typename ...Args, typename
std::enable_if<detail::scatter_or_mask<T>::value, int>::type* = nullptr>
```

```
inline void operator()(Worklet &&worklet, T &&scatterOrMask, Args&&... args) const
```

Launch the worklet that is provided as the first parameter.

Optional second parameter is either the scatter or mask type associated with the worklet. Any additional parameters are the `ControlSignature` arguments for the worklet.

```
template<typename Worklet, typename T, typename U, typename ...Args, typename
std::enable_if<detail::scatter_or_mask<T>::value && detail::scatter_or_mask<U>::value, int>::type* =
nullptr>
```

```
inline void operator()(Worklet &&worklet, T &&scatterOrMaskA, U &&scatterOrMaskB, Args&&...
args) const
```

Launch the worklet that is provided as the first parameter.

Optional second parameter is either the scatter or mask type associated with the worklet. Optional third parameter is either the scatter or mask type associated with the worklet. Any additional parameters are the `ControlSignature` arguments for the worklet.

```
template<typename Worklet, typename T, typename ...Args, typename
std::enable_if<!detail::scatter_or_mask<T>::value, int>::type* = nullptr>
```

```
inline void operator()(Worklet &&worklet, T &&t, Args&&... args) const
```

Launch the worklet that is provided as the first parameter.

Optional second parameter is either the scatter or mask type associated with the worklet. Any additional parameters are the `ControlSignature` arguments for the worklet.

```
inline vtkm::cont::DeviceAdapterId GetDevice() const
```

Get the device adapter that this `Invoker` is bound too.

## 18.6 Preview of More Complex Worklets

This chapter demonstrates the creation of a worklet that performs a very simple math operation in parallel. However, we have just scratched the surface of the kinds of algorithms that can be expressed with VTK-m worklets. There are many more execution patterns and data handling constructs. The following example gives a preview of some of the more advanced features of worklets.

Example 7: A more complex worklet.

```

1  struct EdgesExtract : vtkm::worklet::WorkletVisitCellsWithPoints
2  {
3      using ControlSignature = void(CellSetIn, FieldOutCell edgeIndices);
4      using ExecutionSignature = void(CellShape, PointIndices, VisitIndex, _2);
5      using InputDomain = _1;
6
7      using ScatterType = vtkm::worklet::ScatterCounting;
8
9      template<typename CellShapeTag,
10              typename PointIndexVecType,
11              typename EdgeIndexVecType>
12      VTKM_EXEC void operator()(CellShapeTag cellShape,
13                              const PointIndexVecType& globalPointIndicesForCell,
14                              vtkm::IdComponent edgeIndex,
15                              EdgeIndexVecType& edgeIndices) const
16  {

```

We will discuss the many features available in the worklet framework throughout [Part IV \(Advanced Development\)](#).





## BASIC FILTER IMPLEMENTATION

Chapter 18 (Simple Worklets) introduced the concept of a worklet and demonstrated how to create and run one to execute an algorithm on a device. Although worklets provide a powerful mechanism for designing heavily threaded visualization algorithms, invoking them requires quite a bit of knowledge of the workings of VTK-m. Instead, most users execute algorithms in VTK-m using filters. Thus, to expose algorithms implemented with worklets to general users, we need to implement a filter to encapsulate the worklets. In this chapter we will create a filter that encapsulates the worklet algorithm presented in Chapter 18 (Simple Worklets), which converted the units of a pressure field from pounds per square inch (psi) to Newtons per square meter ( $\text{N/m}^2$ ).

Filters in VTK-m are implemented by deriving `vtkm::filter::Filter`.

The following example shows the declaration of our pressure unit conversion filter. VTK-m filters are divided into libraries. In this example, we are assuming this filter is being compiled in a library named `vtkm::filter::unit_conversion`. By convention, the source files would be placed in a directory named `vtkm/filter/unit_conversion`.

Example 1: Header declaration for a simple filter.

```
1 namespace vtkm
2 {
3   namespace filter
4   {
5     namespace unit_conversion
6     {
7
8       class VTKM_FILTER_UNIT_CONVERSION_EXPORT PoundsPerSquareInchToNewtonsPerSquareMeterFilter
9       : public vtkm::filter::Filter
10      {
11      public:
12        VTKM_CONT PoundsPerSquareInchToNewtonsPerSquareMeterFilter();
13
14        VTKM_CONT vtkm::cont::DataSet DoExecute(const vtkm::cont::DataSet& inDataSet) override;
15      };
16
17    }
18  }
19 } // namespace vtkm::filter::unit_conversion
```

It is typical for a filter to have a constructor to set up its initial state. A filter will also override the `vtkm::filter::Filter::DoExecute()` method. The `vtkm::filter::Filter::DoExecute()` method takes a `vtkm::cont::DataSet` as input and likewise returns a `vtkm::cont::DataSet` containing the results of the filter operation.

```
virtual vtkm::cont::DataSet vtkm::filter::Filter::DoExecute(const vtkm::cont::DataSet &inData) = 0
```

Note that the declaration of the `PoundsPerSquareInchToNewtonsPerSquareMeterFilter` contains the export macro `VTKM_FILTER_UNIT_CONVERSION_EXPORT`. This is a macro generated by CMake to handle the appropriate modifies for exporting a class from a library. Remember that this code is to be placed in a library named `vtkm::filter::unit_conversion`. For this library, CMake creates a header file named `vtkm/filter/unit_conversion.h` that declares macros like `VTKM_FILTER_UNIT_CONVERSION_EXPORT`.

### Did You Know?

A filter can also override the `vtkm::filter::Filter::DoExecutePartitions()`, which operates on a `vtkm::cont::PartitionedDataSet`. If `vtkm::filter::Filter::DoExecutePartitions()` is not overridden, then the filter will call `vtkm::filter::Filter::DoExecute()` on each of the partitions and build a new `vtkm::cont::PartitionedDataSet` with the outputs.

```
virtual vtkm::cont::PartitionedDataSet vtkm::filter::Filter::DoExecutePartitions(const
                                                                    vtkm::cont::PartitionedDataSet
                                                                    &inData)
```

Once the filter class is declared in the `.h` file, the filter implementation is by convention given in a separate `.cxx` file. Given the definition of our filter in [Example 1](#), we will need to provide the implementation for the constructor and the `vtkm::filter::Filter::DoExecute()` method. The constructor is quite simple. It initializes the name of the output field name, which is managed by the superclass.

Example 2: Constructor for a simple filter.

```
1 VTKM_CONT PoundsPerSquareInchToNewtonsPerSquareMeterFilter::
2   PoundsPerSquareInchToNewtonsPerSquareMeterFilter()
3 {
4     this->SetOutputFieldName("");
5 }
```

In this case, we are setting the output field name to the empty string. This is not to mean that the default name of the output field should be the empty string, which is not a good idea. Rather, as we will see later, we will use the empty string to flag an output name that should be derived from the input name.

The meat of the filter implementation is located in the `vtkm::filter::Filter::DoExecute()` method.

Example 3: Implementation of `DoExecute` for a simple filter.

```
1 VTKM_CONT vtkm::cont::DataSet
2 PoundsPerSquareInchToNewtonsPerSquareMeterFilter::DoExecute(
3     const vtkm::cont::DataSet& inDataSet)
4 {
5     vtkm::cont::Field inField = this->GetFieldFromDataSet(inDataSet);
6
7     vtkm::cont::UnknownArrayHandle outArray;
8
9     auto resolveType = [&](const auto& inputArray) {
10         // use std::decay to remove const ref from the decltype of concrete.
11         using T = typename std::decay_t<decltype(inputArray)>::ValueType;
12         vtkm::cont::ArrayHandle<T> result;
13         this->Invoke(
14             PoundsPerSquareInchToNewtonsPerSquareMeterWorklet{}, inputArray, result);
```

(continues on next page)

(continued from previous page)

```

15     outArray = result;
16 };
17
18 this->CastAndCallScalarField(inField, resolveType);
19
20 std::string outFieldName = this->GetOutputFieldName();
21 if (outFieldName == "")
22 {
23     outFieldName = inField.GetName() + "_N/m^2";
24 }
25
26 return this->CreateResultField(
27     inDataSet, outFieldName, inField.GetAssociation(), outArray);
28 }

```

The single argument to `vtkm::filter::Filter::DoExecute()` is a `vtkm::cont::DataSet` containing the data to operate on, and `vtkm::filter::Filter::DoExecute()` returns a derived `vtkm::cont::DataSet`. The filter must pull the appropriate information out of the input `vtkm::cont::DataSet` to operate on. This simple algorithm just operates on a single field array of the data. The `vtkm::filter::Filter` base class provides several methods, documented in [Section 9.2.1 \(Input Fields\)](#), to allow filter users to select the active field to operate on. The filter implementation can get the appropriate field to operate on using the `vtkm::filter::Filter::GetFieldFromDataSet()` method as shown in [Example 3](#), line 5.

```

inline const vtkm::cont::Field &vtkm::filter::Filter::GetFieldFromDataSet(const vtkm::cont::DataSet
                                                                    &input) const

```

Retrieve an input field from a `vtkm::cont::DataSet` object.

When a filter operates on fields, it should use this method to get the input fields that the user has selected with `SetActiveField()` and related methods.

```

inline const vtkm::cont::Field &vtkm::filter::Filter::GetFieldFromDataSet(vtkm::IdComponent index,
                                                                    const vtkm::cont::DataSet
                                                                    &input) const

```

Retrieve an input field from a `vtkm::cont::DataSet` object.

When a filter operates on fields, it should use this method to get the input fields that the user has selected with `SetActiveField()` and related methods.

One of the challenges with writing filters is determining the actual types the algorithm is operating on. The `vtkm::cont::Field` object pulled from the input `vtkm::cont::DataSet` contains a `vtkm::cont::ArrayHandle` (see [Chapter 17 \(Basic Array Handles\)](#)), but you do not know what the template parameters of the `vtkm::cont::ArrayHandle` are. There are numerous ways to extract an array of an unknown type out of a `vtkm::cont::ArrayHandle` (many of which will be explored later in [Chapter 17 \(Basic Array Handles\)](#)), but the `vtkm::filter::Filter` contains some convenience functions to simplify this.

In particular, this filter operates specifically on scalar fields. For this purpose, `vtkm::filter::Filter` provides the `vtkm::filter::Filter::CastAndCallScalarField()` helper method. The first argument to `vtkm::filter::Filter::CastAndCallScalarField()` is the field containing the data to operate on. The second argument is a functor that will operate on the array once it is identified. `vtkm::filter::Filter::CastAndCallScalarField()` will pull a `vtkm::cont::ArrayHandle` out of the field and call the provided functor with that object. `vtkm::filter::Filter::CastAndCallScalarField()` is called in [Example 3](#), line 18.

```

template<typename Functor, typename ...Args>

```

```
inline void vtkm::filter::Filter::CastAndCallScalarField(const vtkm::cont::UnknownArrayHandle
                                                         &fieldArray, Functor &&functor, Args&&...
                                                         args) const
```

Convenience method to get the array from a filter's input scalar field.

A field filter typically gets its input fields using the internal `GetFieldFromDataSet`. To use this field in a worklet, it eventually needs to be converted to an `vtkm::cont::ArrayHandle`. If the input field is limited to be a scalar field, then this method provides a convenient way to determine the correct array type. Like other `CastAndCall` methods, it takes as input a `vtkm::cont::Field` (or `vtkm::cont::UnknownArrayHandle`) and a function/functor to call with the appropriate `vtkm::cont::ArrayHandle` type.

```
template<typename Functor, typename ...Args>
inline void vtkm::filter::Filter::CastAndCallScalarField(const vtkm::cont::Field &field, Functor
                                                         &&functor, Args&&... args) const
```

Convenience method to get the array from a filter's input scalar field.

A field filter typically gets its input fields using the internal `GetFieldFromDataSet`. To use this field in a worklet, it eventually needs to be converted to an `vtkm::cont::ArrayHandle`. If the input field is limited to be a scalar field, then this method provides a convenient way to determine the correct array type. Like other `CastAndCall` methods, it takes as input a `vtkm::cont::Field` (or `vtkm::cont::UnknownArrayHandle`) and a function/functor to call with the appropriate `vtkm::cont::ArrayHandle` type.

---

### Did You Know?

If your filter requires a field containing `vtkm::Vec` values of a particular size (e.g. 3), you can use the convenience method `vtkm::filter::Filter::CastAndCallVecField()`. `vtkm::filter::Filter::CastAndCallVecField()` works similarly to `vtkm::filter::Filter::CastAndCallScalarField()` except that it takes a template parameter specifying the size of the `vtkm::Vec`. For example, `vtkm::filter::Filter::CastAndCallVecField<3>(inField, functor);`.

---

As previously stated, one of the arguments to `vtkm::filter::Filter::CastAndCallScalarField()` is a functor that contains the routine to call with the found `vtkm::cont::ArrayHandle`. A functor can be created as its own class or struct, but a more convenient method is to use a C++ lambda. A lambda is an unnamed function defined inline with the code. The lambda in [Example 3](#) starts on [line 9](#). Apart from being more convenient than creating a named class, lambda functions offer another important feature. Lambda functions can “capture” variables in the current scope. They can therefore access things like local variables and the `this` reference to the method's class (even accessing private members).

The callback to the lambda function in [Example 3](#) first creates an output `vtkm::cont::ArrayHandle` of a compatible type ([line 12](#)), then invokes the worklet that computes the derived field ([line 13](#)), and finally captures the resulting array. Note that the `vtkm::filter::Filter` base class provides a `vtkm::filter::Filter::Invoke()` member that can be used to invoke the worklet. (See [Section 18.5 \(Invoking a Worklet\)](#) for information on invoking a worklet.) Recall that the worklet created in [Chapter 18 \(Simple Worklets\)](#) takes two parameters: an input array and an output array, which are shown in this invocation.

With the output data created, the filter has to build the output structure to return. All implementations of `vtkm::filter::Filter::DoExecute()` must return a `vtkm::cont::DataSet`, and for a simple field filter like this we want to return the same `vtkm::cont::DataSet` as the input with the output field added. The output field needs a name, and we get the appropriate name from the superclass ([Example 3, line 20](#)). However, we would like a special case where if the user does not specify an output field name we construct one based on the input field name. Recall from [Example 2](#) that by default we set the output field name to the empty string. Thus, our filter checks for this empty string, and if it is encountered, it builds a field name by appending “\_N/M^2” to it.

Finally, our filter constructs the output `vtkm::cont::DataSet` using one of the `vtkm::filter::Filter::CreateResult()` member functions ([Example 3, line 26](#)). In this particular case,

the filter uses `vtkm::filter::Filter::CreateResultField()`, which constructs a `vtkm::cont::DataSet` with the same structure as the input and adds the computed filter.

```
vtkm::cont::DataSet vtkm::filter::Filter::CreateResult(const vtkm::cont::DataSet &inDataSet) const
```

Create the output data set for DoExecute.

This form of `CreateResult` will create an output data set with the same cell structure and coordinate system as the input and pass all fields (as requested by the `Filter` state).

#### Parameters

**inDataSet** – [in] The input data set being modified (usually the one passed into `DoExecute`). The returned `DataSet` is filled with the cell set, coordinate system, and fields of `inDataSet` (as selected by the `FieldsToPass` state of the filter).

```
vtkm::cont::PartitionedDataSet vtkm::filter::Filter::CreateResult(const vtkm::cont::PartitionedDataSet
                                                                &input, const
                                                                vtkm::cont::PartitionedDataSet
                                                                &resultPartitions) const
```

Create the output data set for DoExecute.

This form of `CreateResult` will create an output `PartitionedDataSet` with the same partitions and pass all `PartitionedDataSet` fields (as requested by the `Filter` state).

#### Parameters

- **input** – [in] The input data set being modified (usually the one passed into `DoExecute`).
- **resultPartitions** – [in] The output data created by the filter. Fields from the input are passed onto the return result partition as requested by the `Filter` state.

```
template<typename FieldMapper>
```

```
inline vtkm::cont::PartitionedDataSet vtkm::filter::Filter::CreateResult(const
                                                                vtkm::cont::PartitionedDataSet
                                                                &input, const
                                                                vtkm::cont::PartitionedDataSet
                                                                &resultPartitions, FieldMapper
                                                                &&fieldMapper) const
```

Create the output data set for DoExecute.

This form of `CreateResult` will create an output `PartitionedDataSet` with the same partitions and pass all `PartitionedDataSet` fields (as requested by the `Filter` state).

#### Parameters

- **input** – [in] The input data set being modified (usually the one passed into `DoExecute`).
- **resultPartitions** – [in] The output data created by the filter. Fields from the input are passed onto the return result partition as requested by the `Filter` state.
- **fieldMapper** – [in] A function or functor that takes a `PartitionedDataSet` as its first argument and a `Field` as its second argument. The `PartitionedDataSet` is the data being created and will eventually be returned by `CreateResult`. The `Field` comes from `input`.

```
template<typename FieldMapper>
```

```
inline vtkm::cont::DataSet vtkm::filter::Filter::CreateResult(const vtkm::cont::DataSet &inDataSet,
                                                                const vtkm::cont::UnknownCellSet
                                                                &resultCellSet, FieldMapper
                                                                &&fieldMapper) const
```

Create the output data set for DoExecute.

This form of `CreateResult` will create an output data set with the given `CellSet`. You must also provide a field mapper function, which is a function that takes the output `DataSet` being created and a `Field` from the input and then applies any necessary transformations to the field array and adds it to the `DataSet`.

#### Parameters

- **inDataSet** – [in] The input data set being modified (usually the one passed into `DoExecute`). The returned `DataSet` is filled with fields of `inDataSet` (as selected by the `FieldsToPass` state of the filter).
- **resultCellSet** – [in] The `CellSet` of the output will be set to this.
- **fieldMapper** – [in] A function or functor that takes a `DataSet` as its first argument and a `Field` as its second argument. The `DataSet` is the data being created and will eventually be returned by `CreateResult`. The `Field` comes from `inDataSet`. The function should map the `Field` to match `resultCellSet` and then add the resulting field to the `DataSet`. If the mapping is not possible, then the function should do nothing.

```
vtkm::cont::DataSet vtkm::filter::Filter::CreateResultField(const vtkm::cont::DataSet &inDataSet,  
                                                         const vtkm::cont::Field &resultField) const
```

Create the output data set for `DoExecute`

This form of `CreateResult` will create an output data set with the same cell structure and coordinate system as the input and pass all fields (as requested by the `Filter` state). Additionally, it will add the provided field to the result.

#### Parameters

- **inDataSet** – [in] The input data set being modified (usually the one passed into `DoExecute`). The returned `DataSet` is filled with fields of `inDataSet` (as selected by the `FieldsToPass` state of the filter).
- **resultField** – [in] A `Field` that is added to the returned `DataSet`.

```
inline vtkm::cont::DataSet vtkm::filter::Filter::CreateResultField(const vtkm::cont::DataSet  
                                                                &inDataSet, const std::string  
                                                                &resultFieldName,  
                                                                vtkm::cont::Field::Association  
                                                                resultFieldAssociation, const  
                                                                vtkm::cont::UnknownArrayHandle  
                                                                &resultFieldArray) const
```

Create the output data set for `DoExecute`

This form of `CreateResult` will create an output data set with the same cell structure and coordinate system as the input and pass all fields (as requested by the `Filter` state). Additionally, it will add a field matching the provided specifications to the result.

#### Parameters

- **inDataSet** – [in] The input data set being modified (usually the one passed into `DoExecute`). The returned `DataSet` is filled with fields of `inDataSet` (as selected by the `FieldsToPass` state of the filter).
- **resultFieldName** – [in] The name of the field added to the returned `DataSet`.
- **resultFieldAssociation** – [in] The association of the field (e.g. point or cell) added to the returned `DataSet`.
- **resultFieldArray** – [in] An array containing the data for the field added to the returned `DataSet`.

```
inline vtkm::cont::DataSet vtkm::filter::Filter::CreateResultFieldPoint(const vtkm::cont::DataSet
                                                                    &inDataSet, const std::string
                                                                    &resultFieldName, const
                                                                    vtkm::cont::UnknownArrayHandle
                                                                    &resultFieldArray) const
```

Create the output data set for DoExecute

This form of `CreateResult` will create an output data set with the same cell structure and coordinate system as the input and pass all fields (as requested by the `Filter` state). Additionally, it will add a point field matching the provided specifications to the result.

#### Parameters

- **inDataSet** – [in] The input data set being modified (usually the one passed into `DoExecute`). The returned `DataSet` is filled with fields of `inDataSet` (as selected by the `FieldsToPass` state of the filter).
- **resultFieldName** – [in] The name of the field added to the returned `DataSet`.
- **resultFieldArray** – [in] An array containing the data for the field added to the returned `DataSet`.

```
inline vtkm::cont::DataSet vtkm::filter::Filter::CreateResultFieldCell(const vtkm::cont::DataSet
                                                                    &inDataSet, const std::string
                                                                    &resultFieldName, const
                                                                    vtkm::cont::UnknownArrayHandle
                                                                    &resultFieldArray) const
```

Create the output data set for DoExecute

This form of `CreateResult` will create an output data set with the same cell structure and coordinate system as the input and pass all fields (as requested by the `Filter` state). Additionally, it will add a cell field matching the provided specifications to the result.

#### Parameters

- **inDataSet** – [in] The input data set being modified (usually the one passed into `DoExecute`). The returned `DataSet` is filled with fields of `inDataSet` (as selected by the `FieldsToPass` state of the filter).
- **resultFieldName** – [in] The name of the field added to the returned `DataSet`.
- **resultFieldArray** – [in] An array containing the data for the field added to the returned `DataSet`.

```
template<typename FieldMapper>
inline vtkm::cont::DataSet vtkm::filter::Filter::CreateResultCoordinateSystem(const
                                                                    vtkm::cont::DataSet
                                                                    &inDataSet, const
                                                                    vtkm::cont::UnknownCellSet
                                                                    &resultCellSet, const
                                                                    vtkm::cont::CoordinateSystem
                                                                    &resultCoordSystem,
                                                                    FieldMapper
                                                                    &&fieldMapper)
                                                                    const
```

Create the output data set for DoExecute.

This form of `CreateResult` will create an output data set with the given `CellSet` and `CoordinateSystem`. You must also provide a field mapper function, which is a function that takes the output `DataSet` being created



and a `Field` from the input and then applies any necessary transformations to the field array and adds it to the `DataSet`.

#### Parameters

- **inDataSet** – [in] The input data set being modified (usually the one passed into `DoExecute`). The returned `DataSet` is filled with fields of `inDataSet` (as selected by the `FieldsToPass` state of the filter).
- **resultCellSet** – [in] The `CellSet` of the output will be set to this.
- **resultCoordSystem** – [in] This `CoordinateSystem` will be added to the output.
- **fieldMapper** – [in] A function or functor that takes a `DataSet` as its first argument and a `Field` as its second argument. The `DataSet` is the data being created and will eventually be returned by `CreateResult`. The `Field` comes from `inDataSet`. The function should map the `Field` to match `resultCellSet` and then add the resulting field to the `DataSet`. If the mapping is not possible, then the function should do nothing.

```
template<typename FieldMapper>
inline vtkm::cont::DataSet vtkm::filter::Filter::CreateResultCoordinateSystem(const
                                                                              vtkm::cont::DataSet
                                                                              &inDataSet, const
                                                                              vtkm::cont::UnknownCellSet
                                                                              &resultCellSet, const
                                                                              std::string
                                                                              &coordsName, const
                                                                              vtkm::cont::UnknownArrayHandle
                                                                              &coordsData,
                                                                              FieldMapper
                                                                              &&fieldMapper)
                                                                              const
```

Create the output data set for `DoExecute`.

This form of `CreateResult` will create an output data set with the given `CellSet` and `CoordinateSystem`. You must also provide a field mapper function, which is a function that takes the output `DataSet` being created and a `Field` from the input and then applies any necessary transformations to the field array and adds it to the `DataSet`.

#### Parameters

- **inDataSet** – [in] The input data set being modified (usually the one passed into `DoExecute`). The returned `DataSet` is filled with fields of `inDataSet` (as selected by the `FieldsToPass` state of the filter).
- **resultCellSet** – [in] The `CellSet` of the output will be set to this.
- **coordsName** – [in] The name of the coordinate system to be added to the output.
- **coordsData** – [in] The array containing the coordinates of the points.
- **fieldMapper** – [in] A function or functor that takes a `DataSet` as its first argument and a `Field` as its second argument. The `DataSet` is the data being created and will eventually be returned by `CreateResult`. The `Field` comes from `inDataSet`. The function should map the `Field` to match `resultCellSet` and then add the resulting field to the `DataSet`. If the mapping is not possible, then the function should do nothing.

---

#### Common Errors



The `vtkm::filter::Filter::CreateResult()` methods do more than just construct a new `vtkm::cont::DataSet`. They also set up the structure of the data and pass fields as specified by the state of the filter object. Thus, implementations of `vtkm::filter::Filter::DoExecute()` should always return a `vtkm::cont::DataSet` that is created with `vtkm::filter::Filter::CreateResult()` or a similarly named method in the base filter class.

---

This chapter has just provided a brief introduction to creating filters. There are several more filter superclasses to help express algorithms of different types. After some more worklet concepts to implement more complex algorithms are introduced in [Part IV \(Advanced Development\)](#), we will see a more complete documentation of the types of filters in [Chapter 23 \(Extended Filter Implementations\)](#).



# **Part IV**

## **Advanced Development**



## ADVANCED TYPES

Chapter 4 (Base Types) introduced some of the base data types defined for use in VTK-m. However, for simplicity Chapter 4 (Base Types) just briefly touched the high-level concepts of these types. In this chapter we dive into much greater depth and introduce several more types.

### 20.1 Single Number Types

As described in Chapter 4 (Base Types), VTK-m provides aliases for all the base C types to ensure the representation matches the variable use. When a specific type width is not required, then the most common types to use are `vtkm::FloatDefault` for floating-point numbers, `vtkm::Id` for array and similar indices, and `vtkm::IdComponent` for shorter-width vector indices.

If a specific type width is desired, then one of the following is used to clearly declare the type and width.

bytes	floating point	signed integer	unsigned integer
1		<code>vtkm::Int8</code>	<code>vtkm::UInt8</code>
2		<code>vtkm::Int16</code>	<code>vtkm::UInt16</code>
4	<code>vtkm::Float32</code>	<code>vtkm::Int32</code>	<code>vtkm::UInt32</code>
8	<code>vtkm::Float64</code>	<code>vtkm::Int64</code>	<code>vtkm::UInt64</code>

These VTK-m–defined types should be preferred over basic C types like `int` or `float`.

### 20.2 Vector Types

Visualization algorithms also often require operations on short vectors. Arrays indexed in up to three dimensions are common. Data are often defined in 2-space and 3-space, and transformations are typically done in homogeneous coordinates of length 4. To simplify these types of operations, VTK-m provides the `vtkm::Vec` templated type, which is essentially a fixed length array of a given type.

```
template<typename T, vtkm::IdComponent Size>
```

```
class Vec : public vtkm::detail::VecBase<T, Size, Vec<T, Size>>
```

A short fixed-length array.

The `Vec` templated class holds a short array of values of a size and type specified by the template arguments.

The `Vec` class is most often used to represent vectors in the mathematical sense as a quantity with a magnitude and direction. Vectors are, of course, used extensively in computational geometry as well as physical simulations.

The `Vec` class can be (and is) repurposed for more general usage of holding a fixed-length sequence of objects.

There is no real limit to the size of the sequence (other than the largest number representable by `vtkm::IdComponent`), but the `Vec` class is really designed for small sequences (seldom more than 10).

Subclassed by `vtkm::VecFlat< T, false >`

The default constructor of `vtkm::Vec` objects leaves the values uninitialized. All vectors have a constructor with one argument that is used to initialize all components. All `vtkm::Vec` objects also have a constructor that allows you to set the individual components (one per argument). All `vtkm::Vec` objects with a size that is greater than 4 are constructed at run time and support an arbitrary number of initial values. Likewise, there is a `vtkm::make_Vec()` convenience function that builds initialized vector types with an arbitrary number of components. Once created, you can use the bracket operator to get and set component values with the same syntax as an array.

Example 1: Creating vector types.

```

1  vtkm::Vec3f_32 A{ 1 }; // A is (1, 1, 1)
2  A[1] = 2; // A is now (1, 2, 1)
3  vtkm::Vec3f_32 B{ 1, 2, 3 }; // B is (1, 2, 3)
4  vtkm::Vec3f_32 C = vtkm::make_Vec(3, 4, 5); // C is (3, 4, 5)
5  // Longer Vecs specified with template.
6  vtkm::Vec<vtkm::Float32, 5> D{ 1 }; // D is (1, 1, 1, 1, 1)
7  vtkm::Vec<vtkm::Float32, 5> E{ 1, 2, 3, 4, 5 }; // E is (1, 2, 3, 4, 5)
8  vtkm::Vec<vtkm::Float32, 5> F = { 6, 7, 8, 9, 10 }; // F is (6, 7, 8, 9, 10)
9  auto G = vtkm::make_Vec(1, 3, 5, 7, 9); // G is (1, 3, 5, 7, 9)

```

```
template<typename T, typename ...Ts>
```

```
constexpr vtkm::Vec<T, vtkm::IdComponent(sizeof...(Ts) + 1)> vtkm::make_Vec(T value0, Ts&&... args)
```

Initializes and returns a `Vec` containing all the arguments.

The arguments should all be the same type or compile issues will occur.

The types `vtkm::Id2`, `vtkm::Id3`, and `vtkm::Id4` are type aliases of `vtkm::Vec<vtkm::Id, 2>`, `vtkm::Vec<vtkm::Id, 3>`, and `vtkm::Vec<vtkm::Id, 4>`, respectively. These are used to index arrays of 2, 3, and 4 dimensions, which is common. Likewise, `vtkm::IdComponent2`, `vtkm::IdComponent3`, and `vtkm::IdComponent4` are type aliases of `vtkm::Vec<vtkm::IdComponent, 2>`, `vtkm::Vec<vtkm::IdComponent, 3>`, and `vtkm::Vec<vtkm::IdComponent, 4>`, respectively.

Because declaring `vtkm::Vec` with all of its template parameters can be cumbersome, VTK-m provides easy to use aliases for small vectors of base types. As introduced in [Section 4.3 \(Vector Types\)](#), the following type aliases are available.

bytes	size	floating point	signed integer	unsigned integer
1	2		<code>vtkm::Vec2i_8</code>	<code>vtkm::Vec2ui_8</code>
	3		<code>vtkm::Vec3i_8</code>	<code>vtkm::Vec3ui_8</code>
	4		<code>vtkm::Vec4i_8</code>	<code>vtkm::Vec4ui_8</code>
2	2		<code>vtkm::Vec2i_16</code>	<code>vtkm::Vec2ui_16</code>
	3		<code>vtkm::Vec3i_16</code>	<code>vtkm::Vec3ui_16</code>
	4		<code>vtkm::Vec4i_16</code>	<code>vtkm::Vec4ui_16</code>
4	2	<code>vtkm::Vec2f_32</code>	<code>vtkm::Vec2i_32</code>	<code>vtkm::Vec2ui_32</code>
	3	<code>vtkm::Vec3f_32</code>	<code>vtkm::Vec3i_32</code>	<code>vtkm::Vec3ui_32</code>
	4	<code>vtkm::Vec4f_32</code>	<code>vtkm::Vec4i_32</code>	<code>vtkm::Vec4ui_32</code>
8	2	<code>vtkm::Vec2f_64</code>	<code>vtkm::Vec2i_64</code>	<code>vtkm::Vec2ui_64</code>
	3	<code>vtkm::Vec3f_64</code>	<code>vtkm::Vec3i_64</code>	<code>vtkm::Vec3ui_64</code>
	4	<code>vtkm::Vec4f_64</code>	<code>vtkm::Vec4i_64</code>	<code>vtkm::Vec4ui_64</code>

`vtkm::Vec` supports component-wise arithmetic using the operators for plus (+), minus (-), multiply (\*), and divide

(/). It also supports scalar to vector multiplication with the multiply operator. The comparison operators equal (==) is true if every pair of corresponding components are true and not equal (!=) is true otherwise. A special `vtkm::Dot()` function is overloaded to provide a dot product for every type of vector.

Example 2: Vector operations.

```

1  vtkm::Vec3f_32 A{ 1, 2, 3 };
2  vtkm::Vec3f_32 B{ 4, 5, 6.5 };
3  vtkm::Vec3f_32 C = A + B;           // C is (5, 7, 9.5)
4  vtkm::Vec3f_32 D = 2.0f * C;       // D is (10, 14, 19)
5  vtkm::Float32 s = vtkm::Dot(A, B); // s is 33.5
6  bool b1 = (A == B);               // b1 is false
7  bool b2 = (A == vtkm::make_Vec(1, 2, 3)); // b2 is true
8
9  vtkm::Vec<vtkm::Float32, 5> E{ 1, 2.5, 3, 4, 5 }; // E is (1, 2, 3, 4, 5)
10 vtkm::Vec<vtkm::Float32, 5> F{ 6, 7, 8.5, 9, 10.5 }; // F is (6, 7, 8, 9, 10)
11 vtkm::Vec<vtkm::Float32, 5> G = E + F;           // G is (7, 9.5, 11.5, 13, 15.5)
12 bool b3 = (E == F);                             // b3 is false
13 bool b4 = (G == vtkm::make_Vec(7.f, 9.5f, 11.5f, 13.f, 15.5f)); // b4 is true

```

These operators, of course, only work if they are also defined for the component type of the `vtkm::Vec`. For example, the multiply operator will work fine on objects of type `vtkm::Vec<char, 3>`, but the multiply operator will not work on objects of type `vtkm::Vec<std::string, 3>` because you cannot multiply objects of type `std::string`.

In addition to generalizing vector operations and making arbitrarily long vectors, `vtkm::Vec` can be repurposed for creating any sequence of homogeneous objects. Here is a simple example of using `vtkm::Vec` to hold the state of a polygon.

Example 3: Repurposing a `vtkm::Vec`.

```

1  vtkm::Vec<vtkm::Vec2f_32, 3> equilateralTriangle = { { 0.0f, 0.0f },
2                                                       { 1.0f, 0.0f },
3                                                       { 0.5f, 0.8660254f } };

```

## 20.2.1 Vec-like Types

The `vtkm::Vec` class provides a convenient structure for holding and passing small vectors of data. However, there are times when using `vtkm::Vec` is inconvenient or inappropriate. For example, the size of `vtkm::Vec` must be known at compile time, but there may be need for a vector whose size is unknown until compile time. Also, the data populating a `vtkm::Vec` might come from a source that makes it inconvenient or less efficient to construct a `vtkm::Vec`. For this reason, VTK-m also provides several Vec-like objects that behave much like `vtkm::Vec` but are a different class. These Vec-like objects have the same interface as `vtkm::Vec` except that the `NUM_COMPONENTS` constant is not available on those that are sized at run time. Vec-like objects also come with a `CopyInto` method that will take their contents and copy them into a standard `vtkm::Vec` class. (The standard `vtkm::Vec` class also has a `vtkm::Vec::CopyInto()` method for consistency.)

## C-Array Vec Wrapper

The first Vec-like object is `vtkm::VecC`, which exposes a C-type array as a `vtkm::Vec`.

```
template<typename T>
```

```
class VecC : public vtkm::detail::VecCBase<T, VecC<T>>>
```

A Vec-like representation for short arrays.

The `VecC` class takes a short array of values and provides an interface that mimics `Vec`. This provides a mechanism to treat C arrays like a `Vec`. It is useful in situations where you want to use a `Vec` but the data must come from elsewhere or in certain situations where the size cannot be determined at compile time. In particular, `Vec` objects of different sizes can potentially all be converted to a `VecC` of the same type.

Note that `VecC` holds a reference to an outside array given to it. If that array gets destroyed (for example because the source goes out of scope), the behavior becomes undefined.

You cannot use `VecC` with a const type in its template argument. For example, you cannot declare `VecC<const vtkm::Id>`. If you want a non-mutable `VecC`, the `VecCConst` class (e.g. `VecCConst<vtkm::Id>`).

The constructor for `vtkm::VecC` takes a C array and a size of that array. There is also a constant version of `vtkm::VecC` named `vtkm::VecCConst`, which takes a constant array and cannot be mutated.

```
template<typename T>
```

```
class VecCConst : public vtkm::detail::VecCBase<T, VecCConst<T>>>
```

A const version of `VecC`.

`VecCConst` is a non-mutable form of `VecC`. It can be used in place of `VecC` when a constant array is available.

A `VecC` can be automatically converted to a `VecCConst`, but not vice versa, so function arguments should use `VecCConst` when the data do not need to be changed.

The `vtkm/Types.h` header defines both `vtkm::VecC` and `vtkm::VecCConst` as well as multiple versions of `vtkm::make_VecC()` to easily convert a C array to either a `vtkm::VecC` or `vtkm::VecCConst`.

```
template<typename T>
```

```
static inline vtkm::VecC<T> vtkm::make_VecC(T *array, vtkm::IdComponent size)
```

Creates a `VecC` from an input array.

```
template<typename T>
```

```
static inline vtkm::VecCConst<T> vtkm::make_VecC(const T *array, vtkm::IdComponent size)
```

Creates a `VecCConst` from a constant input array.

The following example demonstrates converting values from a constant table into a `vtkm::VecCConst` for further consumption. The table and associated methods define how 8 points come together to form a hexahedron.

Example 4: Using `vtkm::VecCConst` with a constant array.

```
1 VTKM_EXEC vtkm::VecCConst<vtkm::IdComponent> HexagonIndexToIJK(vtkm::IdComponent index)
2 {
3     static const vtkm::IdComponent HexagonIndexToIJKTable[8][3] = {
4         { 0, 0, 0 }, { 1, 0, 0 }, { 1, 1, 0 }, { 0, 1, 0 },
5         { 0, 0, 1 }, { 1, 0, 1 }, { 1, 1, 1 }, { 0, 1, 1 }
6     };
7
8     return vtkm::make_VecC(HexagonIndexToIJKTable[index], 3);
9 }
10
```

(continues on next page)



(continued from previous page)

```

11 VTKM_EXEC vtkm::IdComponent HexagonIJKToIndex(vtkm::VecCConst<vtkm::IdComponent> ijk)
12 {
13     static const vtkm::IdComponent HexagonIJKToIndexTable[2][2][2] = { {
14                                     // i=0
15                                     { 0, 4 }, // j=0
16                                     { 3, 7 }, // j=1
17                                     },
18                                     {
19                                     // i=1
20                                     { 1, 5 }, // j=0
21                                     { 2, 6 }, // j=1
22                                     } }];
23
24     return HexagonIJKToIndexTable[ijk[0]][ijk[1]][ijk[2]];
25 }

```

### Common Errors

The `vtkm::VecC` and `vtkm::VecCConst` classes only hold a pointer to a buffer that contains the data. They do not manage the memory holding the data. Thus, if the pointer given to `vtkm::VecC` or `vtkm::VecCConst` becomes invalid, then using the object becomes invalid. Make sure that the scope of the `vtkm::VecC` or `vtkm::VecCConst` does not outlive the scope of the data it points to.

### Variable-Sized Vec

The next Vec-like object is `vtkm::VecVariable`, which provides a Vec-like object that can be resized at run time to a maximum value. Unlike `vtkm::VecC`, `vtkm::VecVariable` holds its own memory, which makes it a bit safer to use. But also unlike `vtkm::VecC`, you must define the maximum size of `vtkm::VecVariable` at compile time. Thus, `vtkm::VecVariable` is really only appropriate to use when there is a predetermined limit to the vector size that is fairly small.

```
template<typename T, vtkm::IdComponent MaxSize>
```

```
class VecVariable
```

A short variable-length array with maximum length.

The `VecVariable` class is a Vec-like class that holds a short array of some maximum length. To avoid dynamic allocations, the maximum length is specified at compile time. Internally, `VecVariable` holds a `Vec` of the maximum length and exposes a subsection of it.

The following example uses a `vtkm::VecVariable` to store the trace of edges within a hexahedron. This example uses the methods defined in [Example 5](#).

Example 5: Using `vtkm::VecVariable`.

```

1 vtkm::VecVariable<vtkm::IdComponent, 4> HexagonShortestPath(vtkm::IdComponent startPoint,
2                                                             vtkm::IdComponent endPoint)
3 {
4     vtkm::VecCConst<vtkm::IdComponent> startIJK = HexagonIndexToIJK(startPoint);
5     vtkm::VecCConst<vtkm::IdComponent> endIJK = HexagonIndexToIJK(endPoint);
6
7     vtkm::IdComponent3 currentIJK;

```

(continues on next page)

(continued from previous page)

```

8   startIJK.CopyTo(currentIJK);
9
10  vtkm::VecVariable<vtkm::IdComponent, 4> path;
11  path.Append(startPoint);
12  for (vtkm::IdComponent dimension = 0; dimension < 3; dimension++)
13  {
14      if (currentIJK[dimension] != endIJK[dimension])
15      {
16          currentIJK[dimension] = endIJK[dimension];
17          path.Append(HexagonIJKToIndex(currentIJK));
18      }
19  }
20
21  return path;
22  }

```

## Vecs from Portals

VTK-m provides further examples of Vec-like objects as well. For example, the `vtkm::VecFromPortal` and `vtkm::VecFromPortalPermute` objects allow you to treat a subsection of an arbitrarily large array as a `vtkm::Vec`. These objects work by attaching to array portals, which are described in Section~ref{sec:ArrayPortals}.

template<typename **PortalType**>

class **VecFromPortal**

A short variable-length array from a window in an ArrayPortal.

The `VecFromPortal` class is a Vec-like class that holds an array portal and exposes a small window of that portal as if it were a `Vec`.

template<typename **IndexVecType**, typename **PortalType**>

class **VecFromPortalPermute**

A short vector from an ArrayPortal and a vector of indices.

The `VecFromPortalPermute` class is a Vec-like class that holds an array portal and a second Vec-like containing indices into the array. Each value of this vector is the value from the array with the respective index.

## Point Coordinate Vec

Another example of a Vec-like object is `vtkm::VecRectilinearPointCoordinates`, which efficiently represents the point coordinates in an axis-aligned hexahedron. Such shapes are common in structured grids. These and other data sets are described in Chapter 7 (Data Sets).

## 20.3 Range

VTK-m provides a convenience structure named `vtkm::Range` to help manage a range of values. The `vtkm::Range` struct contains two data members, `vtkm::Range::Min` and `vtkm::Range::Max`, which represent the ends of the range of numbers. `vtkm::Range::Min` and `vtkm::Range::Max` are both of type `vtkm::Float64`. `vtkm::Range::Min` and `vtkm::Range::Max` can be directly accessed, but `vtkm::Range` also comes with several helper functions to make it easier to build and use ranges. Note that all of these functions treat the minimum and maximum value as inclusive to the range.

struct **Range**

Represent a continuous scalar range of values.

`vtkm::Range` is a helper class for representing a range of floating point values from a minimum value to a maximum value. This is specified simply enough with a `Min` and `Max` value.

`Range` also contains several helper functions for computing and maintaining the range.

### Public Functions

inline **Range**()

Construct a range with a given minimum and maximum.

If no minimum or maximum is given, the range will be empty.

inline bool **IsEmpty**() const

**Determine** if the range is valid (i.e.

has at least one valid point).

`IsEmpty` return true if the range contains some valid values between `Min` and `Max`. If `Max` is less than `Min`, then no values satisfy the range and `IsEmpty` returns false. Otherwise, return true.

`IsEmpty` assumes `Min` and `Max` are inclusive. That is, if they are equal then true is returned.

template<typename T>

inline bool **Contains**(const T &value) const

**Determines** if a value is within the range.

`Contains` returns true if the give value is within the range, false otherwise. `Contains` treats the min and max as inclusive. That is, if the value is exactly the min or max, true is returned.

inline vtkm::Float64 **Length**() const

**Returns** the length of the range.

`Length` computes the distance between the min and max. If the range is empty, 0 is returned.

inline vtkm::Float64 **Center**() const

**Returns** the center of the range.

`Center` computes the middle value of the range. If the range is empty, NaN is returned.

template<typename T>

inline void **Include**(const T &value)

**Expand** range to include a value.

This version of `Include` expands the range just enough to include the given value. If the range already includes this value, then nothing is done.

inline void **Include**(const vtkm::Range &range)

**Expand** range to include other range.

This version of `Include` expands this range just enough to include that of another range. Essentially it is the union of the two ranges.

inline vtkm::Range **Union**(const vtkm::Range &otherRange) const

**Return** the union of this and another range.

This is a nondestructive form of `Include`.

inline vtkm::Range **Intersection**(const vtkm::Range &otherRange) const

**Return** the intersection of this and another range.

inline vtkm::Range **operator+**(const vtkm::Range &otherRange) const

**Operator** for union

## Public Members

vtkm::Float64 **Min**

The minimum value of the range (inclusive).

vtkm::Float64 **Max**

The maximum value of the range (inclusive).

The following example demonstrates the operation of `vtkm::Range`.

Example 6: Using `vtkm::Range`.

```

1  vtkm::Range range;           // default constructor is empty range
2  bool b1 = range.IsEmpty();  // b1 is false
3
4  range.Include(0.5);          // range now is [0.5 .. 0.5]
5  bool b2 = range.IsEmpty();  // b2 is true
6  bool b3 = range.Contains(0.5); // b3 is true
7  bool b4 = range.Contains(0.6); // b4 is false
8
9  range.Include(2.0);          // range is now [0.5 .. 2]
10 bool b5 = range.Contains(0.5); // b3 is true
11 bool b6 = range.Contains(0.6); // b4 is true
12
13 range.Include(vtkm::Range(-1, 1)); // range is now [-1 .. 2]
14
15 range.Include(vtkm::Range(3, 4)); // range is now [-1 .. 4]
16
17 vtkm::Float64 lower = range.Min; // lower is -1
18 vtkm::Float64 upper = range.Max; // upper is 4
19 vtkm::Float64 length = range.Length(); // length is 5
20 vtkm::Float64 center = range.Center(); // center is 1.5

```

## 20.4 Bounds

VTK-m provides a convenience structure named `vtkm::Bounds` to help manage an axis-aligned region in 3D space. Among other things, this structure is often useful for representing a bounding box for geometry. The `vtkm::Bounds` struct contains three data members, `vtkm::Bounds::X`, `vtkm::Bounds::Y`, and `vtkm::Bounds::Z`, which represent the range of the bounds along each respective axis. All three of these members are of type `vtkm::Range`, which is discussed previously in [Section 20.3 \(Range\)](#). `vtkm::Bounds::X`, `vtkm::Bounds::Y`, and `vtkm::Bounds::Z` can be directly accessed, but `vtkm::Bounds` also comes with the following helper functions to make it easier to build and use ranges.

struct **Bounds**

Represent an axis-aligned 3D bounds in space.

`vtkm::Bounds` is a helper class for representing the axis-aligned box representing some region in space. The typical use of this class is to express the containing box of some geometry. The box is specified as ranges in the x, y, and z directions.

`Bounds` also contains several helper functions for computing and maintaining the bounds.

### Public Functions

inline **Bounds**()

Construct an empty bounds.

The bounds will represent no space until otherwise modified.

inline **Bounds**(const vtkm::Range &xRange, const vtkm::Range &yRange, const vtkm::Range &zRange)

Construct a bounds with a given range in the x, y, and z dimensions.

template<typename **T1**, typename **T2**, typename **T3**, typename **T4**, typename **T5**, typename **T6**>

inline **Bounds**(const *T1* &minX, const *T2* &maxX, const *T3* &minY, const *T4* &maxY, const *T5* &minZ, const *T6* &maxZ)

Construct a bounds with the minimum and maximum coordinates in the x, y, and z directions.

template<typename **T**>

inline explicit **Bounds**(const *T* bounds[6])

Initialize bounds with an array of 6 values in the order xmin, xmax, ymin, ymax, zmin, zmax.

template<typename **T**>

inline **Bounds**(const vtkm::Vec<*T*, 3> &minPoint, const vtkm::Vec<*T*, 3> &maxPoint)

Initialize bounds with the minimum corner point and the maximum corner point.

inline bool **IsNonEmpty**() const

**Determine** if the bounds are valid (i.e.

has at least one valid point).

`IsNonEmpty` returns true if the bounds contain some valid points. If the bounds are any real region, even if a single point or it expands to infinity, true is returned.

template<typename **T**>

inline bool **Contains**(const vtkm::Vec<*T*, 3> &point) const

**Determines** if a point coordinate is within the bounds.

inline vtkm::Float64 **Volume**() const

**Returns** the volume of the bounds.

**Volume** computes the product of the lengths of the ranges in each dimension. If the bounds are empty, 0 is returned.

inline vtkm::Float64 **Area**() const

**Returns** the area of the bounds in the X-Y-plane.

**Area** computes the product of the lengths of the ranges in dimensions X and Y. If the bounds are empty, 0 is returned.

inline vtkm::Vec3f\_64 **Center**() const

**Returns** the center of the range.

**Center** computes the point at the middle of the bounds. If the bounds are empty, the results are undefined.

inline vtkm::Vec3f\_64 **MinCorner**() const

**Returns** the min point of the bounds

**MinCorder** returns the minium point of the bounds.If the bounds are empty, the results are undefined.

inline vtkm::Vec3f\_64 **MaxCorner**() const

**Returns** the max point of the bounds

**MaxCorder** returns the minium point of the bounds.If the bounds are empty, the results are undefined.

template<typename T>

inline void **Include**(const vtkm::Vec<T, 3> &point)

**Expand** bounds to include a point.

This version of **Include** expands the bounds just enough to include the given point coordinates. If the bounds already include this point, then nothing is done.

inline void **Include**(const vtkm::Bounds &bounds)

**Expand** bounds to include other bounds.

This version of **Include** expands these bounds just enough to include that of another bounds. Essentially it is the union of the two bounds.

inline vtkm::Bounds **Union**(const vtkm::Bounds &otherBounds) const

**Return** the union of this and another bounds.

This is a nondestructive form of **Include**.

inline vtkm::Bounds **Intersection**(const vtkm::Bounds &otherBounds) const

**Return** the intersection of this and another range.

inline vtkm::Bounds **operator+**(const vtkm::Bounds &otherBounds) const

**Operator** for union

## Public Members

### `vtkm::Range X`

The range of values in the X direction.

The `vtkm::Range` struct provides the minimum and maximum along that axis.

### `vtkm::Range Y`

The range of values in the Y direction.

The `vtkm::Range` struct provides the minimum and maximum along that axis.

### `vtkm::Range Z`

The range of values in the Z direction.

The `vtkm::Range` struct provides the minimum and maximum along that axis.

The following example demonstrates the operation of `vtkm::Bounds`.

Example 7: Using `vtkm::Bounds`.

```

1  vtkm::Bounds bounds;           // default constructor makes empty
2  bool b1 = bounds.IsNonEmpty(); // b1 is false
3
4  bounds.Include(vtkm::make_Vec(0.5, 2.0, 0.0)); // bounds contains only
5                                                  // the point [0.5, 2, 0]
6  bool b2 = bounds.IsNonEmpty(); // b2 is true
7  bool b3 = bounds.Contains(vtkm::make_Vec(0.5, 2.0, 0.0)); // b3 is true
8  bool b4 = bounds.Contains(vtkm::make_Vec(1, 1, 1)); // b4 is false
9  bool b5 = bounds.Contains(vtkm::make_Vec(0, 0, 0)); // b5 is false
10
11 bounds.Include(vtkm::make_Vec(4, -1, 2)); // bounds is region [0.5 .. 4] in X,
12                                           // [-1 .. 2] in Y,
13                                           // and [0 .. 2] in Z
14 bool b6 = bounds.Contains(vtkm::make_Vec(0.5, 2.0, 0.0)); // b6 is true
15 bool b7 = bounds.Contains(vtkm::make_Vec(1, 1, 1)); // b7 is true
16 bool b8 = bounds.Contains(vtkm::make_Vec(0, 0, 0)); // b8 is false
17
18 vtkm::Bounds otherBounds(vtkm::make_Vec(0, 0, 0), vtkm::make_Vec(3, 3, 3));
19 // otherBounds is region [0 .. 3] in X, Y, and Z
20 bounds.Include(otherBounds); // bounds is now region [0 .. 4] in X,
21                               // [-1 .. 3] in Y,
22                               // and [0 .. 3] in Z
23
24 vtkm::Vec3f_64 lower(bounds.X.Min, bounds.Y.Min, bounds.Z.Min);
25 // lower is [0, -1, 0]
26 vtkm::Vec3f_64 upper(bounds.X.Max, bounds.Y.Max, bounds.Z.Max);
27 // upper is [4, 3, 3]
28
29 vtkm::Vec3f_64 center = bounds.Center(); // center is [2, 1, 1.5]

```

## 20.5 Index Ranges

Just as it is sometimes necessary to track a range of real values, there are times when code has to specify a continuous range of values in an index sequence like an array. For this purpose, VTK-m provides `RangeId`, which behaves similarly to `Range` except for integer values.

struct **RangeId**

Represent a range of *vtkm::Id* values.

*vtkm::RangeId* is a helper class for representing a range of *vtkm::Id* values. This is specified simply with a `Min` and `Max` value, where `Max` is exclusive.

*RangeId* also contains several helper functions for computing and maintaining the range.

### Public Functions

inline **RangeId**()

Construct a range with no indices.

inline **RangeId**(*vtkm::Id* min, *vtkm::Id* max)

Construct a range with the given minimum (inclusive) and maximum (exclusive) indices.

inline bool **IsEmpty**() const

**Determine** if the range is valid.

`IsEmpty` return true if the range contains some valid values between `Min` and `Max`. If `Max <= Min`, then no values satisfy the range and `IsEmpty` returns false. Otherwise, return true.

inline bool **Contains**(*vtkm::Id* value) const

**Determines** if a value is within the range.

`Contains` returns true if the give value is within the range, false otherwise.

inline *vtkm::Id* **Length**() const

**Returns** the length of the range.

`Length` computes the distance between the min and max. If the range is empty, 0 is returned.

inline *vtkm::Id* **Center**() const

**Returns** the center of the range.

`Center` computes the middle value of the range.

inline void **Include**(*vtkm::Id* value)

**Expand** range to include a value.

This version of `Include` expands the range just enough to include the given value. If the range already includes this value, then nothing is done.

inline void **Include**(const *vtkm::RangeId* &range)

**Expand** range to include other range.

This version of `Include` expands this range just enough to include that of another range. Essentially it is the union of the two ranges.



```
inline vtkm::RangeId Union(const vtkm::RangeId &other) const
```

**Return** the union of this and another range.

This is a nondestructive form of `Include`.

```
inline vtkm::RangeId operator+(const vtkm::RangeId &other) const
```

**Operator** for union

## Public Members

```
vtkm::Id Min
```

The minimum index of the range (inclusive).

```
vtkm::Id Max
```

The maximum index of the range (exclusive).

VTK-m also often must operate on 2D and 3D arrays (particularly for structured cell sets). For these use cases, `RangeId2` and `RangeId3` are provided.

```
struct RangeId2
```

Represent 2D integer range.

`vtkm::RangeId2` is a helper class for representing a 2D range of integer values. The typical use of this class is to express a box of indices in the x and y directions.

`RangeId2` also contains several helper functions for computing and maintaining the range.

## Public Functions

```
RangeId2() = default
```

Construct an empty 2D range.

```
inline RangeId2(const vtkm::RangeId &xrange, const vtkm::RangeId &yrange)
```

Construct a range with the given x and y directions.

```
inline RangeId2(vtkm::Id minX, vtkm::Id maxX, vtkm::Id minY, vtkm::Id maxY)
```

Construct a range with the given minimum (inclusive) and maximum (exclusive) points.

```
inline explicit RangeId2(const vtkm::Id range[4])
```

Initialize range with an array of 4 values in the order xmin, xmax, ymin, ymax.

```
inline RangeId2(const vtkm::Id2 &min, const vtkm::Id2 &max)
```

Initialize range with the minimum and the maximum corners.

```
inline bool IsNonEmpty() const
```

**Determine** if the range is non-empty.

`IsNonEmpty` returns true if the range is non-empty.

```
inline bool Contains(const vtkm::Id2 &val) const
```

**Determines** if an `Id2` value is within the range.

inline vtkm::Id2 **Center**() const

**Returns** the center of the range.

Center computes the middle of the range.

template<typename T>

inline void **Include**(const vtkm::Vec<T, 2> &point)

**Expand** range to include a value.

This version of **Include** expands the range just enough to include the given value. If the range already include this value, then nothing is done.

inline void **Include**(const vtkm::RangeId2 &range)

**Expand** range to include other range.

This version of **Include** expands the range just enough to include the other range. Essentially it is the union of the two ranges.

inline vtkm::RangeId2 **Union**(const vtkm::RangeId2 &other) const

**Return** the union of this and another range.

This is a nondestructive form of **Include**.

inline vtkm::RangeId2 **operator+**(const vtkm::RangeId2 &other) const

**Operator** for union

## Public Members

vtkm::RangeId **X**

The range of values in the X direction.

The `vtkm::RangeId` struct provides the minimum and maximum along that axis.

vtkm::RangeId **Y**

The range of values in the Y direction.

The `vtkm::RangeId` struct provides the minimum and maximum along that axis.

struct **RangeId3**

Represent 3D integer range.

`vtkm::RangeId3` is a helper class for representing a 3D range of integer values. The typical use of this class is to express a box of indices in the x, y, and z directions.

`RangeId3` also contains several helper functions for computing and maintaining the range.

## Public Functions

**RangeId3()** = default

Construct an empty 3D range.

inline **RangeId3**(const vtkm::RangeId &xrange, const vtkm::RangeId &yrange, const vtkm::RangeId &zrange)

Construct a range with the given x, y, and z directions.

inline **RangeId3**(vtkm::Id minX, vtkm::Id maxX, vtkm::Id minY, vtkm::Id maxY, vtkm::Id minZ, vtkm::Id maxZ)

Construct a range with the given minimum (inclusive) and maximum (exclusive) points.

inline explicit **RangeId3**(const vtkm::Id range[6])

Initialize range with an array of 6 values in the order xmin, xmax, ymin, ymax, zmin, zmax.

inline **RangeId3**(const vtkm::Id3 &min, const vtkm::Id3 &max)

Initialize range with the minimum and the maximum corners.

inline bool **IsEmpty**() const

**Determine** if the range is non-empty.

IsEmpty returns true if the range is non-empty.

inline bool **Contains**(const vtkm::Id3 &val) const

**Determines** if an Id3 value is within the range.

inline vtkm::Id3 **Center**() const

**Returns** the center of the range.

Center computes the middle of the range.

template<typename T>

inline void **Include**(const vtkm::Vec<T, 3> &point)

**Expand** range to include a value.

This version of Include expands the range just enough to include the given value. If the range already include this value, then nothing is done.

inline void **Include**(const vtkm::RangeId3 &range)

**Expand** range to include other range.

This version of Include expands the range just enough to include the other range. Essentially it is the union of the two ranges.

inline vtkm::RangeId3 **Union**(const vtkm::RangeId3 &other) const

**Return** the union of this and another range.

This is a nondestructive form of Include.

inline vtkm::RangeId3 **operator+**(const vtkm::RangeId3 &other) const

**Operator** for union

## Public Members

`vtkm::RangeId X`

The range of values in the X direction.

The `vtkm::RangeId` struct provides the minimum and maximum along that axis.

`vtkm::RangeId Y`

The range of values in the Y direction.

The `vtkm::RangeId` struct provides the minimum and maximum along that axis.

`vtkm::RangeId Z`

The range of values in the Z direction.

The `vtkm::RangeId` struct provides the minimum and maximum along that axis.

## 20.6 Traits

When using templated types, it is often necessary to get information about the type or specialize code based on general properties of the type. VTK-m uses *traits* classes to publish and retrieve information about types. A traits class is simply a templated structure that provides type aliases for tag structures, empty types used for identification. The traits classes might also contain constant numbers and helpful static functions. See *Effective C++ Third Edition* by Scott Meyers for a description of traits classes and their uses.

### 20.6.1 Type Traits

The `vtkm::TypeTraits` templated class provides basic information about a core type. These type traits are available for all the basic C++ types as well as the core VTK-m types described in [Chapter 4 \(Base Types\)](#). `vtkm::TypeTraits` contains the following elements.

```
template<typename T>
```

```
class TypeTraits
```

The `TypeTraits` class provides helpful compile-time information about the basic types used in VTKm (and a few others for convenience).

The majority of `TypeTraits` contents are typedefs to tags that can be used to easily override behavior of called functions.

Subclassed by `vtkm::TypeTraits< const T >`

#### Public Types

```
using NumericTag = vtkm::TypeTraitsUnknownTag
```

A tag to determine whether the type is integer or real.

This tag is either `TypeTraitsRealTag` or `TypeTraitsIntegerTag`.

using **DimensionalityTag** = vtkm::TypeTraitsUnknownTag

A tag to determine whether the type has multiple components.

This tag is either *TypeTraitsScalarTag* or *TypeTraitsVectorTag*. Scalars can also be treated as vectors with *VecTraits*.

## Public Static Functions

static inline *T* **ZeroInitialization**()

A static function that returns 0 (or the closest equivalent to it) for the given type.

The *vtkm::TypeTraits::NumericTag* will be an alias for one of the following tags.

struct **TypeTraitsRealTag**

Tag used to identify types that store real (floating-point) numbers.

A *TypeTraits* class will typedef this class to *NumericTag* if it stores real numbers (or vectors of real numbers).

struct **TypeTraitsIntegerTag**

Tag used to identify types that store integer numbers.

A *TypeTraits* class will typedef this class to *NumericTag* if it stores integer numbers (or vectors of integers).

The *vtkm::TypeTraits::DimensionalityTag* will be an alias for one of the following tags.

struct **TypeTraitsScalarTag**

Tag used to identify 0 dimensional types (scalars).

Scalars can also be treated like vectors when used with *VecTraits*. A *TypeTraits* class will typedef this class to *DimensionalityTag*.

struct **TypeTraitsVectorTag**

Tag used to identify 1 dimensional types (vectors).

A *TypeTraits* class will typedef this class to *DimensionalityTag*.

If for some reason one of these tags do not apply, *vtkm::TypeTraitsUnknownTag* will be used.

struct **TypeTraitsUnknownTag**

Tag used to identify types that aren't Real, Integer, Scalar or Vector.

The definition of *vtkm::TypeTraits* for *vtkm::Float32* could like something like this.

Example 8: Example definition of  
*vtkm::TypeTraits<vtkm::Float32>*.

```

1 namespace vtkm {
2
3 template<>
4 struct TypeTraits<vtkm::Float32>
5 {
6     using NumericTag = vtkm::TypeTraitsRealTag;
7     using DimensionalityTag = vtkm::TypeTraitsScalarTag;

```

(continues on next page)

(continued from previous page)

```

8
9   VTKM_EXEC_CONT
10  static vtkm::Float32 ZeroInitialization() { return vtkm::Float32(0); }
11 };
12
13 }

```

Here is a simple example of using `vtkm::TypeTraits` to implement a generic function that behaves like the remainder operator (%) for all types including floating points and vectors.

Example 9: Using `vtkm::TypeTraits` for a generic remainder.

```

1  #include <vtkm/TypeTraits.h>
2
3  #include <vtkm/Math.h>
4
5  template<typename T>
6  T AnyRemainder(const T& numerator, const T& denominator);
7
8  namespace detail
9  {
10
11  template<typename T>
12  T AnyRemainderImpl(const T& numerator,
13                    const T& denominator,
14                    vtkm::TypeTraitsIntegerTag,
15                    vtkm::TypeTraitsScalarTag)
16  {
17      return numerator % denominator;
18  }
19
20  template<typename T>
21  T AnyRemainderImpl(const T& numerator,
22                    const T& denominator,
23                    vtkm::TypeTraitsRealTag,
24                    vtkm::TypeTraitsScalarTag)
25  {
26      // The VTK-m math library contains a Remainder function that operates on
27      // floating point numbers.
28      return vtkm::Remainder(numerator, denominator);
29  }
30
31  template<typename T, typename NumericTag>
32  T AnyRemainderImpl(const T& numerator,
33                    const T& denominator,
34                    NumericTag,
35                    vtkm::TypeTraitsVectorTag)
36  {
37      T result;
38      for (int componentIndex = 0; componentIndex < T::NUM_COMPONENTS; componentIndex++)
39      {
40          result[componentIndex] =

```

(continues on next page)

(continued from previous page)

```

41     AnyRemainder(numerator[componentIndex], denominator[componentIndex]);
42 }
43 return result;
44 }
45
46 } // namespace detail
47
48 template<typename T>
49 T AnyRemainder(const T& numerator, const T& denominator)
50 {
51     return detail::AnyRemainderImpl(numerator,
52                                     denominator,
53                                     typename vtkm::TypeTraits<T>::NumericTag(),
54                                     typename vtkm::TypeTraits<T>::DimensionalityTag());
55 }

```

## 20.6.2 Vector Traits

The templated `vtkm::Vec` class contains several items for introspection (such as the component type and its size). However, there are other types that behave similarly to `vtkm::Vec` objects but have different ways to perform this introspection.

For example, VTK-m contains `Vec`-like objects that essentially behave the same but might have different features. Also, there may be reason to interchangeably use basic scalar values, like an integer or floating point number, with vectors. To provide a consistent interface to access these multiple types that represents vectors, the `vtkm::VecTraits` templated class provides information and accessors to vector types. It contains the following elements.

```
template<class T>
```

```
struct VecTraits
```

Traits that can be queried to treat any type as a `Vec`.

The `VecTraits` class gives several static members that define how to use a given type as a vector. This is useful for templated functions and methods that have a parameter that could be either a standard scalar type or a `Vec` or some other `Vec`-like object. When using this class, scalar objects are treated like a `Vec` of size 1.

The default implementation of this template treats the type as a scalar. Types that actually behave like vectors should specialize this template to provide the proper information.

Subclassed by `vtkm::VecTraits< T & >`, `vtkm::VecTraits< T * >`, `vtkm::VecTraits< const T & >`, `vtkm::VecTraits< const T >`, `vtkm::internal::SafeVecTraits< T >`

### Public Types

```
using ComponentType = T
```

Type of the components in the vector.

If the type is really a scalar, then the component type is the same as the scalar type.

```
using BaseComponentType = T
```

Base component type in the vector.

Similar to `ComponentType` except that for nested vectors (e.g. `Vec<Vec<T, M>, N>`), it returns the base scalar type at the end of the composition (T in this example).

using **HasMultipleComponents** = `vtkm::VecTraitsTagSingleComponent`

A tag specifying whether this vector has multiple components (i.e.

is a “real” vector).

This type is set to either `vtkm::VecTraitsTagSingleComponent` if the vector length is size 1 or `vtkm::VecTraitsTagMultipleComponents` otherwise. This tag can be useful for creating specialized functions when a vector is really just a scalar. If the vector type is of variable size (that is, `IsSizeStatic` is `vtkm::VecTraitsTagSizeVariable`), then `HasMultipleComponents` might be `vtkm::VecTraitsTagMultipleComponents` even when at run time there is only one component.

using **IsSizeStatic** = `vtkm::VecTraitsTagSizeStatic`

A tag specifying whether the size of this vector is known at compile time.

If set to `VecTraitsTagSizeStatic`, then `NUM_COMPONENTS` is set. If set to `VecTraitsTagSizeVariable`, then the number of components is not known at compile time and must be queried with `GetNumberOfComponents`.

template<typename **NewComponentType**>

using **ReplaceComponentType** = `NewComponentType`

Get a vector of the same type but with a different component.

This type resolves to another vector with a different component type. For example, `vtkm::VecTraits<vtkm::Vec<T, N>>::ReplaceComponentType<T2>` is `vtkm::Vec<T2, N>`. This replacement is not recursive. So `VecTraits<Vec<Vec<T, M>, N>::ReplaceComponentType<T2>` is `vtkm::Vec<T2, N>`.

template<typename **NewComponentType**>

using **ReplaceBaseComponentType** = `NewComponentType`

Get a vector of the same type but with a different base component.

This type resolves to another vector with a different base component type. The replacement is recursive for nested types. For example, `VecTraits<Vec<Vec<T, M>, N>::ReplaceBaseComponentType<T2>` is `Vec<Vec<T2, M>, N>`.

## Public Static Functions

static inline constexpr `vtkm::IdComponent` **GetNumberOfComponents**(const *T*&)

Returns the number of components in the given vector.

The result of `GetNumberOfComponents()` is the same value of `NUM_COMPONENTS` for vector types that have a static size (that is, `IsSizeStatic` is `vtkm::VecTraitsTagSizeStatic`). But unlike `NUM_COMPONENTS`, `GetNumberOfComponents()` works for vectors of any type.

static inline const *ComponentType* &**GetComponent**(const *T* &vector, `vtkm::IdComponent`)

Returns the value in a given component of the vector.

static inline *ComponentType* &**GetComponent**(*T* &vector, `vtkm::IdComponent`)

Returns the value in a given component of the vector.



```
static inline void SetComponent(T &vector, vtkm::IdComponent, ComponentType value)
```

Changes the value in a given component of the vector.

```
template<vtkm::IdComponent destSize>
```

```
static inline void CopyInto(const T &src, vtkm::Vec<ComponentType, destSize> &dest)
```

Copies the components in the given vector into a given *Vec* object.

## Public Static Attributes

```
static constexpr vtkm::IdComponent NUM_COMPONENTS = 1
```

Number of components in the vector.

This is only defined for vectors of a static size. That is, `NUM_COMPONENTS` is not available when `IsSizeStatic` is set to `vtkm::VecTraitsTagSizeVariable`.

The `vtkm::VecTraits::HasMultipleComponents` could be one of the following tags.

```
struct VecTraitsTagMultipleComponents
```

A tag for vectors that are “true” vectors (i.e. have more than one component).

```
struct VecTraitsTagSingleComponent
```

A tag for vectors that are really just scalars (i.e. have only one component)

The `vtkm::VecTraits::IsSizeStatic` could be one of the following tags.

```
struct VecTraitsTagSizeStatic
```

A tag for vectors where the number of components are known at compile time.

```
struct VecTraitsTagSizeVariable
```

A tag for vectors where the number of components are not determined until run time.

The definition of `vtkm::VecTraits` for `vtkm::Id3` could look something like this.

Example 10: Example definition of `vtkm::VecTraits<vtkm::Id3>`.

```
1 namespace vtkm {
2
3 template<>
4 struct VecTraits<vtkm::Id3>
5 {
6     using ComponentType = vtkm::Id;
7     using BaseComponentType = vtkm::Id;
8     static const int NUM_COMPONENTS = 3;
9     using IsSizeStatic = vtkm::VecTraitsTagSizeStatic;
10    using HasMultipleComponents = vtkm::VecTraitsTagMultipleComponents;
11
12    VTKM_EXEC_CONT
13    static vtkm::IdComponent GetNumberOfComponents(const vtkm::Id3&)
14    {
```

(continues on next page)

(continued from previous page)

```

15     return NUM_COMPONENTS;
16 }
17
18 VTKM_EXEC_CONT
19 static const vtkm::Id& GetComponent(const vtkm::Id3& vector, int component)
20 {
21     return vector[component];
22 }
23 VTKM_EXEC_CONT
24 static vtkm::Id& GetComponent(vtkm::Id3& vector, int component)
25 {
26     return vector[component];
27 }
28
29 VTKM_EXEC_CONT
30 static void SetComponent(vtkm::Id3& vector, int component, vtkm::Id value)
31 {
32     vector[component] = value;
33 }
34
35 template<typename NewComponentType>
36 using ReplaceComponentType = vtkm::Vec<NewComponentType, 3>;
37
38 template<typename NewComponentType>
39 using ReplaceBaseComponentType = vtkm::Vec<NewComponentType, 3>;
40
41 template<vtkm::IdComponent DestSize>
42 VTKM_EXEC_CONT static void CopyInto(const vtkm::Id3& src,
43                                     vtkm::Vec<vtkm::Id, DestSize>& dest)
44 {
45     for (vtkm::IdComponent index = 0; (index < NUM_COMPONENTS) && (index < DestSize);
46          index++)
47     {
48         dest[index] = src[index];
49     }
50 }
51 };
52
53 } // namespace vtkm

```

The real power of vector traits is that they simplify creating generic operations on any type that can look like a vector. This includes operations on scalar values as if they were vectors of size one. The following code uses vector traits to simplify the implementation of less functors that define an ordering that can be used for sorting and other operations.

Example 11: Using `vtkm::VecTraits` for less functors.

```

1 #include <vtkm/VecTraits.h>
2
3 // This functor provides a total ordering of vectors. Every compared vector
4 // will be either less, greater, or equal (assuming all the vector components
5 // also have a total ordering).
6 template<typename T>

```

(continues on next page)

(continued from previous page)

```

7 struct LessTotalOrder
8 {
9     VTKM_EXEC_CONT
10    bool operator()(const T& left, const T& right)
11    {
12        for (int index = 0; index < vtkm::VecTraits<T>::NUM_COMPONENTS; index++)
13        {
14            using ComponentType = typename vtkm::VecTraits<T>::ComponentType;
15            const ComponentType& leftValue = vtkm::VecTraits<T>::GetComponent(left, index);
16            const ComponentType& rightValue = vtkm::VecTraits<T>::GetComponent(right, index);
17            if (leftValue < rightValue)
18            {
19                return true;
20            }
21            if (rightValue < leftValue)
22            {
23                return false;
24            }
25        }
26        // If we are here, the vectors are equal (or at least equivalent).
27        return false;
28    }
29 };
30
31 // This functor provides a partial ordering of vectors. It returns true if and
32 // only if all components satisfy the less operation. It is possible for
33 // vectors to be neither less, greater, nor equal, but the transitive closure
34 // is still valid.
35 template<typename T>
36 struct LessPartialOrder
37 {
38     VTKM_EXEC_CONT
39    bool operator()(const T& left, const T& right)
40    {
41        for (int index = 0; index < vtkm::VecTraits<T>::NUM_COMPONENTS; index++)
42        {
43            using ComponentType = typename vtkm::VecTraits<T>::ComponentType;
44            const ComponentType& leftValue = vtkm::VecTraits<T>::GetComponent(left, index);
45            const ComponentType& rightValue = vtkm::VecTraits<T>::GetComponent(right, index);
46            if (!(leftValue < rightValue))
47            {
48                return false;
49            }
50        }
51        // If we are here, all components satisfy less than relation.
52        return true;
53    }
54 };

```

## 20.7 List Templates

VTK-m internally uses template metaprogramming, which utilizes C++ templates to run source-generating programs, to customize code to various data and compute platforms. One basic structure often used with template metaprogramming is a list of class names (also sometimes called a tuple or vector, although both of those names have different meanings in VTK-m).

Many VTK-m users only need predefined lists, such as the type lists specified in [Section 20.7.2 \(Type Lists\)](#). Those users can skip most of the details of this section. However, it is sometimes useful to modify lists, create new lists, or operate on lists, and these usages are documented here.

### 20.7.1 Building Lists

A basic list is defined with the `vtkm::List` template.

```
template<typename ...Ts>
```

```
struct List
```

A template used to hold a list of types.

`List` is an empty `struct` that is used to hold a list of types as its template arguments. VTK-m provides templated types that allows a `List` to be manipulated and used in numerous ways.

It is common (but not necessary) to use the `using` keyword to define an alias for a list with a particular meaning.

Example 12: Creating lists of types.

```
1  #include <vtkm/List.h>
2
3  // Placeholder classes representing things that might be in a template
4  // metaprogram list.
5  class Foo;
6  class Bar;
7  class Baz;
8  class Qux;
9  class Xyzzy;
10
11 // The names of the following tags are indicative of the lists they contain.
12
13 using FooList = vtkm::List<Foo>;
14
15 using FooBarList = vtkm::List<Foo, Bar>;
16
17 using BazQuxXyzzyList = vtkm::List<Baz, Qux, Xyzzy>;
18
19 using QuxBazBarFooList = vtkm::List<Qux, Baz, Bar, Foo>;
```

VTK-m defines some special and convenience versions of `vtkm::List`.

```
using vtkm::ListEmpty = vtkm::List<>
```

A convenience type for an empty list.

```
using vtkm::ListUniversal = vtkm::List<detail::UniversalTypeTag>
```

A special type for a list that represents holding all potential values.

Note: This list cannot be used with `ForEach` and some list transforms for obvious reasons.

## 20.7.2 Type Lists

One of the major use cases for template metaprogramming lists in VTK-m is to identify a set of potential data types for arrays. The `vtkm/TypeList.h` header contains predefined lists for known VTK-m types. The following lists are provided.

```
using vtkm::TypeListId = vtkm::List<vtkm::Id>
```

A list containing the type `vtkm::Id`.

```
using vtkm::TypeListId2 = vtkm::List<vtkm::Id2>
```

A list containing the type `vtkm::Id2`.

```
using vtkm::TypeListId3 = vtkm::List<vtkm::Id3>
```

A list containing the type `vtkm::Id3`.

```
using vtkm::TypeListId4 = vtkm::List<vtkm::Id4>
```

A list containing the type `vtkm::Id4`.

```
using vtkm::TypeListIdComponent = vtkm::List<vtkm::IdComponent>
```

A list containing the type `vtkm::IdComponent`.

```
using vtkm::TypeListIndex = vtkm::List<vtkm::Id, vtkm::Id2, vtkm::Id3>
```

A list containing types used to index arrays.

Contains `vtkm::Id`, `vtkm::Id2`, and `vtkm::Id3`.

```
using vtkm::TypeListFieldScalar = vtkm::List<vtkm::Float32, vtkm::Float64>
```

A list containing types used for scalar fields.

Specifically, contains floating point numbers of different widths (i.e. `vtkm::Float32` and `vtkm::Float64`).

```
using vtkm::TypeListFieldVec2 = vtkm::List<vtkm::Vec2f_32, vtkm::Vec2f_64>
```

A list containing types for values for fields with two dimensional vectors.

```
using vtkm::TypeListFieldVec3 = vtkm::List<vtkm::Vec3f_32, vtkm::Vec3f_64>
```

A list containing types for values for fields with three dimensional vectors.

```
using vtkm::TypeListFieldVec4 = vtkm::List<vtkm::Vec4f_32, vtkm::Vec4f_64>
```

A list containing types for values for fields with four dimensional vectors.

```
using vtkm::TypeListFloatVec = vtkm::List<vtkm::Vec2f_32, vtkm::Vec2f_64, vtkm::Vec3f_32, vtkm::Vec3f_64,
vtkm::Vec4f_32, vtkm::Vec4f_64>
```

A list containing common types for floating-point vectors.

Specifically contains floating point vectors of size 2, 3, and 4 with floating point components. Scalars are not included.

```
using vtkm::TypeListField = vtkm::List<vtkm::Float32, vtkm::Float64, vtkm::Vec2f_32, vtkm::Vec2f_64,  
vtkm::Vec3f_32, vtkm::Vec3f_64, vtkm::Vec4f_32, vtkm::Vec4f_64>
```

A list containing common types for values in fields.

Specifically contains floating point scalars and vectors of size 2, 3, and 4 with floating point components.

```
using vtkm::TypeListScalarAll = vtkm::List<vtkm::Int8, vtkm::UInt8, vtkm::Int16, vtkm::UInt16, vtkm::Int32,  
vtkm::UInt32, vtkm::Int64, vtkm::UInt64, vtkm::Float32, vtkm::Float64>
```

A list of all scalars defined in vtkm/Types.h.

A scalar is a type that holds a single number. This should containing all true variations of scalars, but there might be some arithmetic C types not included. For example, this list contains signed char, and unsigned char, but not char as one of those types will behave the same as it. Two of the three types behave the same, but be aware that template resolution will treat them differently.

```
using vtkm::TypeListBaseC = vtkm::ListAppend<vtkm::TypeListScalarAll, vtkm::List<bool, char, signed long,  
unsigned long>>
```

```
using vtkm::TypeListVecCommon = vtkm::List<vtkm::Vec2ui_8, vtkm::Vec2i_32, vtkm::Vec2i_64,  
vtkm::Vec2f_32, vtkm::Vec2f_64, vtkm::Vec3ui_8, vtkm::Vec3i_32, vtkm::Vec3i_64, vtkm::Vec3f_32,  
vtkm::Vec3f_64, vtkm::Vec4ui_8, vtkm::Vec4i_32, vtkm::Vec4i_64, vtkm::Vec4f_32, vtkm::Vec4f_64>
```

A list of the most commonly use *Vec* classes.

Specifically, these are vectors of size 2, 3, or 4 containing either unsigned bytes, signed integers of 32 or 64 bits, or floating point values of 32 or 64 bits.

```
using vtkm::TypeListVecAll = vtkm::ListAppend<vtkm::TypeListVecCommon,  
vtkm::internal::TypeListVecUncommon>
```

A list of all vector classes with standard types as components and lengths between 2 and 4.

```
using vtkm::TypeListAll = vtkm::ListAppend<vtkm::TypeListScalarAll, vtkm::TypeListVecAll>
```

A list of all basic types listed in vtkm/Types.h.

Does not include all possible VTK-m types like arbitrarily typed and sized Vecs (only up to length 4) or math types like matrices.

```
using vtkm::TypeListCommon = vtkm::List<vtkm::UInt8, vtkm::Int32, vtkm::Int64, vtkm::Float32, vtkm::Float64,  
vtkm::Vec3f_32, vtkm::Vec3f_64>
```

A list of the most commonly used types across multiple domains.

Includes integers, floating points, and 3 dimensional vectors of floating points.

If these lists are not sufficient, it is possible to build new type lists using the existing type lists and the list bases from [Section 20.7.1 \(Building Lists\)](#) as demonstrated in the following example.

Example 13: Defining new type lists.

```
1 // A list of 2D vector types.  
2 using Vec2List = vtkm::List<vtkm::Vec2f_32, vtkm::Vec2f_64>;  
3  
4 // An application that uses 2D geometry might commonly encounter this list of  
5 // types.  
6 using MyCommonTypes = vtkm::ListAppend<Vec2List, vtkm::TypeListCommon>;
```

The `vtkm/cont/DefaultTypes.h` header defines a macro named `VTKM_DEFAULT_TYPE_LIST` that defines a default list of types to use when, for example, determining the type of a field array. This macro can change depending on VTK-m compile options.

### 20.7.3 Querying Lists

`vtkm/List.h` contains some templated classes to help get information about a list type. These are particularly useful for lists that are provided as templated parameters for which you do not know the exact type.

#### Is a List

The `VTKM_IS_LIST` does a compile-time check to make sure a particular type is actually a `vtkm::List` of types. If the compile-time check fails, then a build error will occur. This is a good way to verify that a templated class or method that expects a list actually gets a list.

##### `VTKM_IS_LIST(type)`

Checks that the argument is a proper list.

This is a handy concept check for functions and classes to make sure that a template argument is actually a device adapter tag. (You can get weird errors elsewhere in the code when a mistake is made.)

Example 14: Checking that a template parameter is a valid `vtkm::List`.

```

1  template<typename List>
2  class MyImportantClass
3  {
4      VTKM_IS_LIST(List);
5      // Implementation...
6  };
7
8  void DoImportantStuff()
9  {
10     MyImportantClass<vtkm::List<vtkm::Id>> important1; // This compiles fine
11     MyImportantClass<vtkm::Id> important2; // COMPILE ERROR: vtkm::Id is not a list

```

#### List Size

The size of a list can be determined by using the `vtkm::ListSize` template. The type of the template will resolve to a `std::integral_constant<vtkm::IdComponent, N>` where `N` is the number of types in the list. `vtkm::ListSize` does not work with `vtkm::ListUniversal`.

template<typename List>

using `vtkm::ListSize` = typename detail::ListSizeImpl<List>::type

Becomes an `std::integral_constant` containing the number of types in a list.

Example 15: Getting the size of a `vtkm::List`.

```

1  using MyList = vtkm::List<vtkm::Int8, vtkm::Int32, vtkm::Int64>;
2
3  constexpr vtkm::IdComponent myListSize = vtkm::ListSize<MyList>::value;
4  // myListSize is 3

```

## List Contains

The `vtkm::ListHas` template can be used to determine if a `vtkm::List` contains a particular type. `vtkm::ListHas` takes two template parameters. The first parameter is a form of `vtkm::List`. The second parameter is any type to check to see if it is in the list. If the type is in the list, then `vtkm::ListHas` resolves to `std::true_type`. Otherwise it resolves to `std::false_type`. `vtkm::ListHas` always returns true for `vtkm::ListUniversal`.

```
template<typename List, typename T>
```

```
using vtkm::ListHas = typename detail::ListHasImpl<List, T>::type
```

Checks to see if the given T is in the list pointed to by `List`.

Becomes `std::true_type` if the T is in `List`. `std::false_type` otherwise.

Example 16: Determining if a `vtkm::List` contains a particular type.

```
1 using MyList = vtkm::List<vtkm::Int8, vtkm::Int16, vtkm::Int32, vtkm::Int64>;
2
3 constexpr bool hasInt = vtkm::ListHas<MyList, int>::value;
4 // hasInt is true
5
6 constexpr bool hasFloat = vtkm::ListHas<MyList, float>::value;
7 // hasFloat is false
```

## List Indices

The `vtkm::ListIndexOf` template can be used to get the index of a particular type in a `vtkm::List`. `vtkm::ListIndexOf` takes two template parameters. The first parameter is a form of `vtkm::List`. The second parameter is any type to check to see if it is in the list. The type of the template will resolve to a `std::integral_constant<vtkm::IdComponent, N>` where N is the index of the type. If the requested type is not in the list, then `vtkm::ListIndexOf` becomes `std::integral_constant<vtkm::IdComponent, -1>`.

```
template<typename List, typename T>
```

```
using vtkm::ListIndexOf = typename detail::ListIndexOfImpl<List, T>::type
```

Finds the index of a given type.

Becomes a `std::integral_constant` for the index of the given type. If the given type is not in the list, the value is set to -1.

Conversely, the `vtkm::ListAt` template can be used to get the type for a particular index. The two template parameters for `vtkm::ListAt` are the `vtkm::List` and an index for the list.

```
template<typename List, vtkm::IdComponent Index>
```

```
using vtkm::ListAt = typename detail::ListAtImpl<List, Index>::type
```

Finds the type at the given index.

This becomes the type of the list at the given index.

Neither `vtkm::ListIndexOf` nor `vtkm::ListAt` works with `vtkm::ListUniversal`.

Example 17: Using indices with `vtkm::List`.

```
1 using MyList = vtkm::List<vtkm::Int8, vtkm::Int32, vtkm::Int64>;
2
3 constexpr vtkm::IdComponent indexOfInt8 = vtkm::ListIndexOf<MyList, vtkm::Int8>::value;
```

(continues on next page)



(continued from previous page)

```

4 // indexOfInt8 is 0
5 constexpr vtkm::IdComponent indexOfInt32 =
6   vtkm::ListIndexOf<MyList, vtkm::Int32>::value;
7 // indexOfInt32 is 1
8 constexpr vtkm::IdComponent indexOfInt64 =
9   vtkm::ListIndexOf<MyList, vtkm::Int64>::value;
10 // indexOfInt64 is 2
11 constexpr vtkm::IdComponent indexOfFloat32 =
12   vtkm::ListIndexOf<MyList, vtkm::Float32>::value;
13 // indexOfFloat32 is -1 (not in list)
14
15 using T0 = vtkm::ListAt<MyList, 0>; // T0 is vtkm::Int8
16 using T1 = vtkm::ListAt<MyList, 1>; // T1 is vtkm::Int32
17 using T2 = vtkm::ListAt<MyList, 2>; // T2 is vtkm::Int64

```

## 20.7.4 Operating on Lists

In addition to providing the base templates for defining and querying lists, `vtkm/List.h` also contains several features for operating on lists.

### Appending Lists

The `vtkm::ListAppend` template joins together 2 or more `vtkm::List` types. The items are concatenated in the order provided to `vtkm::ListAppend`. `vtkm::ListAppend` does not work with `vtkm::ListUniversal`.

```
template<typename ...Lists>
```

```
using vtkm::ListAppend = typename detail::ListAppendImpl<Lists...>::type
```

Concatinates a set of lists into a single list.

Note that this does not work correctly with `vtkm::ListUniversal`.

Example 18: Appending `vtkm::List` types.

```

1 using BigTypes = vtkm::List<vtkm::Int64, vtkm::Float64>;
2 using MediumTypes = vtkm::List<vtkm::Int32, vtkm::Float32>;
3 using SmallTypes = vtkm::List<vtkm::Int8>;
4
5 using SmallAndBigTypes = vtkm::ListAppend<SmallTypes, BigTypes>;
6 // SmallAndBigTypes is vtkm::List<vtkm::Int8, vtkm::Int64, vtkm::Float64>
7
8 using AllMyTypes = vtkm::ListAppend<BigTypes, MediumTypes, SmallTypes>;
9 // AllMyTypes is
10 // vtkm::List<vtkm::Int64, vtkm::Float64, vtkm::Int32, vtkm::Float32, vtkm::Int8>

```

## Intersecting Lists

The `vtkm::ListIntersect` template takes two `vtkm::List` types and becomes a `vtkm::List` containing all types in both lists. If one of the lists is `vtkm::ListUniversal`, the contents of the other list used.

```
template<typename List1, typename List2>
```

```
using vtkm::ListIntersect = typename detail::ListIntersectImpl<List1, List2>::type
```

Constructs a list containing types present in all lists.

Example 19: Intersecting `vtkm::List` types.

```

1 using SignedInts = vtkm::List<vtkm::Int8, vtkm::Int16, vtkm::Int32, vtkm::Int64>;
2 using WordTypes = vtkm::List<vtkm::Int32, vtkm::UInt32, vtkm::Int64, vtkm::UInt64>;
3
4 using SignedWords = vtkm::ListIntersect<SignedInts, WordTypes>;
5 // SignedWords is vtkm::List<vtkm::Int32, vtkm::Int64>

```

## Resolve a Template with all Types in a List

The `vtkm::ListApply` template transfers all of the types in a `vtkm::List` to another template. The first template argument of `vtkm::ListApply` is the `vtkm::List` to apply. The second template argument is another template to apply to. `vtkm::ListApply` becomes an instance of the passed template with all the types in the `vtkm::List`. `vtkm::ListApply` can be used to convert a `vtkm::List` to some other template. `vtkm::ListApply` cannot be used with `vtkm::ListUniversal`.

```
template<typename List, template<typename...> class Target>
```

```
using vtkm::ListApply = typename detail::ListApplyImpl<List, Target>::type
```

Applies the list of types to a template.

Given a ListTag and a templated class, returns the class instantiated with the types represented by the ListTag.

Example 20: Applying a `vtkm::List` to another template.

```

1 using MyList = vtkm::List<vtkm::Id, vtkm::Id3, vtkm::Vec3f>;
2
3 using MyTuple = vtkm::ListApply<MyList, std::tuple>;
4 // MyTuple is std::tuple<vtkm::Id, vtkm::Id3, vtkm::Vec3f>

```

## Transform Each Type in a List

The `vtkm::ListTransform` template applies each item in a `vtkm::List` to another template and constructs a list from all these applications. The first template argument of `vtkm::ListTransform` is the `vtkm::List` to apply. The second template argument is another template to apply to. `vtkm::ListTransform` becomes an instance of a new `vtkm::List` containing the passed template each type. `vtkm::ListTransform` cannot be used with `vtkm::ListUniversal`.

```
template<typename List, template<typename> class Transform>
```

```
using vtkm::ListTransform = typename detail::ListTransformImpl<List, Transform>::type
```

Constructs a list containing all types in a source list applied to a transform template.

Example 21: Transforming a `vtkm::List` using a custom template.

```
1 using MyList = vtkm::List<vtkm::Int32, vtkm::Float32>;
2
3 template<typename T>
4 using MakeVec = vtkm::Vec<T, 3>;
5
6 using MyVecList = vtkm::ListTransform<MyList, MakeVec>;
7 // MyVecList is vtkm::List<vtkm::Vec<vtkm::Int32, 3>, vtkm::Vec<vtkm::Float32, 3>>
```

## Conditionally Removing Items from a List

The `vtkm::ListRemoveIf` template removes items from a `vtkm::List` given a predicate. The first template argument of `vtkm::ListRemoveIf` is the `vtkm::List`. The second argument is another template that is used as a predicate to determine if the type should be removed or not. The predicate should become a type with a value member that is a static true or false value. Any type in the list that the predicate evaluates to true is removed. `vtkm::ListRemoveIf` cannot be used with `vtkm::ListUniversal`.

```
template<typename List, template<typename> class Predicate>
```

```
using vtkm::ListRemoveIf = typename detail::ListRemoveIfImpl<List, Predicate>::type
```

Takes an existing `List` and a predicate template that is applied to each type in the `List`.

Any type in the `List` that has a value element equal to true (the equivalent of `std::true_type`), that item will be removed from the list. For example the following type

```
vtkm::ListRemoveIf<vtkm::List<int, float, long long, double>, std::is_integral>
```

resolves to a `List` that is equivalent to `vtkm::List<float, double>` because `std::is_integral<int>` and `std::is_integral<long long>` resolve to `std::true_type` whereas `std::is_integral<float>` and `std::is_integral<double>` resolve to `std::false_type`.

Example 22: Removing items from a `vtkm::List`.

```
1 using MyList =
2   vtkm::List<vtkm::Int64, vtkm::Float64, vtkm::Int32, vtkm::Float32, vtkm::Int8>;
3
4 using FilteredList = vtkm::ListRemoveIf<MyList, std::is_integral>;
5 // FilteredList is vtkm::List<vtkm::Float64, vtkm::Float32>
```

## Combine all Pairs of Two Lists

The `vtkm::ListCross` takes two lists and performs a cross product of them. It does this by creating a new `vtkm::List` that contains nested `vtkm::List` types, each of length 2 and containing all possible pairs of items in the first list with items in the second list. `vtkm::ListCross` is often used in conjunction with another list processing command, such as `vtkm::ListTransform` to build templated types of many combinations. `vtkm::ListCross` cannot be used with `vtkm::ListUniversal`.

```
template<typename List1, typename List2>
```

```
using vtkm::ListCross = typename detail::ListCrossImpl<List1, List2>::type
```

Generates a list that is the cross product of two input lists.

The resulting list has the form of `vtkm::List<vtkm::List<A1,B1>, vtkm::List<A1,B2>,...>`

Example 23: Creating the cross product of 2 `vtkm::List` types.

```
1 using BaseTypes = vtkm::List<vtkm::Int8, vtkm::Int32, vtkm::Int64>;
2 using BoolCases = vtkm::List<std::false_type, std::true_type>;
3
4 using CrossTypes = vtkm::ListCross<BaseTypes, BoolCases>;
5 // CrossTypes is
6 //   vtkm::List<vtkm::List<vtkm::Int8, std::false_type>,
7 //               vtkm::List<vtkm::Int8, std::true_type>,
8 //               vtkm::List<vtkm::Int32, std::false_type>,
9 //               vtkm::List<vtkm::Int32, std::true_type>,
10 //              vtkm::List<vtkm::Int64, std::false_type>,
11 //              vtkm::List<vtkm::Int64, std::true_type>>
12
13 template<typename TypeAndIsVec>
14 using ListPairToType =
15     typename std::conditional<vtkm::ListAt<TypeAndIsVec, 1>::value,
16                               vtkm::Vec<vtkm::ListAt<TypeAndIsVec, 0>, 3>,
17                               vtkm::ListAt<TypeAndIsVec, 0>>::type;
18
19 using AllTypes = vtkm::ListTransform<CrossTypes, ListPairToType>;
20 // AllTypes is
21 //   vtkm::List<vtkm::Int8,
22 //               vtkm::Vec<vtkm::Int8, 3>,
23 //               vtkm::Int32,
24 //               vtkm::Vec<vtkm::Int32, 3>,
25 //               vtkm::Int64,
26 //               vtkm::Vec<vtkm::Int64, 3>>
```

## Call a Function For Each Type in a List

The `vtkm::ListForEach` function takes a functor object and a `vtkm::List`. It then calls the functor object with the default object of each type in the list. This is most typically used with C++ run-time type information to convert a run-time polymorphic object to a statically typed (and possibly inlined) call.

```
template<typename Functor, typename ...Ts, typename ...Args>
```

```
void vtkm::ListForEach(Functor &&f, vtkm::List<Ts...>, Args&&... args)
```

For each typename represented by the list, call the functor with a default instance of that type.

The following example shows a rudimentary version of converting a dynamically-typed array to a statically-typed array similar to what is done in VTK-m classes like `vtkm::cont::UnknownArrayHandle` (which is documented in [Chapter~ref{chap:UnknownArrayHandle}](#)).

Example 24: Converting dynamic types to static types with `vtkm::ListForEach`.

```

1 struct MyArrayBase
2 {
3     // A virtual destructor makes sure C++ RTTI will be generated. It also helps
4     // ensure subclass destructors are called.
5     virtual ~MyArrayBase() {}
6 };
7
8 template<typename T>
9 struct MyArrayImpl : public MyArrayBase
10 {
11     std::vector<T> Array;
12 };
13
14 template<typename T>
15 void PrefixSum(std::vector<T>& array)
16 {
17     T sum(vtkm::VecTraits<T>::ComponentType(0));
18     for (typename std::vector<T>::iterator iter = array.begin(); iter != array.end();
19          iter++)
20     {
21         sum = sum + *iter;
22         *iter = sum;
23     }
24 }
25
26 struct PrefixSumFunctor
27 {
28     MyArrayBase* ArrayPointer;
29
30     PrefixSumFunctor(MyArrayBase* arrayPointer)
31         : ArrayPointer(arrayPointer)
32     {
33     }
34
35     template<typename T>
36     void operator()(T)
37     {
38         using ConcreteArrayType = MyArrayImpl<T>;
39         ConcreteArrayType* concreteArray =
40             dynamic_cast<ConcreteArrayType*>(this->ArrayPointer);
41         if (concreteArray != NULL)
42         {
43             PrefixSum(concreteArray->Array);
44         }
45     }
46 };
47

```

(continues on next page)

(continued from previous page)

```

48 void DoPrefixSum(MyArrayBase* array)
49 {
50     PrefixSumFunctor functor = PrefixSumFunctor(array);
51     vtkm::ListForEach(functor, vtkm::TypeListCommon());
52 }

```

## 20.8 Pair

VTK-m defines a `vtkm::Pair` templated object that behaves just like `std::pair` from the standard template library. The difference is that `vtkm::Pair` will work in both the execution and control environments, whereas the STL `std::pair` does not always work in the execution environment.

template<typename **T1**, typename **T2**>

struct **Pair**

A `vtkm::Pair` is essentially the same as an STL pair object except that the methods (constructors and operators) are defined to work in both the control and execution environments (whereas `std::pair` is likely to work only in the control environment).

### Public Types

using **FirstType** = *T1*

The type of the first object.

using **SecondType** = *T2*

The type of the second object.

using **first\_type** = *FirstType*

The same as `FirstType`, but follows the naming convention of `std::pair`.

using **second\_type** = *SecondType*

The same as `SecondType`, but follows the naming convention of `std::pair`.

### Public Functions

**Pair**() = default

inline **Pair**(const *FirstType* &firstSrc, const *SecondType* &secondSrc)

inline **Pair**(*FirstType* &&firstSrc, *SecondType* &&secondSrc)  
 noexcept(noexcept(*FirstType*{std::declval<*FirstType*&&>()},  
*SecondType*{std::declval<*SecondType*&&>()}))

**Pair**(const *Pair*&) = default

**Pair**(*Pair*&&) = default

template<typename **U1**, typename **U2**>

```

inline Pair(const vtkm::Pair<U1, U2> &src)

template<typename U1, typename U2>
inline Pair(vtkm::Pair<U1, U2> &&src) noexcept(noexcept(U1{std::declval<U1&&>()}),
        U2{std::declval<U2&&>()}))

template<typename U1, typename U2>
inline Pair(const std::pair<U1, U2> &src)

template<typename U1, typename U2>
inline Pair(std::pair<U1, U2> &&src) noexcept(noexcept(U1{std::declval<U1&&>()}),
        U2{std::declval<U2&&>()}))

vtkm::Pair<FirstType, SecondType> &operator=(const vtkm::Pair<FirstType, SecondType> &src) = default
vtkm::Pair<FirstType, SecondType> &operator=(vtkm::Pair<FirstType, SecondType> &&src) = default

inline bool operator==(const vtkm::Pair<FirstType, SecondType> &other) const
inline bool operator!=(const vtkm::Pair<FirstType, SecondType> &other) const
inline bool operator<(const vtkm::Pair<FirstType, SecondType> &other) const
    Tests ordering on the first object, and then on the second object if the first are equal.
inline bool operator>(const vtkm::Pair<FirstType, SecondType> &other) const
    Tests ordering on the first object, and then on the second object if the first are equal.
inline bool operator<=(const vtkm::Pair<FirstType, SecondType> &other) const
    Tests ordering on the first object, and then on the second object if the first are equal.
inline bool operator>=(const vtkm::Pair<FirstType, SecondType> &other) const
    Tests ordering on the first object, and then on the second object if the first are equal.

```

## Public Members

### *FirstType* first

The pair's first object.

Note that this field breaks VTK-m's naming conventions to make *vtkm::Pair* more compatible with `std::pair`.

### *SecondType* second

The pair's second object.

Note that this field breaks VTK-m's naming conventions to make *vtkm::Pair* more compatible with `std::pair`.

The VTK-m version of *vtkm::Pair* supports the same types, fields, and operations as the STL version. VTK-m also provides a *vtkm::make\_Pair()* function for convenience.

```

template<typename T1, typename T2>
vtkm::Pair<typename std::decay<T1>::type, typename std::decay<T2>::type> vtkm::make_Pair(T1 &&v1, T2
        &&v2)

```

## 20.9 Tuple

VTK-m defines a `vtkm::Tuple` templated object that behaves like `std::tuple` from the standard template library. The main difference is that `vtkm::Tuple` will work in both the execution and control environments, whereas the STL `std::tuple` does not always work in the execution environment.

```
template<typename ...Ts>
```

class **Tuple**

VTK-m replacement for `std::tuple`.

This function serves the same function as `std::tuple` and behaves similarly. However, this version of `Tuple` works on devices that VTK-m supports. There are also some implementation details that makes compiling faster for VTK-m use. We also provide some methods like `Apply` and `ForEach` that are helpful for several VTK-m operations.

### 20.9.1 Defining and Constructing

`vtkm::Tuple` takes any number of template parameters that define the objects stored the tuple.

Example 25: Defining a `vtkm::Tuple`.

```
1  vtkm::Tuple<vtkm::Id, vtkm::Vec3f, vtkm::cont::ArrayHandle<vtkm::Int32>> myTuple;
```

You can construct a `vtkm::Tuple` with arguments that will be used to initialize the respective objects. As a convenience, you can use `vtkm::MakeTuple()` to construct a `vtkm::Tuple` of types based on the arguments.

```
template<typename ...Ts>
```

```
auto vtkm::MakeTuple(Ts&&... args) -> vtkm::Tuple<typename std::decay<Ts>::type...>
```

Creates a new `vtkm::Tuple` with the given types.

```
template<typename ...Ts>
```

```
auto vtkm::make_tuple(Ts&&... args) -> decltype(vtkm::MakeTuple(std::forward<Ts>(args)...))
```

Compatible with `std::make_tuple` for `vtkm::Tuple`.

Example 26: Initializing values in a `vtkm::Tuple`.

```
1  // Initialize a tuple with 0, [0, 1, 2], and an existing ArrayHandle.
2  vtkm::Tuple<vtkm::Id, vtkm::Vec3f, vtkm::cont::ArrayHandle<vtkm::Float32>> myTuple1(
3      0, vtkm::Vec3f(0, 1, 2), array);
4
5  // Another way to create the same tuple.
6  auto myTuple2 = vtkm::MakeTuple(vtkm::Id(0), vtkm::Vec3f(0, 1, 2), array);
```

### 20.9.2 Querying

The size of a `vtkm::Tuple` can be determined by using the `vtkm::TupleSize` template, which resolves to an `std::integral_constant`. The types at particular indices can be determined with `vtkm::TupleElement`.

```
template<typename TupleType>
```

```
using vtkm::TupleSize = std::integral_constant<vtkm::IdComponent, TupleType::Size>
```

Get the size of a tuple.

Given a `vtkm::Tuple` type, becomes a `std::integral_constant` of the type.



```
template<vtkm::IdComponent Index, typename TupleType>
```

```
using vtkm::TupleElement = typename detail::TupleElementImpl<Index, TupleType>::type
```

Becomes the type of the given index for the given `vtkm::Tuple`.

Example 27: Querying `vtkm::Tuple` types.

```
1 using TupleType = vtkm::Tuple<vtkm::Id, vtkm::Float32, vtkm::Float64>;
2
3 // Becomes 3
4 constexpr vtkm::IdComponent size = vtkm::TupleSize<TupleType>::value;
5
6 using FirstType = vtkm::TupleElement<0, TupleType>; // vtkm::Id
7 using SecondType = vtkm::TupleElement<1, TupleType>; // vtkm::Float32
8 using ThirdType = vtkm::TupleElement<2, TupleType>; // vtkm::Float64
```

The function `vtkm::Get()` can be used to retrieve an element from the `vtkm::Tuple`. `vtkm::Get()` returns a reference to the element, so you can set a `vtkm::Tuple` element by setting the return value of `vtkm::Get()`.

```
template<vtkm::IdComponent Index, typename ...Ts>
```

```
auto vtkm::Get(const vtkm::Tuple<Ts...> &tuple)
```

Retrieve the object from a `vtkm::Tuple` at the given index.

```
template<vtkm::IdComponent Index, typename ...Ts>
```

```
auto vtkm::Get(vtkm::Tuple<Ts...> &tuple)
```

Retrieve the object from a `vtkm::Tuple` at the given index.

```
template<std::size_t Index, typename ...Ts>
```

```
auto vtkm::get(const vtkm::Tuple<Ts...> &tuple) ->
```

```
decltype(vtkm::Get<static_cast<vtkm::IdComponent>(Index)>(tuple))
```

Compatible with `std::get` for `vtkm::Tuple`.

```
template<std::size_t Index, typename ...Ts>
```

```
auto vtkm::get(vtkm::Tuple<Ts...> &tuple) ->
```

```
decltype(vtkm::Get<static_cast<vtkm::IdComponent>(Index)>(tuple))
```

Compatible with `std::get` for `vtkm::Tuple`.

Example 28: Retrieving values from a `vtkm::Tuple`.

```
1  auto myTuple = vtkm::MakeTuple(vtkm::Id3(0, 1, 2), vtkm::Vec3f(3, 4, 5));
2
3  // Gets the value [0, 1, 2]
4  vtkm::Id3 x = vtkm::Get<0>(myTuple);
5
6  // Changes the second object in myTuple to [6, 7, 8]
7  vtkm::Get<1>(myTuple) = vtkm::Vec3f(6, 7, 8);
```

### 20.9.3 For Each Tuple Value

The `vtkm::ForEach()` function takes a tuple and a function or functor and calls the function for each of the items in the tuple. Nothing is returned from `vtkm::ForEach()`, and any return value from the function is ignored.

```
template<typename ...Ts, typename Function>
void vtkm::ForEach(const vtkm::Tuple<Ts...> &tuple, Function &&f)
```

Call a function with each value of the given tuple.

The function calls will be done in the order of the values in the `vtkm::Tuple`.

```
template<typename ...Ts, typename Function>
void vtkm::ForEach(vtkm::Tuple<Ts...> &tuple, Function &&f)
```

Call a function with each value of the given tuple.

The function calls will be done in the order of the values in the `vtkm::Tuple`.

`vtkm::ForEach()` can be used to check the validity of each item in a `vtkm::Tuple`.

Example 29: Using `vtkm::Tuple::ForEach()` to check the contents.

```
1  void CheckPositive(vtkm::Float64 x)
2  {
3      if (x < 0)
4      {
5          throw vtkm::cont::ErrorBadValue("Values need to be positive.");
6      }
7  }
8
9  // ...
10
11  vtkm::Tuple<vtkm::Float64, vtkm::Float64, vtkm::Float64> tuple(
12      CreateValue(0), CreateValue(1), CreateValue(2));
13
14  // Will throw an error if any of the values are negative.
15  vtkm::ForEach(tuple, CheckPositive);
```

`vtkm::ForEach()` can also be used to aggregate values in a `vtkm::Tuple`.

Example 30: Using `vtkm::Tuple::ForEach()` to aggregate.

```
1  struct SumFunctor
2  {
3      vtkm::Float64 Sum = 0;
```

(continues on next page)

(continued from previous page)

```

4
5  template<typename T>
6  void operator()(const T& x)
7  {
8      this->Sum = this->Sum + static_cast<vtkm::Float64>(x);
9  }
10 };
11
12 // ...
13
14 vtkm::Tuple<vtkm::Float32, vtkm::Float64, vtkm::Id> tuple(
15     CreateValue(0), CreateValue(1), CreateValue(2));
16
17 SumFunction sum;
18 vtkm::ForEach(tuple, sum);
19 vtkm::Float64 average = sum.Sum / 3;

```

The previous examples used an explicit struct as the functor for clarity. However, it is often less verbose to use a C++ lambda function.

Example 31: Using `vtkm::Tuple::ForEach()` to aggregate.

```

1  vtkm::Tuple<vtkm::Float32, vtkm::Float64, vtkm::Id> tuple(
2      CreateValue(0), CreateValue(1), CreateValue(2));
3
4  vtkm::Float64 sum = 0;
5  auto sumFunction = [&sum](auto x) { sum += static_cast<vtkm::Float64>(x); };
6
7  vtkm::ForEach(tuple, sumFunction);
8  vtkm::Float64 average = sum / 3;

```

## 20.9.4 Transform Each Tuple Value

The `vtkm::Transform()` function builds a new `vtkm::Tuple` by calling a function or functor on each of the items in an existing `vtkm::Tuple`. The return value is placed in the corresponding part of the resulting `vtkm::Tuple`, and the type is automatically created from the return type of the function.

```

template<typename TupleType, typename Function>
auto vtkm::Transform(const TupleType &&tuple, Function &&f) -> decltype(Apply(tuple,
    detail::TupleTransformFunctor(), std::forward<Function>(f)))

```

Construct a new `vtkm::Tuple` by applying a function to each value.

The `vtkm::Transform` function builds a new `vtkm::Tuple` by calling a function or functor on each of the items in the given tuple. The return value is placed in the corresponding part of the resulting `Tuple`, and the type is automatically created from the return type of the function.

```

template<typename TupleType, typename Function>
auto vtkm::Transform(TupleType &&tuple, Function &&f) -> decltype(Apply(tuple,
    detail::TupleTransformFunctor(), std::forward<Function>(f)))

```

Get the size of a tuple.

Given a `vtkm::Tuple` type, becomes a `std::integral_constant` of the type.

Example 32: Transforming a `vtkm::Tuple`.

```

1 struct GetReadPortalFuncor
2 {
3     template<typename Array>
4     typename Array::ReadPortalType operator()(const Array& array) const
5     {
6         VTKM_IS_ARRAY_HANDLE(Array);
7         return array.ReadPortal();
8     }
9 };
10
11 // ...
12
13 auto arrayTuple = vtkm::MakeTuple(array1, array2, array3);
14
15 auto portalTuple = vtkm::Transform(arrayTuple, GetReadPortalFuncor{});

```

## 20.9.5 Apply

The `vtkm::Apply()` function calls a function or functor using the objects in a `vtkm::Tuple` as the arguments. If the function returns a value, that value is returned from `vtkm::Apply()`.

```

template<typename ...Ts, typename Function, typename ...Args>
auto vtkm::Apply(const vtkm::Tuple<Ts...> &tuple, Function &&f, Args&&... args) ->
    decltype(tuple.Apply(std::forward<Function>(f), std::forward<Args>(args)...))

```

Call a function with the values of a `vtkm::Tuple` as arguments.

If a `vtkm::Tuple<A, B, C>` is given with values a, b, and c, then f will be called as f(a, b, c).

Additional arguments can optionally be given to `vtkm::Apply()`. These arguments will be added to the *beginning* of the arguments to the function.

The returned value of the function (if any) will be returned from `vtkm::Apply()`.

```

template<typename ...Ts, typename Function, typename ...Args>
auto vtkm::Apply(vtkm::Tuple<Ts...> &tuple, Function &&f, Args&&... args) ->
    decltype(tuple.Apply(std::forward<Function>(f), std::forward<Args>(args)...))

```

Call a function with the values of a `vtkm::Tuple` as arguments.

If a `vtkm::Tuple<A, B, C>` is given with values a, b, and c, then f will be called as f(a, b, c).

Additional arguments can optionally be given to `vtkm::Apply()`. These arguments will be added to the *beginning* of the arguments to the function.

The returned value of the function (if any) will be returned from `vtkm::Apply()`.

Example 33: Applying a `vtkm::Tuple` as arguments to a function.

```

1 struct AddArraysFuncor
2 {
3     template<typename Array1, typename Array2, typename Array3>
4     vtkm::Id operator()(Array1 inArray1, Array2 inArray2, Array3 outArray) const
5     {
6         VTKM_IS_ARRAY_HANDLE(Array1);

```

(continues on next page)

(continued from previous page)

```

7   VTKM_IS_ARRAY_HANDLE(Array2);
8   VTKM_IS_ARRAY_HANDLE(Array3);
9
10  vtkm::Id length = inArray1.GetNumberOfValues();
11  VTKM_ASSERT(inArray2.GetNumberOfValues() == length);
12  outArray.Allocate(length);
13
14  auto inPortal1 = inArray1.ReadPortal();
15  auto inPortal2 = inArray2.ReadPortal();
16  auto outPortal = outArray.WritePortal();
17  for (vtkm::Id index = 0; index < length; ++index)
18  {
19      outPortal.Set(index, inPortal1.Get(index) + inPortal2.Get(index));
20  }
21
22  return length;
23 }
24 };
25
26 // ...
27
28 auto arrayTuple = vtkm::MakeTuple(array1, array2, array3);
29
30 vtkm::Id arrayLength = vtkm::Apply(arrayTuple, AddArraysFunctor{});

```

If additional arguments are given to `vtkm::Apply()`, they are also passed to the function (before the objects in the `vtkm::Tuple`). This is helpful for passing state to the function.

Example 34: Using extra arguments with `vtkm::Tuple::Apply()`.

```

1  struct ScanArrayLengthFunctor
2  {
3      template<vtkm::IdComponent N, typename Array, typename... Remaining>
4      vtkm::Vec<vtkm::Id, N + 1 + vtkm::IdComponent(sizeof...(Remaining))> operator()(
5          const vtkm::Vec<vtkm::Id, N>& partialResult,
6          const Array& nextArray,
7          const Remaining&... remainingArrays) const
8      {
9          vtkm::Vec<vtkm::Id, N + 1> nextResult;
10         std::copy(&partialResult[0], &partialResult[0] + N, &nextResult[0]);
11         nextResult[N] = nextResult[N - 1] + nextArray.GetNumberOfValues();
12         return (*this)(nextResult, remainingArrays...);
13     }
14
15     template<vtkm::IdComponent N>
16     vtkm::Vec<vtkm::Id, N> operator()(const vtkm::Vec<vtkm::Id, N>& result) const
17     {
18         return result;
19     }
20 };
21
22 // ...

```

(continues on next page)

(continued from previous page)

```

23
24 auto arrayTuple = vtkm::MakeTuple(array1, array2, array3);
25
26 vtkm::Vec<vtkm::Id, 4> sizeScan =
27   vtkm::Apply(arrayTuple, ScanArrayLengthFunctor{}, vtkm::Vec<vtkm::Id, 1>{ 0 });

```

## 20.10 Error Codes

For operations that occur in the control environment, VTK-m uses exceptions to report errors as described in [Chapter 12 \(Error Handling\)](#). However, when operating in the execution environment, it is not feasible to throw exceptions. Thus, for operations designed for the execution environment, the status of an operation that can fail is returned as an `vtkm::ErrorCode`, which is an enum.

enum class `vtkm::ErrorCode`

Identifies whether an operation was successful or what type of error it had.

Most errors in VTK-m are reported by throwing an exception. However, there are some places, most notably the execution environment, where it is not possible to throw an exception. For those cases, it is typical for a function to return an `ErrorCode` identifier. The calling code can check to see if the operation was a success or what kind of error was encountered otherwise.

Use the `vtkm::ErrorString()` function to get a descriptive string of the error type.

*Values:*

enumerator **Success**

A successful operation.

This code is returned when the operation was successful. Calling code should check the error code against this identifier when checking the status.

enumerator **InvalidShapeId**

A unknown shape identifier was encountered.

All cell shapes must be listed in `vtkm::CellShapeIdEnum`.

enumerator **InvalidNumberOfPoints**

The wrong number of points was provided for a given cell type.

For example, if a triangle has 4 points associated with it, you are likely to get this error.

enumerator **InvalidCellMetric**

A cell metric was requested for a cell that does not support that metric.

enumerator **WrongShapeIdForTagType**

This is an internal error from the lightweight cell library.

enumerator **InvalidPointId**

A bad point identifier was detected while operating on a cell.

**enumerator InvalidEdgeId**

A bad edge identifier was detected while operating on a cell.

**enumerator InvalidFaceId**

A bad face identifier was detected while operating on a cell.

**enumerator SolutionDidNotConverge**

An iterative operation did not find an appropriate solution.

This error code might be returned with some results of an iterative solution. However, solution did not appear to resolve, so the results might not be accurate.

**enumerator MatrixFactorizationFailed**

A solution was not found for a linear system.

Some VTK-m computations use linear algebra to solve a system of equations. If the equations does not give a valid result, this error can be returned.

**enumerator DegenerateCellDetected**

An operation detected a degenerate cell.

A degenerate cell has two or more vertices combined into one, which changes the structure of the cell. For example, if 2 vertices of a tetrahedron are at the same point, the cell degenerates to a triangle. Degenerate cells have the potential to interfere with some computations on cells.

**enumerator MalformedCellDetected**

An operation detected on a malformed cell.

Most cell shapes have some assumptions about their geometry (e.g. not self intersecting). If an operation detects an expected behavior is violated, this error is returned. (Note that `vtkm::DegenerateCellDetected` has its own error code.)

**enumerator OperationOnEmptyCell**

An operation was attempted on a cell with an empty shape.

There is a special “empty” cell shape type (`vtkm::CellShapeTagEmpty`) that can be used as a placeholder for a cell with no information. Math operations such as interpolation cannot be performed on empty cells, and attempting to do so will result in this error.

**enumerator CellNotFound**

A cell matching some given criteria could not be found.

This error code is most often used in a cell locator where no cell in the given region could be found.

**enumerator UnknownError**

If a function or method returns an `vtkm::ErrorCode`, it is a good practice to check to make sure that the returned value is `vtkm::ErrorCode::Success`. If it is not, you can use the `vtkm::ErrorString()` function to convert the `vtkm::ErrorCode` to a descriptive C string. The easiest thing to do from within a worklet is to call the worklet's `RaiseError` method.

inline const char \*vtkm::ErrorString(vtkm::ErrorCode code) noexcept

Convert a *vtkm::ErrorCode* into a human-readable string.

This method is useful when reporting the results of a function that failed.

Example 35: Checking an *vtkm::ErrorCode* and reporting errors in a worklet.

```
1  vtkm::ErrorCode status = cellLocator.FindCell(point, cellId, parametric);
2  if (status != vtkm::ErrorCode::Success)
3  {
4      this->RaiseError(vtkm::ErrorString(status));
5  }
```



## LOGGING

VTK-m features a logging system that allows status updates and timing. VTK-m uses the loguru project to provide runtime logging facilities. A sample of the log output can be found at <https://gitlab.kitware.com/snippets/427>.

### 21.1 Initializing Logging

Logging features are enabled by calling `vtkm::cont::Initialize()` as described in [Chapter 6 \(Initialization\)](#). Although calling `vtkm::cont::Initialize()` is not strictly necessary for output messages, initialization adds the following features.

- Set human-readable names for the log levels in the output.
- Allow the stderr logging level to be set at runtime by passing a `--vtkm-log-level [level]` argument to the executable.
- Name the main thread.
- Print a preamble with details of the program's startup (arguments, etc).

[Example 1](#) in the following section provides an example of initializing with additional logging setup.

The logging implementation is thread-safe. When working in a multithreaded environment, each thread may be assigned a human-readable name using `vtkm::cont::SetLogThreadName()` (which can later be retrieved with `vtkm::cont::GetLogThreadName()`). This name will appear in the log output so that per-thread messages can be easily tracked.

```
void vtkm::cont::SetLogThreadName(const std::string &name)
```

Specifies a human-readable name to identify the current thread in the log output.

```
std::string vtkm::cont::GetLogThreadName()
```

Specifies a human-readable name to identify the current thread in the log output.

### 21.2 Logging Levels

The logging in VTK-m provides several “levels” of logging. Logging levels are ordered by precedence. When selecting which log message to output, a single logging level is provided. Any logging message with that or a higher precedence is output. For example, if warning messages are on, then error messages are also outputted because errors are a higher precedence than warnings. Likewise, if information messages are on, then error and warning messages are also outputted.

---

#### Common Errors

All logging levels are assigned a number, and logging levels with a higher precedence actually have a smaller number.

---

All logging levels are listed in the `vtkm::cont::LogLevel` enum.

enum class `vtkm::cont::LogLevel`

Log levels for use with the logging macros.

*Values:*

enumerator **Off**

A placeholder used to silence all logging.

Do not actually log to this level.

enumerator **Fatal**

Fatal errors that should abort execution.

enumerator **Error**

Important but non-fatal errors, such as device fail-over.

enumerator **Warn**

Less important user errors, such as out-of-bounds parameters.

enumerator **Info**

Information messages (detected hardware, etc) and temporary debugging output.

enumerator **UserFirst**

The first in a range of logging levels reserved for code that uses VTK-m.

Internal VTK-m code will not log on these levels but will report these logs.

enumerator **UserLast**

The last in a range of logging levels reserved for code that uses VTK-m.

enumerator **DevicesEnabled**

Information about which devices are enabled/disabled.

enumerator **Perf**

General timing data and algorithm flow information, such as filter execution, worklet dispatches, and device algorithm calls.

enumerator **MemCont**

Host-side resource allocations/frees (e.g. *ArrayHandle* control buffers).

enumerator **MemExec**

Device-side resource allocations/frees (e.g *ArrayHandle* device buffers).

**enumerator `MemTransfer`**

Transferring of data between a host and device.

**enumerator `KernelLaunches`**

Details on device-side kernel launches.

**enumerator `Cast`**

Reports when a dynamic object is (or is not) resolved via a `CastAndCall` or other casting method.

**enumerator `UserVerboseFirst`**

The first in a range of logging levels reserved for code that uses VTK-m.

Internal VTK-m code will not log on these levels but will report these logs. These are used similarly to those in the `UserFirst` range but are at a lower precedence that also includes more verbose reporting from VTK-m.

**enumerator `UserVerboseLast`**

The last in a range of logging levels reserved for code that uses VTK-m.

When VTK-m outputs an entry in its log, it annotates the message with the logging level. VTK-m will automatically provide descriptions for all log levels described in `vtkm::cont::LogLevel`. A custom log level can be described by calling the `vtkm::cont::SetLogLevelName()` function. (The log name can likewise be retrieved with `vtkm::cont::GetLogLevelName()`.)

void `vtkm::cont::SetLogLevelName`(`vtkm::cont::LogLevel` level, const std::string &name)

Register a custom name to identify a log level.

The name will be truncated to 4 characters internally.

Must not be called after `InitLogging`. Such calls will fail and log an error.

There is no need to call this for the default `vtkm::cont::LogLevels`. They are populated in `InitLogging` and will be overwritten.

std::string `vtkm::cont::GetLogLevelName`(`vtkm::cont::LogLevel` level)

Get a human readable name for the log level.

If a name has not been registered via `InitLogging` or `SetLogLevelName`, the returned string just contains the integer representation of the level.

---

**Common Errors**

The `vtkm::cont::SetLogLevelName()` function must be called before `vtkm::cont::Initialize()` to have an effect.

---

---

**Common Errors**

The descriptions for each log level are only set up if `vtkm::cont::Initialize()` is called. If it is not, then all log levels will be represented with a numerical value.

---

If `vtkm::cont::Initialize()` is called with `argc/argv`, then the user can control the logging level with the `--vtkm-log-level` command line argument. Alternatively, you can control which logging levels are reported with the `vtkm::cont::SetStderrLogLevel()`.

`void vtkm::cont::SetStderrLogLevel(vtkm::cont::LogLevel level)`

Set the range of log levels that will be printed to stderr.

All levels with an enum value less-than-or-equal-to *level* will be printed.

`void vtkm::cont::SetStderrLogLevel(const char *verbosity)`

Set the range of log levels that will be printed to stderr.

All levels with an enum value less-than-or-equal-to *level* will be printed.

`vtkm::cont::LogLevel vtkm::cont::GetStderrLogLevel()`

Get the active highest log level that will be printed to stderr.

Example 1: Initializing logging.

```
1 static const vtkm::cont::LogLevel CustomLogLevel = vtkm::cont::LogLevel::UserFirst;
2
3 int main(int argc, char** argv)
4 {
5     vtkm::cont::SetLogLevelName(CustomLogLevel, "custom");
6
7     // For this example we will set the log level manually.
8     // The user can override this with the --vtkm-log-level command line flag.
9     vtkm::cont::SetStderrLogLevel(CustomLogLevel);
10
11     vtkm::cont::Initialize(argc, argv);
12
13     // Do interesting stuff...
```

## 21.3 Log Entries

Log entries are created with a collection of macros provided in `vtkm/cont/Logging.h`. In addition to basic log entries, VTK-m logging can also provide conditional logging and scope levels of logs.

### 21.3.1 Basic Log Entries

The main logging entry points are the macros `VTKM_LOG_S` and `VTKM_LOG_F`, which use C++ stream and printf syntax, respectively. Both macros take a logging level as the first argument. The remaining arguments specify the message printed to the log. `VTKM_LOG_S` takes a single argument with a C++ stream expression (so << operators can exist in the expression). `VTKM_LOG_F` takes a C string as its second argument that has printf-style formatting codes. The remaining arguments fulfill those codes.

**VTKM\_LOG\_S(level, ...)**

Writes a message using stream syntax to the indicated log *level*.

The ellipsis may be replaced with the log message as if constructing a C++ stream, e.g:

```
VTKM_LOG_S(vtkm::cont::LogLevel::Perf,
           "Executed functor " << vtkm::cont::TypeToString(functor)
           << " on device " << deviceId.GetName());
```

**VTKM\_LOG\_F**(level, ...)

Writes a message using printf syntax to the indicated log *level*.

The ellipsis may be replaced with the log message as if constructing a printf call, e.g:

```
VTKM_LOG_F(vtkm::cont::LogLevel::Perf,
           "Executed functor %s on device %s",
           vtkm::cont::TypeToString(functor).c_str(),
           deviceId.GetName().c_str());
```

Example 2: Basic logging.

```
1 VTKM_LOG_F(vtkm::cont::LogLevel::Info,
2           "Base VTK-m version: %d.%d",
3           VTKM_VERSION_MAJOR,
4           VTKM_VERSION_MINOR);
5 VTKM_LOG_S(vtkm::cont::LogLevel::Info, "Full VTK-m version: " << VTKM_VERSION_FULL);
```

### 21.3.2 Conditional Log Entries

The macros *VTKM\_LOG\_IF\_S* *VTKM\_LOG\_IF\_F* behave similarly to *VTKM\_LOG\_S* and *VTKM\_LOG\_F*, respectively, except they have an extra argument that contains the condition. If the condition is true, then the log entry is created. If the condition is false, then the statement is ignored and nothing is recorded in the log.

**VTKM\_LOG\_IF\_S**(level, cond, ...)

Same as *VTKM\_LOG\_S*, but only logs if *cond* is true.

**VTKM\_LOG\_IF\_F**(level, cond, ...)

Same as *VTKM\_LOG\_F*, but only logs if *cond* is true.

Example 3: Conditional logging.

```
1 for (vtkm::Id i = 0; i < 5; i++)
2 {
3     VTKM_LOG_IF_S(vtkm::cont::LogLevel::Info, i % 2 == 0, "Found an even number: " << i);
4 }
```

### 21.3.3 Scoped Log Entries

The logging back end supports the concept of scopes. Scopes allow the nesting of log messages, which allows a complex operation to report when it starts, when it ends, and what log messages happen in the middle. Scoped log entries are also timed so you can get an idea of how long operations take. Scoping can happen to arbitrary depths.

#### Common Errors

Although the timing reported in scoped log entries can give an idea of the time each operation takes, the reported time should not be considered accurate in regards to timing parallel operations. If a parallel algorithm is invoked inside a log scope, the program may return from that scope before the parallel algorithm is complete. See [Chapter 14 \(Timers\)](#) for information on more accurate timers.

Scoped log entries follow the same scoping of your C++ code. A scoped log can be created with the `VTKM_LOG_SCOPE` macro. This macro behaves similarly to `VTKM_LOG_F` except that it creates a scoped log that starts when `VTKM_LOG_SCOPE` and ends when the program leaves the given scope.

**VTKM\_LOG\_SCOPE**(level, ...)

Creates a new scope at the requested *level*.

The log scope ends when the code scope ends. The ellipses form the scope name using printf syntax.

```
{
    VTKM_LOG_SCOPE(vtkm::cont::LogLevel::Perf,
                  "Executing filter %s",
                  vtkm::cont::TypeToString(myFilter).c_str());
    myFilter.Execute();
}
```

Example 4: Scoped logging.

```
1  for (vtkm::IdComponent trial = 0; trial < numTrials; ++trial)
2  {
3      VTKM_LOG_SCOPE(CustomLogLevel, "Trial %d", trial);
4
5      VTKM_LOG_F(CustomLogLevel, "Do thing 1");
6
7      VTKM_LOG_F(CustomLogLevel, "Do thing 2");
8
9      //...
10 }
```

It is also common, and typically good code structure, to structure scoped concepts around functions or methods. Thus, VTK-m provides `VTKM_LOG_SCOPE_FUNCTION`. When placed at the beginning of a function or macro, `VTKM_LOG_SCOPE_FUNCTION` will automatically create a scoped log around it.

**VTKM\_LOG\_SCOPE\_FUNCTION**(level)

Equivalent to `VTKM_LOG_SCOPE(level, __func__)`

Example 5: Scoped logging in a function.

```
1  void TestFunc()
2  {
3      VTKM_LOG_SCOPE_FUNCTION(vtkm::cont::LogLevel::Info);
4      VTKM_LOG_S(vtkm::cont::LogLevel::Info, "Showcasing function logging");
5  }
```

## 21.4 Helper Functions

The `vtkm/cont/Logging.h` header file also contains several helper functions that provide useful functions when reporting information about the system.

---

### Did You Know?

Although provided with the logging utilities, these functions can be useful in contexts outside of the logging as well.

These functions are available even if VTK-m is compiled with logging off.

The `vtkm::cont::TypeToString()` function provides run-time type information (RTTI) based type-name information. `vtkm::cont::TypeToString()` is a templated function for which you have to explicitly declare the type. `vtkm::cont::TypeToString()` returns a `std::string` containing a representation of the type provided. When logging is enabled, `vtkm::cont::TypeToString()` uses the logging back end to demangle symbol names on supported platforms.

```
template<typename T>
```

```
inline std::string vtkm::cont::TypeToString()
```

Use RTTI information to retrieve the name of the type T.

If logging is enabled and the platform supports it, the type name will also be demangled.

```
template<typename T>
```

```
inline std::string vtkm::cont::TypeToString(const T&)
```

Use RTTI information to retrieve the name of the type T.

If logging is enabled and the platform supports it, the type name will also be demangled.

```
std::string vtkm::cont::TypeToString(const std::type_index &t)
```

Use RTTI information to retrieve the name of the type T.

If logging is enabled and the platform supports it, the type name will also be demangled.

```
std::string vtkm::cont::TypeToString(const std::type_info &t)
```

Use RTTI information to retrieve the name of the type T.

If logging is enabled and the platform supports it, the type name will also be demangled.

The `vtkm::cont::GetHumanReadableSize()` function takes a size of memory in bytes and returns a human readable string (for example “64 bytes”, “1.44 MiB”, “128 GiB”, etc). `vtkm::cont::GetSizeString()` is a similar function that returns the same thing as `vtkm::cont::GetHumanReadableSize()` followed by (# bytes) (with # replaced with the number passed to the function). Both `vtkm::cont::GetHumanReadableSize()` and `vtkm::cont::GetSizeString()` take an optional second argument that is the number of digits of precision to display. By default, they display 2 digits of precision.

```
std::string vtkm::cont::GetHumanReadableSize(vtkm::UInt64 bytes, int prec = 2)
```

Convert a size in bytes to a human readable string (such as “64 bytes”, “1.44 MiB”, “128 GiB”, etc).

*prec* controls the fixed point precision of the stringified number.

```
std::string vtkm::cont::GetSizeString(vtkm::UInt64 bytes, int prec = 2)
```

Returns “%1 (%2 bytes)” where %1 is the result from `GetHumanReadableSize` and %2 is the exact number of bytes.

The `vtkm::cont::GetStackTrace()` function returns a string containing a trace of the stack, which can be helpful for debugging. `vtkm::cont::GetStackTrace()` takes an optional argument for the number of stack frames to skip. Reporting the stack trace is not available on all platforms. On platforms that are not supported, a simple string reporting that the stack trace is unavailable is returned.

```
std::string vtkm::cont::GetStackTrace(vtkm::Int32 skip = 0)
```

Returns a stacktrace on supported platforms.

Argument is the number of frames to skip (GetStackTrace and below are already skipped).

Example 6: Helper functions provided for logging.

```
1  template<typename T>
2  void DoSomething(T&& x)
3  {
4      VTKM_LOG_S(CustomLogLevel,
5                  "Doing something with type " << vtkm::cont::TypeToString<T>());
6
7      vtkm::Id arraySize = 1000000 * sizeof(T);
8      VTKM_LOG_S(CustomLogLevel,
9                  "Size of array is " << vtkm::cont::GetHumanReadableSize(arraySize));
10     VTKM_LOG_S(CustomLogLevel,
11                 "More precisely it is " << vtkm::cont::GetSizeString(arraySize, 4));
12
13     VTKM_LOG_S(CustomLogLevel, "Stack location: " << vtkm::cont::GetStackTrace());
```



## WORKLET TYPES

Chapter 18 (Simple Worklets) introduces worklets and provides a simple example of creating a worklet to run an algorithm on a many core device. Different operations in visualization can have different data access patterns, perform different execution flow, and require different provisions. VTK-m manages these different accesses, execution, and provisions by grouping visualization algorithms into common classes of operation and supporting each class with its own worklet type.

Each worklet type has a generic superclass that worklets of that particular type must inherit. This makes the type of the worklet easy to identify. The following list describes each worklet type provided by VTK-m and the superclass that supports it.

- **Field Map** A worklet deriving `vtkm::worklet::WorkletMapField` performs a basic mapping operation that applies a function (the operator in the worklet) on all the field values at a single point or cell and creates a new field value at that same location. Although the intention is to operate on some variable over a mesh, a `vtkm::worklet::WorkletMapField` may actually be applied to any array. Thus, a field map can be used as a basic map operation.
- **Topology Map** A worklet deriving `vtkm::worklet::WorkletMapTopology` or one of its child classes performs a mapping operation that applies a function (the operator in the worklet) on all elements of a particular type (such as points or cells) and creates a new field for those elements. The basic operation is similar to a field map except that in addition to access fields being mapped on, the worklet operation also has access to incident fields.

There are multiple convenience classes available for the most common types of topology mapping. `vtkm::worklet::WorkletVisitCellsWithPoints` calls the worklet operation for each cell and makes every incident point available. This type of map also has access to cell structures and can interpolate point fields. Likewise, `vtkm::worklet::WorkletVisitPointsWithCells` calls the worklet operation for each point and makes every incident cell available.

- **Point Neighborhood** A worklet deriving from `vtkm::worklet::WorkletPointNeighborhood` performs a mapping operation that applies a function (the operator in the worklet) on all points of a structured mesh. The basic operation is similar to a field map except that in addition to having access to the point being operated on, you can get the field values of nearby points within a neighborhood of a given size. Point neighborhood worklets can only be applied to structured cell sets.
- **Reduce by Key** A worklet deriving `class vtkm::worklet::WorkletReduceByKey`` operates on an array of keys and one or more associated arrays of values. When a reduce by key worklet is invoked, all identical keys are collected and the worklet is called once for each unique key. Each worklet invocation is given a `Vec`-like containing all values associated with the unique key. Reduce by key worklets are very useful for combining like items such as shared topology elements or coincident points.

The remainder of this chapter provides details on how to create worklets of each type.

## 22.1 Field Map

A worklet deriving `vtkm::worklet::WorkletMapField` performs a basic mapping operation that applies a function (the operator in the worklet) on all the field values at a single point or cell and creates a new field value at that same location. Although the intention is to operate on some variable over the mesh, a `vtkm::worklet::WorkletMapField` can actually be applied to any array.

```
class WorkletMapField : public vtkm::worklet::internal::WorkletBase
```

Base class for worklets that do a simple mapping of field arrays.

All inputs and outputs are on the same domain. That is, all the arrays are the same size.

Subclassed by `vtkm::rendering::Triangulator::IndicesSort`, `vtkm::rendering::Triangulator::InterleaveArrays12`, `vtkm::rendering::Triangulator::InterleaveArrays2`, `vtkm::rendering::Triangulator::UniqueTriangles`, `vtkm::worklet::FieldStatistics< FieldType >::CalculatePowers`, `vtkm::worklet::FieldStatistics< FieldType >::SubtractConst`, `vtkm::worklet::KernelSplatterFilterUniformGrid< Kernel, DeviceAdapter >::ComputeLocalNeighborId`, `vtkm::worklet::KernelSplatterFilterUniformGrid< Kernel, DeviceAdapter >::GetFootprint`, `vtkm::worklet::KernelSplatterFilterUniformGrid< Kernel, DeviceAdapter >::GetSplatValue`, `vtkm::worklet::KernelSplatterFilterUniformGrid< Kernel, DeviceAdapter >::UpdateVoxelSplats`, `vtkm::worklet::KernelSplatterFilterUniformGrid< Kernel, DeviceAdapter >::zero_voxel`, `vtkm::worklet::Normal`, `vtkm::worklet::Normalize`, `vtkm::worklet::TriangleWinding::WorkletWindToCellNormals`, `vtkm::worklet::streamline::MakeStreamLines< FieldType >`

A field map worklet supports the following tags in the parameters of its `ControlSignature`.

```
struct FieldIn : public vtkm::cont::arg::ControlSignatureTagBase
```

`#include <WorkletMapField.h>` A control signature tag for input fields.

A `FieldIn` argument expects a `vtkm::cont::ArrayHandle` in the associated parameter of the invoke. Each invocation of the worklet gets a single value out of this array.

This tag means that the field is read only.

The worklet's `InputDomain` can be set to a `FieldIn` argument. In this case, the input domain will be the size of the array.

```
struct FieldOut : public vtkm::cont::arg::ControlSignatureTagBase
```

`#include <WorkletMapField.h>` A control signature tag for output fields.

A `FieldOut` argument expects a `vtkm::cont::ArrayHandle` in the associated parameter of the invoke. The array is resized before scheduling begins, and each invocation of the worklet sets a single value in the array.

This tag means that the field is write only.

Although uncommon, it is possible to set the worklet's `InputDomain` to a `FieldOut` argument. If this is the case, then the `vtkm::cont::ArrayHandle` passed as the argument must be allocated before being passed to the invoke, and the input domain will be the size of the array.

```
struct FieldInOut : public vtkm::cont::arg::ControlSignatureTagBase
```

`#include <WorkletMapField.h>` A control signature tag for input-output (in-place) fields.

A `FieldInOut` argument expects a `vtkm::cont::ArrayHandle` in the associated parameter of the invoke. Each invocation of the worklet gets a single value out of this array, which is replaced by the resulting value after the worklet completes.

This tag means that the field is read and write.

The worklet's `InputDomain` can be set to a `FieldInOut` argument. In this case, the input domain will be the size of the array.

```
struct WholeArrayIn : public vtkm::worklet::internal::WorkletBase::WholeArrayIn
    #include <WorkletMapField.h> ControlSignature tag for whole input arrays.
```

The `WholeArrayIn` control signature tag specifies a `vtkm::cont::ArrayHandle` passed to the invoke of the worklet. An array portal capable of reading from any place in the array is given to the worklet.

```
struct WholeArrayOut : public vtkm::worklet::internal::WorkletBase::WholeArrayOut
    #include <WorkletMapField.h> ControlSignature tag for whole output arrays.
```

The `WholeArrayOut` control signature tag specifies an `vtkm::cont::ArrayHandle` passed to the invoke of the worklet. An array portal capable of writing to any place in the array is given to the worklet. Developers should take care when using writable whole arrays as introducing race conditions is possible.

```
struct WholeArrayInOut : public vtkm::worklet::internal::WorkletBase::WholeArrayInOut
    #include <WorkletMapField.h> ControlSignature tag for whole input/output arrays.
```

The `WholeArrayOut` control signature tag specifies a `vtkm::cont::ArrayHandle` passed to the invoke of the worklet. An array portal capable of reading from or writing to any place in the array is given to the worklet. Developers should take care when using writable whole arrays as introducing race conditions is possible.

```
struct AtomicArrayInOut : public vtkm::worklet::internal::WorkletBase::AtomicArrayInOut
    #include <WorkletMapField.h> ControlSignature tag for whole input/output arrays.
```

The `AtomicArrayInOut` control signature tag specifies `vtkm::cont::ArrayHandle` passed to the invoke of the worklet. A `vtkm::exec::AtomicArray` object capable of performing atomic operations to the entries in the array is given to the worklet. Atomic arrays can help avoid race conditions but can slow down the running of a parallel algorithm.

```
template<typename VisitTopology = Cell, typename IncidentTopology = Point>
```

```
struct WholeCellSetIn : public vtkm::worklet::internal::WorkletBase::WholeCellSetIn<Cell, Point>
    #include <WorkletMapField.h> ControlSignature tag for whole input topology.
```

The `WholeCellSetIn` control signature tag specifies a `vtkm::cont::CellSet` passed to the invoke of the worklet. A connectivity object capable of finding elements of one type that are incident on elements of a different type. This can be used to global lookup for arbitrary topology information

```
struct ExecObject : public vtkm::worklet::internal::WorkletBase::ExecObject
    #include <WorkletMapField.h> ControlSignature tag for execution object inputs.
```

This tag represents an execution object that is passed directly from the control environment to the worklet. A `ExecObject` argument expects a subclass of `vtkm::exec::ExecutionObjectBase`. Subclasses of `vtkm::exec::ExecutionObjectBase` behave like a factory for objects that work on particular devices. They do this by implementing a `PrepareForExecution()` method that takes a device adapter tag and returns an object that works on that device. That device-specific object is passed directly to the worklet.

Furthermore, a field map worklet supports the following tags in the parameters of its `ExecutionSignature`.

```
struct _1 : public vtkm::placeholders::Arg<1>
    #include <WorkletMapField.h> Argument placeholders for an ExecutionSignature.
```

All worklet superclasses declare numeric tags in the form of `_1`, `_2`, `_3` etc. that are used in the `ExecutionSignature` to refer to the corresponding parameter in the `ControlSignature`.

struct **WorkIndex** : public vtkm::exec::arg::WorkIndex

*#include <WorkletMapField.h>* The `ExecutionSignature` tag to use to get the work index.

This tag produces a `vtkm::Id` that uniquely identifies the invocation instance of the worklet. When a worklet is dispatched, it broken into pieces defined by the input domain and scheduled on independent threads. This tag in the `ExecutionSignature` passes the index for this work.

struct **VisitIndex** : public vtkm::exec::arg::VisitIndex

*#include <WorkletMapField.h>* The `ExecutionSignature` tag to use to get the visit index.

This tag produces a `vtkm::IdComponent` that uniquely identifies when multiple worklet invocations operate on the same input item, which can happen when defining a worklet with scatter.

When a worklet is dispatched, there is a scatter operation defined that optionally allows each input to go to multiple output entries. When one input is assigned to multiple outputs, there needs to be a mechanism to uniquely identify which output is which. The visit index is a value between 0 and the number of outputs a particular input goes to. This tag in the `ExecutionSignature` passes the visit index for this work.

struct **InputIndex** : public vtkm::exec::arg::InputIndex

*#include <WorkletMapField.h>* The `ExecutionSignature` tag to use to get the input index.

This tag produces a `vtkm::Id` that identifies the index of the input element, which can differ from the `WorkIndex` in a worklet with a scatter.

When a worklet is dispatched, it broken into pieces defined by the input domain and scheduled on independent threads. This tag in the `ExecutionSignature` passes the index of the input element that the work thread is currently working on. When a worklet has a scatter associated with it, the input and output indices can be different.

struct **OutputIndex** : public vtkm::exec::arg::OutputIndex

*#include <WorkletMapField.h>* The `ExecutionSignature` tag to use to get the output index.

This tag produces a `vtkm::Id` that identifies the index of the output element. (This is generally the same as `WorkIndex`.)

When a worklet is dispatched, it broken into pieces defined by the output domain and scheduled on independent threads. This tag in the `ExecutionSignature` passes the index of the output element that the work thread is currently working on. When a worklet has a scatter associated with it, the output and output indices can be different.

struct **ThreadIndices** : public vtkm::exec::arg::ThreadIndices

*#include <WorkletMapField.h>* The `ExecutionSignature` tag to use to get the thread indices.

This tag produces an internal object that manages indices and other metadata of the current thread. Thread indices objects vary by worklet type, but most users can get the information they need through other signature tags.

When a worklet is dispatched, it broken into pieces defined by the input domain and scheduled on independent threads. During this process multiple indices associated with the input and output can be generated. This tag in the `ExecutionSignature` passes the index for this work.

struct **Device** : public vtkm::worklet::internal::WorkletBase::Device

*#include <WorkletMapField.h>* `ExecutionSignature` tag for getting the device adapter tag.

This tag passes a device adapter tag object. This allows the worklet function to template on or overload itself based on the type of device that it is being executed on.

Field maps most commonly perform basic calculator arithmetic, as demonstrated in the following example.

Example 1: Implementation and use of a field map worklet.

```

1 class ComputeMagnitude : public vtkm::worklet::WorkletMapField
2 {
3 public:
4     using ControlSignature = void(FieldIn inputVectors, FieldOut outputMagnitudes);
5     using ExecutionSignature = _2(_1);
6
7     using InputDomain = _1;
8
9     template<typename T, vtkm::IdComponent Size>
10     VTKM_EXEC T operator()(const vtkm::Vec<T, Size>& inVector) const
11     {
12         return vtkm::Magnitude(inVector);
13     }
14 };

```

Although simple, the `vtkm::worklet::WorkletMapField` worklet type can be used (and abused) as a general parallel-for/scheduling mechanism. In particular, the `WorkIndex` execution signature tag can be used to get a unique index, the `WholeArray*` tags can be used to get random access to arrays, and the `ExecObject` control signature tag can be used to pass execution objects directly to the worklet. Whole arrays and execution objects are talked about in more detail in [Chapters `ref{chap:Globals}`](#) and [ref{chap:ExecutionObjects}](#), respectively, in more detail, but here is a simple example that uses the random access of `:class`WholeArrayOut`` to make a worklet that copies an array in reverse order.

Example 2: Leveraging field maps and field maps for general processing.

```

1 namespace vtkm
2 {
3     namespace worklet
4     {
5
6         struct ReverseArrayCopyWorklet : vtkm::worklet::WorkletMapField
7         {
8             using ControlSignature = void(FieldIn inputArray, WholeArrayOut outputArray);
9             using ExecutionSignature = void(_1, _2, WorkIndex);
10            using InputDomain = _1;
11
12            template<typename InputType, typename OutputArrayPortalType>
13            VTKM_EXEC void operator()(const InputType& inputValue,
14                                     const OutputArrayPortalType& outputArrayPortal,
15                                     vtkm::Id workIndex) const
16            {
17                vtkm::Id outIndex = outputArrayPortal.GetNumberOfValues() - workIndex - 1;
18                if (outIndex >= 0)
19                {
20                    outputArrayPortal.Set(outIndex, inputValue);
21                }
22                else
23                {
24                    this->RaiseError("Output array not sized correctly.");
25                }
26            }
27        }
28    }
29 }

```

(continues on next page)

(continued from previous page)

```

26     }
27 };
28
29 } // namespace worklet
30 } // namespace vtkm

```

## 22.2 Topology Map

A topology map performs a mapping that it applies a function (the operator in the worklet) on all the elements of a `vtkm::cont::DataSet` of a particular type (i.e. point, edge, face, or cell). While operating on the element, the worklet has access to data from all incident elements of another type.

There are several versions of topology maps that differ in what type of element being mapped from and what type of element being mapped to. The subsequent sections describe these different variations of the topology maps.

### 22.2.1 Visit Cells with Points

A worklet deriving `vtkm::worklet::WorkletVisitCellsWithPoints` performs a mapping operation that applies a function (the operator in the worklet) on all the cells of a `vtkm::cont::DataSet`. While operating on the cell, the worklet has access to fields associated both with the cell and with all incident points. Additionally, the worklet can get information about the structure of the cell and can perform operations like interpolation on it.

class **WorkletVisitCellsWithPoints** : public

`vtkm::worklet::WorkletMapTopology<vtkm::TopologyElementTagCell, vtkm::TopologyElementTagPoint>`

Base class for worklets that map from Points to Cells.

Subclassed by `vtkm::cont::internal::RConnTableHelpers::WriteConnectivity`, `vtkm::cont::internal::RConnTableHelpers::WriteNumIndices`, `vtkm::rendering::Cylinderizer::CountSegments`, `vtkm::rendering::Cylinderizer::Cylinderize`, `vtkm::rendering::Cylinderizer::SegmentedStructured< DIM >`, `vtkm::rendering::Quadralizer::CountQuads`, `vtkm::rendering::Quadralizer::Quadralize`, `vtkm::rendering::Quadralizer::SegmentedStructured< DIM >`, `vtkm::rendering::Triangulator::CountTriangles`, `vtkm::rendering::Triangulator::Triangulate`, `vtkm::rendering::Triangulator::TriangulateStructured< DIM >`, `vtkm::worklet::CellDeepCopy::CountCellPoints`, `vtkm::worklet::CellDeepCopy::PassCellStructure`, `vtkm::worklet::TriangleWinding::WorkletGetCellShapesAndSizes`, `vtkm::worklet::TriangleWinding::WorkletWindToCellNormal`

A visit cells with points worklet supports the following tags in the parameters of its `ControlSignature`.

struct **CellSetIn** : public `vtkm::worklet::WorkletMapTopology<VisitTopology, IncidentTopology>::CellSetIn`

*#include <WorkletMapTopology.h>* A control signature tag for input connectivity.

The associated parameter of the invoke should be a subclass of `vtkm::cont::CellSet`.

There should be exactly one `CellSetIn` argument in the `ControlSignature`, and the `InputDomain` must point to it.

struct **FieldInCell** : public `vtkm::worklet::WorkletVisitCellsWithPoints::FieldInVisit`

*#include <WorkletMapTopology.h>* A control signature tag for input fields on the cells of the topology.

The associated parameter of the invoke should be a `vtkm::cont::ArrayHandle` that has the same number of values as the cells of the provided `CellSet`. The worklet gets a single value that is the field at that cell.



struct **FieldInPoint** : public vtkm::worklet::WorkletVisitCellsWithPoints::FieldInIncident

*#include <WorkletMapTopology.h>* A control signature tag for input fields on the points of the topology.

The associated parameter of the invoke should be a *vtkm::cont::ArrayHandle* that has the same number of values as the points of the provided CellSet. The worklet gets a Vec-like object containing the field values on all incident points.

struct **FieldInVisit** : public vtkm::worklet::WorkletMapTopology<VisitTopology, IncidentTopology>::FieldInVisit

*#include <WorkletMapTopology.h>* A control signature tag for input fields from the visited topology.

For *WorkletVisitCellsWithPoints*, this is the same as *FieldInCell*.

Subclassed by *vtkm::worklet::WorkletVisitCellsWithPoints::FieldInCell*

struct **FieldInIncident** : public vtkm::worklet::WorkletMapTopology<VisitTopology, IncidentTopology>::FieldInIncident

*#include <WorkletMapTopology.h>* A control signature tag for input fields from the incident topology.

For *WorkletVisitCellsWithPoints*, this is the same as *FieldInPoint*.

Subclassed by *vtkm::worklet::WorkletVisitCellsWithPoints::FieldInPoint*

struct **FieldOutCell** : public vtkm::worklet::WorkletVisitCellsWithPoints::FieldOut

*#include <WorkletMapTopology.h>* A control signature tag for output fields.

A *WorkletVisitCellsWithPoints* always has the output on the cells of the topology. The associated parameter of the invoke should be a *vtkm::cont::ArrayHandle*, and it will be resized to the number of cells in the provided CellSet.

struct **FieldOut** : public vtkm::worklet::WorkletMapTopology<VisitTopology, IncidentTopology>::FieldOut

*#include <WorkletMapTopology.h>* A control signature tag for output fields.

A *WorkletVisitCellsWithPoints* always has the output on the cells of the topology. The associated parameter of the invoke should be a *vtkm::cont::ArrayHandle*, and it will be resized to the number of cells in the provided CellSet.

Subclassed by *vtkm::worklet::WorkletVisitCellsWithPoints::FieldOutCell*

struct **FieldInOutCell** : public vtkm::worklet::WorkletVisitCellsWithPoints::FieldInOut

*#include <WorkletMapTopology.h>* A control signature tag for input-output (in-place) fields.

A *WorkletVisitCellsWithPoints* always has the output on the cells of the topology. The associated parameter of the invoke should be a *vtkm::cont::ArrayHandle*, and it must have the same number of values as the number of cells of the topology.

struct **FieldInOut** : public vtkm::worklet::WorkletMapTopology<VisitTopology, IncidentTopology>::FieldInOut

*#include <WorkletMapTopology.h>* A control signature tag for input-output (in-place) fields.

A *WorkletVisitCellsWithPoints* always has the output on the cells of the topology. The associated parameter of the invoke should be a *vtkm::cont::ArrayHandle*, and it must have the same number of values as the number of cells of the topology.

Subclassed by *vtkm::worklet::WorkletVisitCellsWithPoints::FieldInOutCell*

struct **WholeArrayIn** : public vtkm::worklet::internal::WorkletBase::WholeArrayIn

*#include <WorkletMapTopology.h>* ControlSignature tag for whole input arrays.

The *WholeArrayIn* control signature tag specifies a *vtkm::cont::ArrayHandle* passed to the invoke of the worklet. An array portal capable of reading from any place in the array is given to the worklet.

struct **WholeArrayOut** : public vtkm::worklet::internal::WorkletBase::WholeArrayOut

*#include <WorkletMapTopology.h>* ControlSignature tag for whole output arrays.

The *WholeArrayOut* control signature tag specifies an *vtkm::cont::ArrayHandle* passed to the invoke of the worklet. An array portal capable of writing to any place in the array is given to the worklet. Developers should take care when using writable whole arrays as introducing race conditions is possible.

struct **WholeArrayInOut** : public vtkm::worklet::internal::WorkletBase::WholeArrayInOut

*#include <WorkletMapTopology.h>* ControlSignature tag for whole input/output arrays.

The *WholeArrayOut* control signature tag specifies a *vtkm::cont::ArrayHandle* passed to the invoke of the worklet. An array portal capable of reading from or writing to any place in the array is given to the worklet. Developers should take care when using writable whole arrays as introducing race conditions is possible.

struct **AtomicArrayInOut** : public vtkm::worklet::internal::WorkletBase::AtomicArrayInOut

*#include <WorkletMapTopology.h>* ControlSignature tag for whole input/output arrays.

The *AtomicArrayInOut* control signature tag specifies *vtkm::cont::ArrayHandle* passed to the invoke of the worklet. A *vtkm::exec::AtomicArray* object capable of performing atomic operations to the entries in the array is given to the worklet. Atomic arrays can help avoid race conditions but can slow down the running of a parallel algorithm.

template<typename **VisitTopology** = Cell, typename **IncidentTopology** = Point>

struct **WholeCellSetIn** : public vtkm::worklet::internal::WorkletBase::WholeCellSetIn<Cell, Point>

*#include <WorkletMapTopology.h>* ControlSignature tag for whole input topology.

The *WholeCellSetIn* control signature tag specifies a *vtkm::cont::CellSet* passed to the invoke of the worklet. A connectivity object capable of finding elements of one type that are incident on elements of a different type. This can be used to global lookup for arbitrary topology information

struct **ExecObject** : public vtkm::worklet::internal::WorkletBase::ExecObject

*#include <WorkletMapTopology.h>* ControlSignature tag for execution object inputs.

This tag represents an execution object that is passed directly from the control environment to the worklet. A *ExecObject* argument expects a subclass of *vtkm::exec::ExecutionObjectBase*. Subclasses of *vtkm::exec::ExecutionObjectBase* behave like a factory for objects that work on particular devices. They do this by implementing a *PrepareForExecution()* method that takes a device adapter tag and returns an object that works on that device. That device-specific object is passed directly to the worklet.

A visit cells with points worklet supports the following tags in the parameters of its *ExecutionSignature*.

struct **\_1** : public vtkm::placeholders::Arg<1>

*#include <WorkletMapTopology.h>* Argument placeholders for an *ExecutionSignature*.

All worklet superclasses declare numeric tags in the form of *\_1*, *\_2*, *\_3* etc. that are used in the *ExecutionSignature* to refer to the corresponding parameter in the *ControlSignature*.



struct **CellShape** : public vtkm::worklet::WorkletMapTopology<VisitTopology, IncidentTopology>::CellShape  
*#include <WorkletMapTopology.h>* An execution signature tag to get the shape of the visited cell.

This tag causes a *vtkm::UInt8* to be passed to the worklet containing an id for the shape of the cell being visited.

struct **PointCount** : public vtkm::worklet::WorkletMapTopology<VisitTopology, IncidentTopology>::IncidentElementCount

*#include <WorkletMapTopology.h>* An execution signature tag to get the number of incident points.

Each cell in a *vtkm::cont::CellSet* can be incident on a number of points. This tag causes a *vtkm::IdComponent* to be passed to the worklet containing the number of incident points.

struct **PointIndices** : public vtkm::worklet::WorkletMapTopology<VisitTopology, IncidentTopology>::IncidentElementIndices

*#include <WorkletMapTopology.h>* An execution signature tag to get the indices of the incident points.

The indices will be provided in a Vec-like object containing *vtkm::Id* indices for the cells in the data set.

struct **WorkIndex** : public vtkm::exec::arg::WorkIndex

*#include <WorkletMapTopology.h>* The *ExecutionSignature* tag to use to get the work index.

This tag produces a *vtkm::Id* that uniquely identifies the invocation instance of the worklet. When a worklet is dispatched, it broken into pieces defined by the input domain and scheduled on independent threads. This tag in the *ExecutionSignature* passes the index for this work.

struct **VisitIndex** : public vtkm::exec::arg::VisitIndex

*#include <WorkletMapTopology.h>* The *ExecutionSignature* tag to use to get the visit index.

This tag produces a *vtkm::IdComponent* that uniquely identifies when multiple worklet invocations operate on the same input item, which can happen when defining a worklet with scatter.

When a worklet is dispatched, there is a scatter operation defined that optionally allows each input to go to multiple output entries. When one input is assigned to multiple outputs, there needs to be a mechanism to uniquely identify which output is which. The visit index is a value between 0 and the number of outputs a particular input goes to. This tag in the *ExecutionSignature* passes the visit index for this work.

struct **InputIndex** : public vtkm::exec::arg::InputIndex

*#include <WorkletMapTopology.h>* The *ExecutionSignature* tag to use to get the input index.

This tag produces a *vtkm::Id* that identifies the index of the input element, which can differ from the *WorkIndex* in a worklet with a scatter.

When a worklet is dispatched, it broken into pieces defined by the input domain and scheduled on independent threads. This tag in the *ExecutionSignature* passes the index of the input element that the work thread is currently working on. When a worklet has a scatter associated with it, the input and output indices can be different.

struct **OutputIndex** : public vtkm::exec::arg::OutputIndex

*#include <WorkletMapTopology.h>* The *ExecutionSignature* tag to use to get the output index.

This tag produces a *vtkm::Id* that identifies the index of the output element. (This is generally the same as *WorkIndex*.)

When a worklet is dispatched, it is broken into pieces defined by the output domain and scheduled on independent threads. This tag in the `ExecutionSignature` passes the index of the output element that the work thread is currently working on. When a worklet has a scatter associated with it, the output and output indices can be different.

struct **ThreadIndices** : public `vtkm::exec::arg::ThreadIndices`

*#include <WorkletMapTopology.h>* The `ExecutionSignature` tag to use to get the thread indices.

This tag produces an internal object that manages indices and other metadata of the current thread. Thread indices objects vary by worklet type, but most users can get the information they need through other signature tags.

When a worklet is dispatched, it is broken into pieces defined by the input domain and scheduled on independent threads. During this process multiple indices associated with the input and output can be generated. This tag in the `ExecutionSignature` passes the index for this work.

struct **Device** : public `vtkm::worklet::internal::WorkletBase::Device`

*#include <WorkletMapTopology.h>* `ExecutionSignature` tag for getting the device adapter tag.

This tag passes a device adapter tag object. This allows the worklet function to template on or overload itself based on the type of device that it is being executed on.

Point to cell field maps are a powerful construct that allow you to interpolate point fields throughout the space of the data set. See [Chapter 26 \(Working with Cells\)](#) for a description on how to work with the cell information provided to the worklet. The following example provides a simple demonstration that finds the geometric center of each cell by interpolating the point coordinates to the cell centers.

Example 3: Implementation and use of a visit cells with points worklet.

```

1 namespace vtkm
2 {
3   namespace worklet
4   {
5
6     struct CellCenter : public vtkm::worklet::WorkletVisitCellsWithPoints
7     {
8     public:
9       using ControlSignature = void(CellSetIn cellSet,
10                                     FieldInPoint inputPointField,
11                                     FieldOut outputCellField);
12       using ExecutionSignature = void(_1, PointCount, _2, _3);
13
14       using InputDomain = _1;
15
16       template<typename CellShape, typename InputPointFieldType, typename OutputType>
17       VTKM_EXEC void operator()(CellShape shape,
18                                vtkm::IdComponent numPoints,
19                                const InputPointFieldType& inputPointField,
20                                OutputType& centerOut) const
21       {
22         vtkm::Vec3f parametricCenter;
23         vtkm::exec::ParametricCoordinatesCenter(numPoints, shape, parametricCenter);
24         vtkm::exec::CellInterpolate(inputPointField, parametricCenter, shape, centerOut);
25       }
26     };

```

(continues on next page)

(continued from previous page)

```

27
28 } // namespace worklet
29 } // namespace vtkm

```

### 22.2.2 Visit Points with Cells

A worklet deriving `vtkm::worklet::WorkletVisitPointsWithCells` performs a mapping operation that applies a function (the operator in the worklet) on all the points of a `vtkm::cont::DataSet`. While operating on the point, the worklet has access to fields associated both with the point and with all incident cells.

```
class WorkletVisitPointsWithCells : public
vtkm::worklet::WorkletMapTopology<vtkm::TopologyElementTagPoint, vtkm::TopologyElementTagCell>
```

Base class for worklets that map from Cells to Points.

A visit points with cells worklet supports the following tags in the parameters of its `ControlSignature`.

```
struct CellSetIn : public vtkm::worklet::WorkletMapTopology<VisitTopology, IncidentTopology>::CellSetIn
```

*#include <WorkletMapTopology.h>* A control signature tag for input connectivity.

The associated parameter of the invoke should be a subclass of `vtkm::cont::CellSet`.

There should be exactly one `CellSetIn` argument in the `ControlSignature`, and the `InputDomain` must point to it.

```
struct FieldInPoint : public vtkm::worklet::WorkletVisitPointsWithCells::FieldInVisit
```

*#include <WorkletMapTopology.h>* A control signature tag for input fields on the points of the topology.

The associated parameter of the invoke should be a `vtkm::cont::ArrayHandle` that has the same number of values as the points of the provided `CellSet`. The worklet gets a single value that is the field at that point.

```
struct FieldInCell : public vtkm::worklet::WorkletVisitPointsWithCells::FieldInIncident
```

*#include <WorkletMapTopology.h>* A control signature tag for input fields on the cells of the topology.

The associated parameter of the invoke should be a `vtkm::cont::ArrayHandle` that has the same number of values as the cells of the provided `CellSet`. The worklet gets a Vec-like object containing the field values on all incident cells.

```
struct FieldInVisit : public vtkm::worklet::WorkletMapTopology<VisitTopology,
IncidentTopology>::FieldInVisit
```

*#include <WorkletMapTopology.h>* A control signature tag for input fields from the visited topology.

For `WorkletVisitPointsWithCells`, this is the same as `FieldInPoint`.

Subclassed by `vtkm::worklet::WorkletVisitPointsWithCells::FieldInPoint`

```
struct FieldInIncident : public vtkm::worklet::WorkletMapTopology<VisitTopology,
IncidentTopology>::FieldInIncident
```

*#include <WorkletMapTopology.h>* A control signature tag for input fields from the incident topology.

For `WorkletVisitPointsWithCells`, this is the same as `FieldInCell`.

Subclassed by `vtkm::worklet::WorkletVisitPointsWithCells::FieldInCell`

struct **FieldOutPoint** : public vtkm::worklet::WorkletVisitPointsWithCells::FieldOut

*#include <WorkletMapTopology.h>* A control signature tag for output fields.

A *WorkletVisitPointsWithCells* always has the output on the points of the topology. The associated parameter of the invoke should be a *vtkm::cont::ArrayHandle*, and it will be resized to the number of points in the provided *CellSet*.

struct **FieldOut** : public vtkm::worklet::WorkletMapTopology<VisitTopology, IncidentTopology>::FieldOut

*#include <WorkletMapTopology.h>* A control signature tag for output fields.

A *WorkletVisitPointsWithCells* always has the output on the points of the topology. The associated parameter of the invoke should be a *vtkm::cont::ArrayHandle*, and it will be resized to the number of points in the provided *CellSet*.

Subclassed by *vtkm::worklet::WorkletVisitPointsWithCells::FieldOutPoint*

struct **FieldInOutPoint** : public vtkm::worklet::WorkletVisitPointsWithCells::FieldInOut

*#include <WorkletMapTopology.h>* A control signature tag for input-output (in-place) fields.

A *WorkletVisitPointsWithCells* always has the output on the points of the topology. The associated parameter of the invoke should be a *vtkm::cont::ArrayHandle*, and it must have the same number of values as the number of points of the topology.

struct **FieldInOut** : public vtkm::worklet::WorkletMapTopology<VisitTopology, IncidentTopology>::FieldInOut

*#include <WorkletMapTopology.h>* A control signature tag for input-output (in-place) fields.

A *WorkletVisitPointsWithCells* always has the output on the points of the topology. The associated parameter of the invoke should be a *vtkm::cont::ArrayHandle*, and it must have the same number of values as the number of points of the topology.

Subclassed by *vtkm::worklet::WorkletVisitPointsWithCells::FieldInOutPoint*

struct **WholeArrayIn** : public vtkm::worklet::internal::WorkletBase::WholeArrayIn

*#include <WorkletMapTopology.h>* ControlSignature tag for whole input arrays.

The *WholeArrayIn* control signature tag specifies a *vtkm::cont::ArrayHandle* passed to the invoke of the worklet. An array portal capable of reading from any place in the array is given to the worklet.

struct **WholeArrayOut** : public vtkm::worklet::internal::WorkletBase::WholeArrayOut

*#include <WorkletMapTopology.h>* ControlSignature tag for whole output arrays.

The *WholeArrayOut* control signature tag specifies an *vtkm::cont::ArrayHandle* passed to the invoke of the worklet. An array portal capable of writing to any place in the array is given to the worklet. Developers should take care when using writable whole arrays as introducing race conditions is possible.

struct **WholeArrayInOut** : public vtkm::worklet::internal::WorkletBase::WholeArrayInOut

*#include <WorkletMapTopology.h>* ControlSignature tag for whole input/output arrays.

The *WholeArrayOut* control signature tag specifies a *vtkm::cont::ArrayHandle* passed to the invoke of the worklet. An array portal capable of reading from or writing to any place in the array is given to the worklet. Developers should take care when using writable whole arrays as introducing race conditions is possible.

```
struct AtomicArrayInOut : public vtkm::worklet::internal::WorkletBase::AtomicArrayInOut
```

*#include <WorkletMapTopology.h>* ControlSignature tag for whole input/output arrays.

The *AtomicArrayInOut* control signature tag specifies *vtkm::cont::ArrayHandle* passed to the invoke of the worklet. A *vtkm::exec::AtomicArray* object capable of performing atomic operations to the entries in the array is given to the worklet. Atomic arrays can help avoid race conditions but can slow down the running of a parallel algorithm.

```
template<typename VisitTopology = Cell, typename IncidentTopology = Point>
```

```
struct WholeCellSetIn : public vtkm::worklet::internal::WorkletBase::WholeCellSetIn<Cell, Point>
```

*#include <WorkletMapTopology.h>* ControlSignature tag for whole input topology.

The *WholeCellSetIn* control signature tag specifies a *vtkm::cont::CellSet* passed to the invoke of the worklet. A connectivity object capable of finding elements of one type that are incident on elements of a different type. This can be used to global lookup for arbitrary topology information

```
struct ExecObject : public vtkm::worklet::internal::WorkletBase::ExecObject
```

*#include <WorkletMapTopology.h>* ControlSignature tag for execution object inputs.

This tag represents an execution object that is passed directly from the control environment to the worklet. A *ExecObject* argument expects a subclass of *vtkm::exec::ExecutionObjectBase*. Subclasses of *vtkm::exec::ExecutionObjectBase* behave like a factory for objects that work on particular devices. They do this by implementing a *PrepareForExecution()* method that takes a device adapter tag and returns an object that works on that device. That device-specific object is passed directly to the worklet.

A visit points with cells worklet supports the following tags in the parameters of its *ExecutionSignature*.

```
struct _1 : public vtkm::placeholders::Arg<1>
```

*#include <WorkletMapTopology.h>* Argument placeholders for an *ExecutionSignature*.

All worklet superclasses declare numeric tags in the form of *\_1*, *\_2*, *\_3* etc. that are used in the *ExecutionSignature* to refer to the corresponding parameter in the *ControlSignature*.

```
struct CellCount : public vtkm::worklet::WorkletMapTopology<VisitTopology,  
IncidentTopology>::IncidentElementCount
```

*#include <WorkletMapTopology.h>* An execution signature tag to get the number of incident cells.

Each point in a *vtkm::cont::CellSet* can be incident on a number of cells. This tag causes a *vtkm::IdComponent* to be passed to the worklet containing the number of incident cells.

```
struct CellIndices : public vtkm::worklet::WorkletMapTopology<VisitTopology,  
IncidentTopology>::IncidentElementIndices
```

*#include <WorkletMapTopology.h>* An execution signature tag to get the indices of the incident cells.

The indices will be provided in a Vec-like object containing *vtkm::Id* indices for the points in the data set.

```
struct WorkIndex : public vtkm::exec::arg::WorkIndex
```

*#include <WorkletMapTopology.h>* The *ExecutionSignature* tag to use to get the work index.

This tag produces a *vtkm::Id* that uniquely identifies the invocation instance of the worklet. When a worklet is dispatched, it broken into pieces defined by the input domain and scheduled on independent threads. This tag in the *ExecutionSignature* passes the index for this work.

struct **VisitIndex** : public vtkm::exec::arg::VisitIndex

*#include <WorkletMapTopology.h>* The ExecutionSignature tag to use to get the visit index.

This tag produces a [vtkm::IdComponent](#) that uniquely identifies when multiple worklet invocations operate on the same input item, which can happen when defining a worklet with scatter.

When a worklet is dispatched, there is a scatter operation defined that optionally allows each input to go to multiple output entries. When one input is assigned to multiple outputs, there needs to be a mechanism to uniquely identify which output is which. The visit index is a value between 0 and the number of outputs a particular input goes to. This tag in the ExecutionSignature passes the visit index for this work.

struct **InputIndex** : public vtkm::exec::arg::InputIndex

*#include <WorkletMapTopology.h>* The ExecutionSignature tag to use to get the input index.

This tag produces a [vtkm::Id](#) that identifies the index of the input element, which can differ from the [WorkIndex](#) in a worklet with a scatter.

When a worklet is dispatched, it broken into pieces defined by the input domain and scheduled on independent threads. This tag in the ExecutionSignature passes the index of the input element that the work thread is currently working on. When a worklet has a scatter associated with it, the input and output indices can be different.

struct **OutputIndex** : public vtkm::exec::arg::OutputIndex

*#include <WorkletMapTopology.h>* The ExecutionSignature tag to use to get the output index.

This tag produces a [vtkm::Id](#) that identifies the index of the output element. (This is generally the same as [WorkIndex](#).)

When a worklet is dispatched, it broken into pieces defined by the output domain and scheduled on independent threads. This tag in the ExecutionSignature passes the index of the output element that the work thread is currently working on. When a worklet has a scatter associated with it, the output and output indices can be different.

struct **ThreadIndices** : public vtkm::exec::arg::ThreadIndices

*#include <WorkletMapTopology.h>* The ExecutionSignature tag to use to get the thread indices.

This tag produces an internal object that manages indices and other metadata of the current thread. Thread indices objects vary by worklet type, but most users can get the information they need through other signature tags.

When a worklet is dispatched, it broken into pieces defined by the input domain and scheduled on independent threads. During this process multiple indices associated with the input and output can be generated. This tag in the ExecutionSignature passes the index for this work.

struct **Device** : public vtkm::worklet::internal::WorkletBase::Device

*#include <WorkletMapTopology.h>* ExecutionSignature tag for getting the device adapter tag.

This tag passes a device adapter tag object. This allows the worklet function to template on or overload itself based on the type of device that it is being executed on.

Cell to point field maps are typically used for converting fields associated with cells to points so that they can be interpolated. The following example does a simple averaging, but you can also implement other strategies such as a volume weighted average.

Example 4: Implementation and use of a visit points with cells worklet.

```

1 class AverageCellField : public vtkm::worklet::WorkletVisitPointsWithCells
2 {
3 public:
4     using ControlSignature = void(CellSetIn cellSet,
5                                   FieldInCell inputCellField,
6                                   FieldOut outputPointField);
7     using ExecutionSignature = void(CellCount, _2, _3);
8
9     using InputDomain = _1;
10
11     template<typename InputCellFieldType, typename OutputFieldType>
12     VTKM_EXEC void operator()(vtkm::IdComponent numCells,
13                               const InputCellFieldType& inputCellField,
14                               OutputFieldType& fieldAverage) const
15     {
16         fieldAverage = OutputFieldType(0);
17
18         for (vtkm::IdComponent cellIndex = 0; cellIndex < numCells; cellIndex++)
19         {
20             fieldAverage = fieldAverage + inputCellField[cellIndex];
21         }
22
23         fieldAverage = fieldAverage / OutputFieldType(numCells);
24     }
25 };
26
27 //
28 // Later in the associated Filter class...
29 //
30
31 vtkm::cont::ArrayHandle<T> outFieldData;
32 this->Invoke(AverageCellField{}, inCellSet, inFieldData, outFieldData);

```

## 22.3 Neighborhood Mapping

VTK-m provides a pair of worklets that allow easy access to data within a neighborhood of nearby elements. This simplifies operations like smoothing a field by blending each value with that of its neighbors. This can only be done on data sets with `vtkm::cont::CellSetStructured` cell sets where extended adjacencies are easy to find. There are two flavors of the worklet: a point neighborhood worklet and a cell neighborhood worklet.



### 22.3.1 Point Neighborhood

A worklet deriving `vtkm::worklet::WorkletPointNeighborhood` performs a mapping operation that applies a function (the operator in the worklet) on all the points of a `vtkm::cont::DataSet`. While operating on the point, the worklet has access to field values on nearby points within a neighborhood.

class **WorkletPointNeighborhood** : public `vtkm::worklet::WorkletNeighborhood`

Base class for worklets that map over the points in a structured grid with neighborhood information.

The domain of a `WorkletPointNeighborhood` is a `vtkm::cont::CellSetStructured`. It visits all the points in the mesh and provides access to the point field values of the visited point and the field values of the nearby connected neighborhood of a prescribed size.

Subclassed by `vtkm::worklet::AveragePointNeighborhood`

A point neighborhood worklet supports the following tags in the parameters of its `ControlSignature`.

struct **CellSetIn** : public `vtkm::worklet::WorkletNeighborhood::CellSetIn`

*#include <WorkletPointNeighborhood.h>* A control signature tag for input connectivity.

This tag represents the cell set that defines the collection of points the map will operate on. A `CellSetIn` argument expects a `vtkm::cont::CellSetStructured` object in the associated parameter of the invoke.

There must be exactly one `CellSetIn` argument, and the worklet's `InputDomain` must be set to this argument.

struct **FieldIn** : public `vtkm::worklet::WorkletNeighborhood::FieldIn`

*#include <WorkletPointNeighborhood.h>* A control signature tag for input fields.

A `FieldIn` argument expects a `vtkm::cont::ArrayHandle` in the associated parameter of the invoke. Each invocation of the worklet gets a single value out of this array.

This tag means that the field is read only.

struct **FieldInNeighborhood** : public `vtkm::worklet::WorkletNeighborhood::FieldInNeighborhood`

*#include <WorkletPointNeighborhood.h>* A control signature tag for neighborhood input values.

A neighborhood worklet operates by allowing access to a adjacent element values in a  $N \times N \times N$  patch called a neighborhood. No matter the size of the neighborhood it is symmetric across its center in each axis, and the current point value will be at the center. For example a  $3 \times 3 \times 3$  neighborhood would have local indices ranging from -1 to 1 in each dimension.

This tag specifies a `vtkm::cont::ArrayHandle` object that holds the values. It is an input array with entries for each element.

What differentiates `FieldInNeighborhood` from `FieldIn` is that `FieldInNeighborhood` allows the worklet function to access the field value at the element it is visiting and the field values in the neighborhood around it. Thus, instead of getting a single value out of the array, each invocation of the worklet gets a `vtkm::exec::FieldNeighborhood` object. These objects allow retrieval of field values using indices relative to the visited element.

struct **FieldOut** : public `vtkm::worklet::WorkletNeighborhood::FieldOut`

*#include <WorkletPointNeighborhood.h>* A control signature tag for output fields.

A `FieldOut` argument expects a `vtkm::cont::ArrayHandle` in the associated parameter of the invoke. The array is resized before scheduling begins, and each invocation of the worklet sets a single value in the array.

This tag means that the field is write only.



```
struct FieldInOut : public vtkm::worklet::WorkletNeighborhood::FieldInOut
```

*#include <WorkletPointNeighborhood.h>* A control signature tag for input-output (in-place) fields.

A *FieldInOut* argument expects a *vtkm::cont::ArrayHandle* in the associated parameter of the invoke. Each invocation of the worklet gets a single value out of this array, which is replaced by the resulting value after the worklet completes.

This tag means that the field is read and write.

```
struct WholeArrayIn : public vtkm::worklet::internal::WorkletBase::WholeArrayIn
```

*#include <WorkletPointNeighborhood.h>* ControlSignature tag for whole input arrays.

The *WholeArrayIn* control signature tag specifies a *vtkm::cont::ArrayHandle* passed to the invoke of the worklet. An array portal capable of reading from any place in the array is given to the worklet.

```
struct WholeArrayOut : public vtkm::worklet::internal::WorkletBase::WholeArrayOut
```

*#include <WorkletPointNeighborhood.h>* ControlSignature tag for whole output arrays.

The *WholeArrayOut* control signature tag specifies an *vtkm::cont::ArrayHandle* passed to the invoke of the worklet. An array portal capable of writing to any place in the array is given to the worklet. Developers should take care when using writable whole arrays as introducing race conditions is possible.

```
struct WholeArrayInOut : public vtkm::worklet::internal::WorkletBase::WholeArrayInOut
```

*#include <WorkletPointNeighborhood.h>* ControlSignature tag for whole input/output arrays.

The *WholeArrayOut* control signature tag specifies a *vtkm::cont::ArrayHandle* passed to the invoke of the worklet. An array portal capable of reading from or writing to any place in the array is given to the worklet. Developers should take care when using writable whole arrays as introducing race conditions is possible.

```
struct AtomicArrayInOut : public vtkm::worklet::internal::WorkletBase::AtomicArrayInOut
```

*#include <WorkletPointNeighborhood.h>* ControlSignature tag for whole input/output arrays.

The *AtomicArrayInOut* control signature tag specifies *vtkm::cont::ArrayHandle* passed to the invoke of the worklet. A *vtkm::exec::AtomicArray* object capable of performing atomic operations to the entries in the array is given to the worklet. Atomic arrays can help avoid race conditions but can slow down the running of a parallel algorithm.

```
template<typename VisitTopology = Cell, typename IncidentTopology = Point>
```

```
struct WholeCellSetIn : public vtkm::worklet::internal::WorkletBase::WholeCellSetIn<Cell, Point>
```

*#include <WorkletPointNeighborhood.h>* ControlSignature tag for whole input topology.

The *WholeCellSetIn* control signature tag specifies a *vtkm::cont::CellSet* passed to the invoke of the worklet. A connectivity object capable of finding elements of one type that are incident on elements of a different type. This can be used to global lookup for arbitrary topology information

```
struct ExecObject : public vtkm::worklet::internal::WorkletBase::ExecObject
```

*#include <WorkletPointNeighborhood.h>* ControlSignature tag for execution object inputs.

This tag represents an execution object that is passed directly from the control environment to the worklet. A *ExecObject* argument expects a subclass of *vtkm::exec::ExecutionObjectBase*. Subclasses of *vtkm::exec::ExecutionObjectBase* behave like a factory for objects that work on particular devices. They do this by implementing a *PrepareForExecution()* method that takes a device adapter tag and returns an object that works on that device. That device-specific object is passed directly to the worklet.

A point neighborhood worklet supports the following tags in the parameters of its `ExecutionSignature`.

```
struct _1 : public vtkm::placeholders::Arg<1>
```

*#include* <WorkletPointNeighborhood.h> Argument placeholders for an `ExecutionSignature`.

All worklet superclasses declare numeric tags in the form of `_1`, `_2`, `_3` etc. that are used in the `ExecutionSignature` to refer to the corresponding parameter in the `ControlSignature`.

```
struct Boundary : public vtkm::worklet::WorkletNeighborhood::Boundary
```

*#include* <WorkletPointNeighborhood.h> The `ExecutionSignature` tag to query if the current iteration is inside the boundary.

This `ExecutionSignature` tag provides a `vtkm::exec::BoundaryState` object that provides information about where the local neighborhood is in relationship to the full mesh. It allows you to query whether the neighborhood of the current worklet call is completely inside the bounds of the mesh or if it extends beyond the mesh. This is important as when you are on a boundary the neighborhood will contain empty values for a certain subset of values, and in this case the values returned will depend on the boundary behavior.

```
struct WorkIndex : public vtkm::exec::arg::WorkIndex
```

*#include* <WorkletPointNeighborhood.h> The `ExecutionSignature` tag to use to get the work index.

This tag produces a `vtkm::Id` that uniquely identifies the invocation instance of the worklet. When a worklet is dispatched, it broken into pieces defined by the input domain and scheduled on independent threads. This tag in the `ExecutionSignature` passes the index for this work.

```
struct VisitIndex : public vtkm::exec::arg::VisitIndex
```

*#include* <WorkletPointNeighborhood.h> The `ExecutionSignature` tag to use to get the visit index.

This tag produces a `vtkm::IdComponent` that uniquely identifies when multiple worklet invocations operate on the same input item, which can happen when defining a worklet with scatter.

When a worklet is dispatched, there is a scatter operation defined that optionally allows each input to go to multiple output entries. When one input is assigned to multiple outputs, there needs to be a mechanism to uniquely identify which output is which. The visit index is a value between 0 and the number of outputs a particular input goes to. This tag in the `ExecutionSignature` passes the visit index for this work.

```
struct InputIndex : public vtkm::exec::arg::InputIndex
```

*#include* <WorkletPointNeighborhood.h> The `ExecutionSignature` tag to use to get the input index.

This tag produces a `vtkm::Id` that identifies the index of the input element, which can differ from the `WorkIndex` in a worklet with a scatter.

When a worklet is dispatched, it broken into pieces defined by the input domain and scheduled on independent threads. This tag in the `ExecutionSignature` passes the index of the input element that the work thread is currently working on. When a worklet has a scatter associated with it, the input and output indices can be different.

```
struct OutputIndex : public vtkm::exec::arg::OutputIndex
```

*#include* <WorkletPointNeighborhood.h> The `ExecutionSignature` tag to use to get the output index.

This tag produces a `vtkm::Id` that identifies the index of the output element. (This is generally the same as `WorkIndex`.)

When a worklet is dispatched, it broken into pieces defined by the output domain and scheduled on independent threads. This tag in the `ExecutionSignature` passes the index of the output element that the work thread is

currently working on. When a worklet has a scatter associated with it, the output and output indices can be different.

```
struct ThreadIndices : public vtkm::exec::arg::ThreadIndices
```

*#include <WorkletPointNeighborhood.h>* The `ExecutionSignature` tag to use to get the thread indices.

This tag produces an internal object that manages indices and other metadata of the current thread. Thread indices objects vary by worklet type, but most users can get the information they need through other signature tags.

When a worklet is dispatched, it broken into pieces defined by the input domain and scheduled on independent threads. During this process multiple indices associated with the input and output can be generated. This tag in the `ExecutionSignature` passes the index for this work.

```
struct Device : public vtkm::worklet::internal::WorkletBase::Device
```

*#include <WorkletPointNeighborhood.h>* `ExecutionSignature` tag for getting the device adapter tag.

This tag passes a device adapter tag object. This allows the worklet function to template on or overload itself based on the type of device that it is being executed on.

## 22.3.2 Cell Neighborhood

A worklet deriving `vtkm::worklet::WorkletCellNeighborhood` performs a mapping operation that applies a function (the operator in the worklet) on all the cells of a `vtkm::cont::DataSet`. While operating on the cell, the worklet has access to field values on nearby cells within a neighborhood.

```
class WorkletCellNeighborhood : public vtkm::worklet::WorkletNeighborhood
```

Base class for worklets that map over the cells in a structured grid with neighborhood information.

The domain of a `WorkletCellNeighborhood` is a `vtkm::cont::CellSetStructured`. It visits all the cells in the mesh and provides access to the cell field values of the visited cell and the field values of the nearby connected neighborhood of a prescribed size.

A cell neighborhood worklet supports the following tags in the parameters of its `ControlSignature`.

```
struct CellSetIn : public vtkm::worklet::WorkletNeighborhood::CellSetIn
```

*#include <WorkletCellNeighborhood.h>* A control signature tag for input connectivity.

This tag represents the cell set that defines the collection of points the map will operate on. A `CellSetIn` argument expects a `vtkm::cont::CellSetStructured` object in the associated parameter of the invoke.

There must be exactly one `CellSetIn` argument, and the worklet's `InputDomain` must be set to this argument.

```
struct FieldIn : public vtkm::worklet::WorkletNeighborhood::FieldIn
```

*#include <WorkletCellNeighborhood.h>* A control signature tag for input fields.

A `FieldIn` argument expects a `vtkm::cont::ArrayHandle` in the associated parameter of the invoke. Each invocation of the worklet gets a single value out of this array.

This tag means that the field is read only.

```
struct FieldInNeighborhood : public vtkm::worklet::WorkletNeighborhood::FieldInNeighborhood
```

*#include <WorkletCellNeighborhood.h>* A control signature tag for neighborhood input values.

A neighborhood worklet operates by allowing access to a adjacent element values in a NxNxN patch called a neighborhood. No matter the size of the neighborhood it is symmetric across its center in each axis, and the current point value will be at the center. For example a 3x3x3 neighborhood would have local indices ranging from -1 to 1 in each dimension.

This tag specifies a `vtkm::cont::ArrayHandle` object that holds the values. It is an input array with entries for each element.

What differentiates `FieldInNeighborhood` from `FieldIn` is that `FieldInNeighborhood` allows the worklet function to access the field value at the element it is visiting and the field values in the neighborhood around it. Thus, instead of getting a single value out of the array, each invocation of the worklet gets a `vtkm::exec::FieldNeighborhood` object. These objects allow retrieval of field values using indices relative to the visited element.

```
struct FieldOut : public vtkm::worklet::WorkletNeighborhood::FieldOut
```

```
#include <WorkletCellNeighborhood.h> A control signature tag for output fields.
```

A `FieldOut` argument expects a `vtkm::cont::ArrayHandle` in the associated parameter of the invoke. The array is resized before scheduling begins, and each invocation of the worklet sets a single value in the array.

This tag means that the field is write only.

```
struct FieldInOut : public vtkm::worklet::WorkletNeighborhood::FieldInOut
```

```
#include <WorkletCellNeighborhood.h> A control signature tag for input-output (in-place) fields.
```

A `FieldInOut` argument expects a `vtkm::cont::ArrayHandle` in the associated parameter of the invoke. Each invocation of the worklet gets a single value out of this array, which is replaced by the resulting value after the worklet completes.

This tag means that the field is read and write.

```
struct WholeArrayIn : public vtkm::worklet::internal::WorkletBase::WholeArrayIn
```

```
#include <WorkletCellNeighborhood.h> ControlSignature tag for whole input arrays.
```

The `WholeArrayIn` control signature tag specifies a `vtkm::cont::ArrayHandle` passed to the invoke of the worklet. An array portal capable of reading from any place in the array is given to the worklet.

```
struct WholeArrayOut : public vtkm::worklet::internal::WorkletBase::WholeArrayOut
```

```
#include <WorkletCellNeighborhood.h> ControlSignature tag for whole output arrays.
```

The `WholeArrayOut` control signature tag specifies an `vtkm::cont::ArrayHandle` passed to the invoke of the worklet. An array portal capable of writing to any place in the array is given to the worklet. Developers should take care when using writable whole arrays as introducing race conditions is possible.

```
struct WholeArrayInOut : public vtkm::worklet::internal::WorkletBase::WholeArrayInOut
```

```
#include <WorkletCellNeighborhood.h> ControlSignature tag for whole input/output arrays.
```

The `WholeArrayOut` control signature tag specifies a `vtkm::cont::ArrayHandle` passed to the invoke of the worklet. An array portal capable of reading from or writing to any place in the array is given to the worklet. Developers should take care when using writable whole arrays as introducing race conditions is possible.

```
struct AtomicArrayInOut : public vtkm::worklet::internal::WorkletBase::AtomicArrayInOut
```

```
#include <WorkletCellNeighborhood.h> ControlSignature tag for whole input/output arrays.
```

The `AtomicArrayInOut` control signature tag specifies `vtkm::cont::ArrayHandle` passed to the invoke of the worklet. A `vtkm::exec::AtomicArray` object capable of performing atomic operations to the entries in

the array is given to the worklet. Atomic arrays can help avoid race conditions but can slow down the running of a parallel algorithm.

```
template<typename VisitTopology = Cell, typename IncidentTopology = Point>
```

```
struct WholeCellSetIn : public vtkm::worklet::internal::WorkletBase::WholeCellSetIn<Cell, Point>
```

*#include <WorkletCellNeighborhood.h>* ControlSignature tag for whole input topology.

The *WholeCellSetIn* control signature tag specifies a *vtkm::cont::CellSet* passed to the invoke of the worklet. A connectivity object capable of finding elements of one type that are incident on elements of a different type. This can be used to global lookup for arbitrary topology information

```
struct ExecObject : public vtkm::worklet::internal::WorkletBase::ExecObject
```

*#include <WorkletCellNeighborhood.h>* ControlSignature tag for execution object inputs.

This tag represents an execution object that is passed directly from the control environment to the worklet. A *ExecObject* argument expects a subclass of *vtkm::exec::ExecutionObjectBase*. Subclasses of *vtkm::exec::ExecutionObjectBase* behave like a factory for objects that work on particular devices. They do this by implementing a *PrepareForExecution()* method that takes a device adapter tag and returns an object that works on that device. That device-specific object is passed directly to the worklet.

A cell neighborhood worklet supports the following tags in the parameters of its *ExecutionSignature*.

```
struct _1 : public vtkm::placeholders::Arg<1>
```

*#include <WorkletCellNeighborhood.h>* Argument placeholders for an *ExecutionSignature*.

All worklet superclasses declare numeric tags in the form of *\_1*, *\_2*, *\_3* etc. that are used in the *ExecutionSignature* to refer to the corresponding parameter in the *ControlSignature*.

```
struct Boundary : public vtkm::worklet::WorkletNeighborhood::Boundary
```

*#include <WorkletCellNeighborhood.h>* The *ExecutionSignature* tag to query if the current iteration is inside the boundary.

This *ExecutionSignature* tag provides a *vtkm::exec::BoundaryState* object that provides information about where the local neighborhood is in relationship to the full mesh. It allows you to query whether the neighborhood of the current worklet call is completely inside the bounds of the mesh or if it extends beyond the mesh. This is important as when you are on a boundary the neighborhood will contain empty values for a certain subset of values, and in this case the values returned will depend on the boundary behavior.

```
struct WorkIndex : public vtkm::exec::arg::WorkIndex
```

*#include <WorkletCellNeighborhood.h>* The *ExecutionSignature* tag to use to get the work index.

This tag produces a *vtkm::Id* that uniquely identifies the invocation instance of the worklet. When a worklet is dispatched, it broken into pieces defined by the input domain and scheduled on independent threads. This tag in the *ExecutionSignature* passes the index for this work.

```
struct VisitIndex : public vtkm::exec::arg::VisitIndex
```

*#include <WorkletCellNeighborhood.h>* The *ExecutionSignature* tag to use to get the visit index.

This tag produces a *vtkm::IdComponent* that uniquely identifies when multiple worklet invocations operate on the same input item, which can happen when defining a worklet with scatter.

When a worklet is dispatched, there is a scatter operation defined that optionally allows each input to go to multiple output entries. When one input is assigned to multiple outputs, there needs to be a mechanism to uniquely identify which output is which. The visit index is a value between 0 and the number of outputs a particular input goes to. This tag in the *ExecutionSignature* passes the visit index for this work.

struct **InputIndex** : public vtkm::exec::arg::InputIndex

*#include <WorkletCellNeighborhood.h>* The `ExecutionSignature` tag to use to get the input index.

This tag produces a `vtkm::Id` that identifies the index of the input element, which can differ from the `WorkIndex` in a worklet with a scatter.

When a worklet is dispatched, it broken into pieces defined by the input domain and scheduled on independent threads. This tag in the `ExecutionSignature` passes the index of the input element that the work thread is currently working on. When a worklet has a scatter associated with it, the input and output indices can be different.

struct **OutputIndex** : public vtkm::exec::arg::OutputIndex

*#include <WorkletCellNeighborhood.h>* The `ExecutionSignature` tag to use to get the output index.

This tag produces a `vtkm::Id` that identifies the index of the output element. (This is generally the same as `WorkIndex`.)

When a worklet is dispatched, it broken into pieces defined by the output domain and scheduled on independent threads. This tag in the `ExecutionSignature` passes the index of the output element that the work thread is currently working on. When a worklet has a scatter associated with it, the output and output indices can be different.

struct **ThreadIndices** : public vtkm::exec::arg::ThreadIndices

*#include <WorkletCellNeighborhood.h>* The `ExecutionSignature` tag to use to get the thread indices.

This tag produces an internal object that manages indices and other metadata of the current thread. Thread indices objects vary by worklet type, but most users can get the information they need through other signature tags.

When a worklet is dispatched, it broken into pieces defined by the input domain and scheduled on independent threads. During this process multiple indices associated with the input and output can be generated. This tag in the `ExecutionSignature` passes the index for this work.

struct **Device** : public vtkm::worklet::internal::WorkletBase::Device

*#include <WorkletCellNeighborhood.h>* `ExecutionSignature` tag for getting the device adapter tag.

This tag passes a device adapter tag object. This allows the worklet function to template on or overload itself based on the type of device that it is being executed on.

### 22.3.3 Neighborhood Information

As stated earlier in this section, what makes a `vtkm::worklet::WorkletPointNeighborhood` worklet special is its ability to get field information in a neighborhood surrounding a point rather than just the point itself. This is done using the special `FieldInNeighborhood` in the `ControlSignature`. When you use this tag, rather than getting the single field value for the point, you get a `vtkm::exec::FieldNeighborhood` object.

The `vtkm::exec::FieldNeighborhood` class contains a `vtkm::exec::FieldNeighborhood::Get()` method that retrieves a field value relative to the local neighborhood. `vtkm::exec::FieldNeighborhood::Get()` takes the  $i, j, k$  index of the point with respect to the local point. So, calling `Get(0,0,0)` retrieves at the point being visited. Likewise, `Get(-1,0,0)` gets the value to the “left” of the point visited and `Get(1,0,0)` gets the value to the “right.”

template<typename **FieldPortalType**>

struct **FieldNeighborhood**



Retrieves field values from a neighborhood.

*FieldNeighborhood* manages the retrieval of field values within the neighborhood of a *vtkm::worklet::WorkletPointNeighborhood* worklet. The *Get* methods take *ijk* indices relative to the neighborhood (with 0, 0, 0 being the element visited) and return the field value at that part of the neighborhood. If the requested neighborhood is outside the boundary, the value at the nearest boundary will be returned. A *vtkm::exec::BoundaryState* object can be used to determine if the neighborhood extends beyond the boundary of the mesh.

This class is typically constructed using the *FieldInNeighborhood* tag in an *ExecutionSignature*. There is little reason to construct this in user code.

## Public Functions

```
inline ValueType Get(vtkm::IdComponent i, vtkm::IdComponent j, vtkm::IdComponent k) const
```

Retrieve a field value relative to the visited element.

The index is given as three dimensional *i*, *j*, *k* indices. These indices are relative to the currently visited element. So, calling *Get*(0, 0, 0) retrieves the field value at the visited element. Calling *Get*(-1, 0, 0) retrieves the value to the “left” and *Get*(1, 0, 0) retrieves the value to the “right.”

If the relative index points outside the bounds of the mesh, *Get* will return the value closest to the boundary (i.e. clamping behavior). For example, if the visited element is at the leftmost index of the mesh, *Get*(-1, 0, 0) will refer to a value outside the bounds of the mesh. In this case, *Get* will return the value at the visited index, which is the closest element at that boundary.

When referring to values in a mesh of less than 3 dimensions (such as a 2D structured), simply use 0 for the unused dimensions.

```
inline ValueType GetUnchecked(vtkm::IdComponent i, vtkm::IdComponent j, vtkm::IdComponent k) const
```

Retrieve a field value relative to the visited element without bounds checking.

*GetUnchecked* behaves the same as *Get* except that no bounds checking is done before retrieving the field value. If the relative index is out of bounds of the mesh, the results are undefined.

*GetUnchecked* is useful in circumstances where the bounds have already be checked. This prevents wasting time repeating checks.

```
inline ValueType Get(const vtkm::Id3 &ijk) const
```

Retrieve a field value relative to the visited element.

The index is given as three dimensional *i*, *j*, *k* indices. These indices are relative to the currently visited element. So, calling *Get*(0, 0, 0) retrieves the field value at the visited element. Calling *Get*(-1, 0, 0) retrieves the value to the “left” and *Get*(1, 0, 0) retrieves the value to the “right.”

If the relative index points outside the bounds of the mesh, *Get* will return the value closest to the boundary (i.e. clamping behavior). For example, if the visited element is at the leftmost index of the mesh, *Get*(-1, 0, 0) will refer to a value outside the bounds of the mesh. In this case, *Get* will return the value at the visited index, which is the closest element at that boundary.

When referring to values in a mesh of less than 3 dimensions (such as a 2D structured), simply use 0 for the unused dimensions.

```
inline ValueType GetUnchecked(const vtkm::Id3 &ijk) const
```

Retrieve a field value relative to the visited element without bounds checking.

*GetUnchecked* behaves the same as *Get* except that no bounds checking is done before retrieving the field value. If the relative index is out of bounds of the mesh, the results are undefined.

GetUnchecked is useful in circumstances where the bounds have already be checked. This prevents wasting time repeating checks.

## Public Members

`vtkm::exec::BoundaryState` const \*const **Boundary**

The `vtkm::exec::BoundaryState` used to find field values from local indices.

`FieldPortalType` **Portal**

The array portal containing field values.

Example 5: Retrieve neighborhood field value.

```
sum = sum + inputField.Get(i, j, k);
```

When performing operations on a neighborhood within the mesh, it is often important to know whether the expected neighborhood is contained completely within the mesh or whether the neighborhood extends beyond the borders of the mesh. This can be queried using a `vtkm::exec::BoundaryState` object, which is provided when a **Boundary** tag is listed in the **ExecutionSignature**.

Generally, `vtkm::exec::BoundaryState` allows you to specify the size of the neighborhood at runtime. The neighborhood size is specified by a radius. The radius specifies the number of items in each direction the neighborhood extends. So, for example, a point neighborhood with radius 1 would contain a  $3 \times 3 \times 3$  neighborhood centered around the point. Likewise, a point neighborhood with radius 2 would contain a  $5 \times 5 \times 5$  neighborhood centered around the point. `vtkm::exec::BoundaryState` provides several methods to determine if the neighborhood is contained in the mesh.

struct **BoundaryState**

Provides a neighborhood's placement with respect to the mesh's boundary.

`BoundaryState` provides functionality for `vtkm::worklet::WorkletPointNeighborhood` algorithms to determine if they are operating on a point near the boundary. It allows you to query about overlaps of the neighborhood and the mesh boundary. It also helps convert local neighborhood ids to the corresponding location in the mesh.

This class is typically constructed using the **Boundary** tag in an **ExecutionSignature**. There is little reason to construct this in user code.

## Unnamed Group

inline bool **IsRadiusInXBoundary**(`vtkm::IdComponent` radius) const

Returns true if a neighborhood of the given radius is contained within the bounds of the cell set in the X, Y, or Z direction. Returns false if the neighborhood extends outside of the boundary of the data in the X, Y, or Z direction.

The radius defines the size of the neighborhood in terms of how far away it extends from the center. So if there is a radius of 1, the neighborhood extends 1 unit away from the center in each direction and is  $3 \times 3 \times 3$ . If there is a radius of 2, the neighborhood extends 2 units for a size of  $5 \times 5 \times 5$ .

inline bool **IsRadiusInYBoundary**(`vtkm::IdComponent` radius) const

Returns true if a neighborhood of the given radius is contained within the bounds of the cell set in the X, Y, or Z direction. Returns false if the neighborhood extends outside of the boundary of the data in the X, Y, or Z direction.



The radius defines the size of the neighborhood in terms of how far away it extends from the center. So if there is a radius of 1, the neighborhood extends 1 unit away from the center in each direction and is 3x3x3. If there is a radius of 2, the neighborhood extends 2 units for a size of 5x5x5.

inline bool **IsRadiusInZBoundary**(vtkm::IdComponent radius) const

Returns true if a neighborhood of the given radius is contained within the bounds of the cell set in the X, Y, or Z direction. Returns false if the neighborhood extends outside of the boundary of the data in the X, Y, or Z direction.

The radius defines the size of the neighborhood in terms of how far away it extends from the center. So if there is a radius of 1, the neighborhood extends 1 unit away from the center in each direction and is 3x3x3. If there is a radius of 2, the neighborhood extends 2 units for a size of 5x5x5.

## Unnamed Group

inline bool **IsNeighborInXBoundary**(vtkm::IdComponent offset) const

Returns true if the neighbor at the specified *offset* is contained within the bounds of the cell set in the X, Y, or Z direction. Returns false if the neighbor falls outside of the boundary of the data in the X, Y, or Z direction.

inline bool **IsNeighborInYBoundary**(vtkm::IdComponent offset) const

Returns true if the neighbor at the specified *offset* is contained within the bounds of the cell set in the X, Y, or Z direction. Returns false if the neighbor falls outside of the boundary of the data in the X, Y, or Z direction.

inline bool **IsNeighborInZBoundary**(vtkm::IdComponent offset) const

Returns true if the neighbor at the specified *offset* is contained within the bounds of the cell set in the X, Y, or Z direction. Returns false if the neighbor falls outside of the boundary of the data in the X, Y, or Z direction.

## Public Functions

inline const vtkm::Id3 &**GetCenterIndex**() const

Returns the center index of the neighborhood.

This is typically the position of the invocation of the worklet given this boundary condition.

inline bool **IsRadiusInBoundary**(vtkm::IdComponent radius) const

Returns true if a neighborhood of the given radius is contained within the bounds of the cell set.

Returns false if the neighborhood extends outside of the boundary of the data.

The radius defines the size of the neighborhood in terms of how far away it extends from the center. So if there is a radius of 1, the neighborhood extends 1 unit away from the center in each direction and is 3x3x3. If there is a radius of 2, the neighborhood extends 2 units for a size of 5x5x5.

inline bool **IsNeighborInBoundary**(const vtkm::IdComponent3 &neighbor) const

Returns true if the neighbor at the specified offset vector is contained within the bounds of the cell set.

Returns false if the neighbor falls outside of the boundary of the data.

inline vtkm::IdComponent3 **MinNeighborIndices**(vtkm::IdComponent radius) const

Returns the minimum neighborhood indices that are within the bounds of the data.

Given a radius for the neighborhood, returns a *vtkm::IdComponent3* for the “lower left” (minimum) index. If the visited point is in the middle of the mesh, the returned triplet is the negative radius for all components. But if the visited point is near the mesh boundary, then the minimum index will be clipped.

For example, if the visited point is at [5,5,5] and `MinNeighborIndices(2)` is called, then [-2,-2,-2] is returned. However, if the visited point is at [0,1,2] and `MinNeighborIndices(2)` is called, then [0,-1,-2] is returned.

`inline vtkm::IdComponent3 MaxNeighborIndices(vtkm::IdComponent radius) const`

Returns the minimum neighborhood indices that are within the bounds of the data.

Given a radius for the neighborhood, returns a `vtkm::IdComponent3` for the “upper right” (maximum) index. If the visited point is in the middle of the mesh, the returned triplet is the positive radius for all components. But if the visited point is near the mesh boundary, then the maximum index will be clipped.

For example, if the visited point is at [5,5,5] in a 10 by 10 by 10 mesh and `MaxNeighborIndices(2)` is called, then [2,2,2] is returned. However, if the visited point is at [7, 8, 9] in the same mesh and `MaxNeighborIndices(2)` is called, then [2, 1, 0] is returned.

`inline vtkm::Id3 NeighborIndexToFullIndexClamp(const vtkm::IdComponent3 &neighbor) const`

Takes a local neighborhood index (in the ranges of -neighborhood size to neighborhood size) and returns the ijk of the equivalent point in the full data set.

If the given value is out of range, the value is clamped to the nearest boundary. For example, if given a neighbor index that is past the minimum x range of the data, the index at the minimum x boundary is returned.

`inline vtkm::Id3 NeighborIndexToFullIndexClamp(vtkm::IdComponent neighborI, vtkm::IdComponent neighborJ, vtkm::IdComponent neighborK) const`

Takes a local neighborhood index (in the ranges of -neighborhood size to neighborhood size) and returns the ijk of the equivalent point in the full data set.

If the given value is out of range, the value is clamped to the nearest boundary. For example, if given a neighbor index that is past the minimum x range of the data, the index at the minimum x boundary is returned.

`inline vtkm::Id3 NeighborIndexToFullIndex(const vtkm::IdComponent3 &neighbor) const`

Takes a local neighborhood index (in the ranges of -neighborhood size to neighborhood size) and returns the ijk of the equivalent point in the full data set.

If the given value is out of range, the returned value is undefined.

`inline vtkm::Id3 NeighborIndexToFullIndex(vtkm::IdComponent neighborI, vtkm::IdComponent neighborJ, vtkm::IdComponent neighborK) const`

Takes a local neighborhood index (in the ranges of -neighborhood size to neighborhood size) and returns the ijk of the equivalent point in the full data set.

If the given value is out of range, the returned value is undefined.

`inline vtkm::IdComponent3 ClampNeighborIndex(const vtkm::IdComponent3 &neighbor) const`

Takes a local neighborhood index (in the ranges of -neighborhood size to neighborhood size), clamps it to the dataset bounds, and returns a new neighborhood index.

For example, if given a neighbor index that is past the minimum x range of the data, the neighbor index of the minimum x boundary is returned.

`inline vtkm::IdComponent3 ClampNeighborIndex(vtkm::IdComponent neighborI, vtkm::IdComponent neighborJ, vtkm::IdComponent neighborK) const`

Takes a local neighborhood index (in the ranges of -neighborhood size to neighborhood size), clamps it to the dataset bounds, and returns a new neighborhood index.

For example, if given a neighbor index that is past the minimum x range of the data, the neighbor index of the minimum x boundary is returned.

inline vtkm::Id NeighborIndexToFlatIndexClamp(const vtkm::IdComponent3 &neighbor) const

Takes a local neighborhood index (in the ranges of -neighborhood size to neighborhood size) and returns the flat index of the equivalent point in the full data set.

If the given value is out of range, the value is clamped to the nearest boundary. For example, if given a neighbor index that is past the minimum x range of the data, the index at the minimum x boundary is returned.

inline vtkm::Id NeighborIndexToFlatIndexClamp(vtkm::IdComponent neighborI, vtkm::IdComponent neighborJ, vtkm::IdComponent neighborK) const

Takes a local neighborhood index (in the ranges of -neighborhood size to neighborhood size) and returns the flat index of the equivalent point in the full data set.

If the given value is out of range, the value is clamped to the nearest boundary. For example, if given a neighbor index that is past the minimum x range of the data, the index at the minimum x boundary is returned.

inline vtkm::Id NeighborIndexToFlatIndex(const vtkm::IdComponent3 &neighbor) const

Takes a local neighborhood index (in the ranges of -neighborhood size to neighborhood size) and returns the flat index of the equivalent point in the full data set.

If the given value is out of range, the result is undefined.

inline vtkm::Id NeighborIndexToFlatIndex(vtkm::IdComponent neighborI, vtkm::IdComponent neighborJ, vtkm::IdComponent neighborK) const

Takes a local neighborhood index (in the ranges of -neighborhood size to neighborhood size) and returns the flat index of the equivalent point in the full data set.

If the given value is out of range, the result is undefined.

## Public Members

vtkm::Id3 IJK

The 3D index of the visited element.

vtkm::Id3 PointDimensions

The dimensions of the elements in the mesh.

The `vtkm::exec::BoundaryState::MinNeighborIndices()` and `vtkm::exec::BoundaryState::MaxNeighborIndices()` are particularly useful for iterating over the valid portion of the neighborhood.

Example 6: Iterating over the valid portion of a neighborhood.

```

1  auto minIndices = boundary.MinNeighborIndices(this->NumberOfLayers);
2  auto maxIndices = boundary.MaxNeighborIndices(this->NumberOfLayers);
3
4  T sum = 0;
5  vtkm::IdComponent size = 0;
6  for (vtkm::IdComponent k = minIndices[2]; k <= maxIndices[2]; ++k)
7  {
8      for (vtkm::IdComponent j = minIndices[1]; j <= maxIndices[1]; ++j)
9      {
10         for (vtkm::IdComponent i = minIndices[0]; i <= maxIndices[0]; ++i)
11         {

```

(continues on next page)

(continued from previous page)

```

12         sum = sum + inputField.Get(i, j, k);
13         ++size;
14     }
15 }
16 }

```

### 22.3.4 Convolver Small Kernels

A common use case for point neighborhood worklets is to convolve a small kernel with a structured mesh. A very simple example of this is averaging out the values the values within some distance to the central point. This has the effect of smoothing out the field (although smoothing filters with better properties exist). The following example shows a worklet that applies this simple “box” averaging.

Example 7: Implementation and use of a point neighborhood worklet.

```

1  class ApplyBoxKernel : public vtkm::worklet::WorkletPointNeighborhood
2  {
3  private:
4      vtkm::IdComponent NumberOfLayers;
5
6  public:
7      using ControlSignature = void(CellSetIn cellSet,
8                                   FieldInNeighborhood inputField,
9                                   FieldOut outputField);
10     using ExecutionSignature = _3(_2, Boundary);
11
12     using InputDomain = _1;
13
14     ApplyBoxKernel(vtkm::IdComponent kernelSize)
15     {
16         VTKM_ASSERT(kernelSize >= 3);
17         VTKM_ASSERT((kernelSize % 2) == 1);
18
19         this->NumberOfLayers = (kernelSize - 1) / 2;
20     }
21
22     template<typename InputFieldPortalType>
23     VTKM_EXEC typename InputFieldPortalType::ValueType operator()(
24         const vtkm::exec::FieldNeighborhood<InputFieldPortalType>& inputField,
25         const vtkm::exec::BoundaryState& boundary) const
26     {
27         using T = typename InputFieldPortalType::ValueType;
28
29         auto minIndices = boundary.MinNeighborIndices(this->NumberOfLayers);
30         auto maxIndices = boundary.MaxNeighborIndices(this->NumberOfLayers);
31
32         T sum = 0;
33         vtkm::IdComponent size = 0;
34         for (vtkm::IdComponent k = minIndices[2]; k <= maxIndices[2]; ++k)
35         {
36             for (vtkm::IdComponent j = minIndices[1]; j <= maxIndices[1]; ++j)

```

(continues on next page)

(continued from previous page)

```

37 {
38   for (vtkm::IdComponent i = minIndices[0]; i <= maxIndices[0]; ++i)
39   {
40     sum = sum + inputField.Get(i, j, k);
41     ++size;
42   }
43 }
44 }
45
46 return static_cast<T>(sum / size);
47 }
48 };

```

## 22.4 Reduce by Key

A worklet deriving `vtkm::worklet::WorkletReduceByKey` operates on an array of keys and one or more associated arrays of values. When a reduce by key worklet is invoked, all identical keys are collected and the worklet is called once for each unique key. Each worklet invocation is given a Vec-like containing all values associated with the unique key. Reduce by key worklets are very useful for combining like items such as shared topology elements or coincident points.

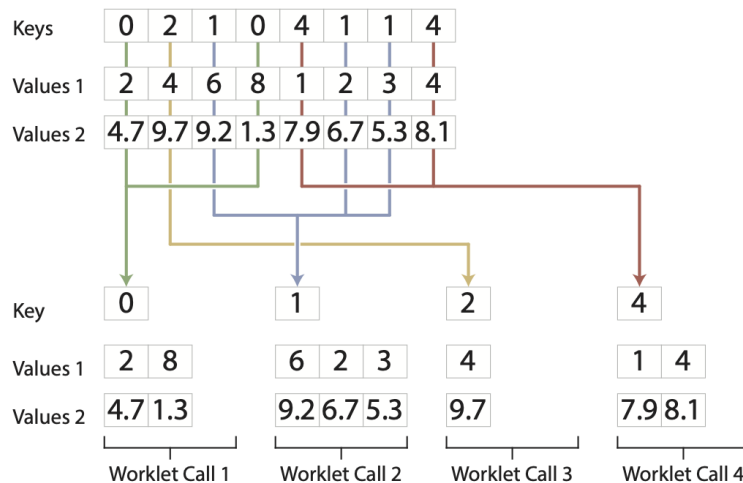


Figure 1: The collection of values for a reduce by key worklet.

Figure 1 shows a pictorial representation of how VTK-m collects data for a reduce by key worklet. All calls to a reduce by key worklet has exactly one array of keys. The key array in this example has 4 unique keys: 0, 1, 2, 4. These 4 unique keys will result in 4 calls to the worklet function. This example also has 2 arrays of values associated with the keys. (A reduce by keys worklet can have any number of values arrays.)

When the worklet is invoked, all these common keys will be collected with their associated values. The parenthesis operator of the worklet will be called once per each unique key. The worklet call will be given a Vec-like containing all values that have the key.

## 22.4.1 WorkletReduceByKey Reference

class **WorkletReduceByKey** : public vtkm::worklet::internal::WorkletBase

Base class for worklets that group elements by keys.

The InputDomain of this worklet is a `vtkm::worklet::Keys` object, which holds an array of keys. All entries of this array with the same key are collected together, and the operator of the worklet is called once for each unique key.

Input arrays are (typically) the same size as the number of keys. When these objects are passed to the operator of the worklet, all values of the associated key are placed in a Vec-like object. Output arrays get sized by the number of unique keys, and each call to the operator produces one result for each output.

Subclassed by `vtkm::worklet::AverageByKey::AverageWorklet`

A reduce by key worklet supports the following tags in the parameters of its `ControlSignature`.

struct **KeysIn** : public vtkm::cont::arg::ControlSignatureTagBase

*#include* <WorkletReduceByKey.h> A control signature tag for input keys.

A `WorkletReduceByKey` operates by collecting all identical keys and then executing the worklet on each unique key. This tag specifies a `vtkm::worklet::Keys` object that defines and manages these keys.

A `WorkletReduceByKey` should have exactly one `KeysIn` tag in its `ControlSignature`, and the `InputDomain` should point to it.

struct **ValuesIn** : public vtkm::cont::arg::ControlSignatureTagBase

*#include* <WorkletReduceByKey.h> A control signature tag for input values associated with the keys.

A `WorkletReduceByKey` operates by collecting all values associated with identical keys and then giving the worklet a Vec-like object containing all values with a matching key. This tag specifies an `vtkm::cont::ArrayHandle` object that holds the values. The number of values in this array must be equal to the size of the array used with the `KeysIn` argument.

struct **ValuesInOut** : public vtkm::cont::arg::ControlSignatureTagBase

*#include* <WorkletReduceByKey.h> A control signature tag for input/output values associated with the keys.

A `WorkletReduceByKey` operates by collecting all values associated with identical keys and then giving the worklet a Vec-like object containing all values with a matching key. This tag specifies an `vtkm::cont::ArrayHandle` object that holds the values. The number of values in this array must be equal to the size of the array used with the `KeysIn` argument.

This tag might not work with scatter operations.

struct **ValuesOut** : public vtkm::cont::arg::ControlSignatureTagBase

*#include* <WorkletReduceByKey.h> A control signature tag for output values associated with the keys.

This tag behaves the same as `ValuesInOut` except that the array is resized appropriately and no input values are passed to the worklet. As with `ValuesInOut`, values the worklet writes to its [Veclike] object get placed in the location of the original arrays.

Use of `ValuesOut` is rare.

This tag might not work with scatter operations.

```
struct ReducedValuesOut : public vtkm::cont::arg::ControlSignatureTagBase
```

*#include* <WorkletReduceByKey.h> A control signature tag for reduced output values.

A *WorkletReduceByKey* operates by collecting all identical keys and calling one instance of the worklet for those identical keys. The worklet then produces a “reduced” value per key. This tag specifies a *vtkm::cont::ArrayHandle* object that holds the values. The array is resized to be the number of unique keys, and each call of the operator sets a single value in the array

```
struct ReducedValuesIn : public vtkm::cont::arg::ControlSignatureTagBase
```

*#include* <WorkletReduceByKey.h> A control signature tag for reduced input values.

A *WorkletReduceByKey* operates by collecting all identical keys and calling one instance of the worklet for those identical keys. The worklet then produces a “reduced” value per key.

This tag specifies a *vtkm::cont::ArrayHandle* object that holds the values. It is an input array with entries for each reduced value. The number of values in the array must equal the number of *unique* keys.

A *ReducedValuesIn* argument is usually used to pass reduced values from one invoke of a reduce by key worklet to another invoke of a reduced by key worklet such as in an algorithm that requires iterative steps.

```
struct ReducedValuesInOut : public vtkm::cont::arg::ControlSignatureTagBase
```

*#include* <WorkletReduceByKey.h> A control signature tag for reduced output values.

A *WorkletReduceByKey* operates by collecting all identical keys and calling one instance of the worklet for those identical keys. The worklet then produces a “reduced” value per key.

This tag specifies a *vtkm::cont::ArrayHandle* object that holds the values. It is an input/output array with entries for each reduced value. The number of values in the array must equal the number of *unique* keys.

This tag behaves the same as *ReducedValuesIn* except that the worklet may write values back into the array. Make sure that the associated parameter to the worklet operator is a reference so that the changed value gets written back to the array.

```
struct WholeArrayIn : public vtkm::worklet::internal::WorkletBase::WholeArrayIn
```

*#include* <WorkletReduceByKey.h> *ControlSignature* tag for whole input arrays.

The *WholeArrayIn* control signature tag specifies a *vtkm::cont::ArrayHandle* passed to the invoke of the worklet. An array portal capable of reading from any place in the array is given to the worklet.

```
struct WholeArrayOut : public vtkm::worklet::internal::WorkletBase::WholeArrayOut
```

*#include* <WorkletReduceByKey.h> *ControlSignature* tag for whole output arrays.

The *WholeArrayOut* control signature tag specifies an *vtkm::cont::ArrayHandle* passed to the invoke of the worklet. An array portal capable of writing to any place in the array is given to the worklet. Developers should take care when using writable whole arrays as introducing race conditions is possible.

```
struct WholeArrayInOut : public vtkm::worklet::internal::WorkletBase::WholeArrayInOut
```

*#include* <WorkletReduceByKey.h> *ControlSignature* tag for whole input/output arrays.

The *WholeArrayOut* control signature tag specifies a *vtkm::cont::ArrayHandle* passed to the invoke of the worklet. An array portal capable of reading from or writing to any place in the array is given to the worklet. Developers should take care when using writable whole arrays as introducing race conditions is possible.

```
struct AtomicArrayInOut : public vtkm::worklet::internal::WorkletBase::AtomicArrayInOut
```



*#include <WorkletReduceByKey.h>* ControlSignature tag for whole input/output arrays.

The *AtomicArrayInOut* control signature tag specifies *vtkm::cont::ArrayHandle* passed to the invoke of the worklet. A *vtkm::exec::AtomicArray* object capable of performing atomic operations to the entries in the array is given to the worklet. Atomic arrays can help avoid race conditions but can slow down the running of a parallel algorithm.

```
template<typename VisitTopology = Cell, typename IncidentTopology = Point>
```

```
struct WholeCellSetIn : public vtkm::worklet::internal::WorkletBase::WholeCellSetIn<Cell, Point>
```

*#include <WorkletReduceByKey.h>* ControlSignature tag for whole input topology.

The *WholeCellSetIn* control signature tag specifies a *vtkm::cont::CellSet* passed to the invoke of the worklet. A connectivity object capable of finding elements of one type that are incident on elements of a different type. This can be used to global lookup for arbitrary topology information

```
struct ExecObject : public vtkm::worklet::internal::WorkletBase::ExecObject
```

*#include <WorkletReduceByKey.h>* ControlSignature tag for execution object inputs.

This tag represents an execution object that is passed directly from the control environment to the worklet. A *ExecObject* argument expects a subclass of *vtkm::exec::ExecutionObjectBase*. Subclasses of *vtkm::exec::ExecutionObjectBase* behave like a factory for objects that work on particular devices. They do this by implementing a *PrepareForExecution()* method that takes a device adapter tag and returns an object that works on that device. That device-specific object is passed directly to the worklet.

A reduce by key worklet supports the following tags in the parameters of its *ExecutionSignature*.

```
struct _1 : public vtkm::placeholders::Arg<1>
```

*#include <WorkletReduceByKey.h>* Argument placeholders for an *ExecutionSignature*.

All worklet superclasses declare numeric tags in the form of *\_1*, *\_2*, *\_3* etc. that are used in the *ExecutionSignature* to refer to the corresponding parameter in the *ControlSignature*.

```
struct ValueCount : public vtkm::exec::arg::ValueCount
```

*#include <WorkletReduceByKey.h>* The *ExecutionSignature* tag to get the number of values.

A *WorkletReduceByKey* operates by collecting all values associated with identical keys and then giving the worklet a Vec-like object containing all values with a matching key. This tag produces a *vtkm::IdComponent* that is equal to the number of times the key associated with this call to the worklet occurs in the input. This is the same size as the Vec-like objects provided by *ValuesIn* arguments.

```
struct WorkIndex : public vtkm::exec::arg::WorkIndex
```

*#include <WorkletReduceByKey.h>* The *ExecutionSignature* tag to use to get the work index.

This tag produces a *vtkm::Id* that uniquely identifies the invocation instance of the worklet. When a worklet is dispatched, it broken into pieces defined by the input domain and scheduled on independent threads. This tag in the *ExecutionSignature* passes the index for this work.

```
struct VisitIndex : public vtkm::exec::arg::VisitIndex
```

*#include <WorkletReduceByKey.h>* The *ExecutionSignature* tag to use to get the visit index.

This tag produces a *vtkm::IdComponent* that uniquely identifies when multiple worklet invocations operate on the same input item, which can happen when defining a worklet with scatter.

When a worklet is dispatched, there is a scatter operation defined that optionally allows each input to go to multiple output entries. When one input is assigned to multiple outputs, there needs to be a mechanism to



uniquely identify which output is which. The visit index is a value between 0 and the number of outputs a particular input goes to. This tag in the `ExecutionSignature` passes the visit index for this work.

```
struct InputIndex : public vtkm::exec::arg::InputIndex
```

*#include <WorkletReduceByKey.h>* The `ExecutionSignature` tag to use to get the input index.

This tag produces a `vtkm::Id` that identifies the index of the input element, which can differ from the `WorkIndex` in a worklet with a scatter.

When a worklet is dispatched, it broken into pieces defined by the input domain and scheduled on independent threads. This tag in the `ExecutionSignature` passes the index of the input element that the work thread is currently working on. When a worklet has a scatter associated with it, the input and output indices can be different.

```
struct OutputIndex : public vtkm::exec::arg::OutputIndex
```

*#include <WorkletReduceByKey.h>* The `ExecutionSignature` tag to use to get the output index.

This tag produces a `vtkm::Id` that identifies the index of the output element. (This is generally the same as `WorkIndex`.)

When a worklet is dispatched, it broken into pieces defined by the output domain and scheduled on independent threads. This tag in the `ExecutionSignature` passes the index of the output element that the work thread is currently working on. When a worklet has a scatter associated with it, the output and output indices can be different.

```
struct ThreadIndices : public vtkm::exec::arg::ThreadIndices
```

*#include <WorkletReduceByKey.h>* The `ExecutionSignature` tag to use to get the thread indices.

This tag produces an internal object that manages indices and other metadata of the current thread. Thread indices objects vary by worklet type, but most users can get the information they need through other signature tags.

When a worklet is dispatched, it broken into pieces defined by the input domain and scheduled on independent threads. During this process multiple indices associated with the input and output can be generated. This tag in the `ExecutionSignature` passes the index for this work.

```
struct Device : public vtkm::worklet::internal::WorkletBase::Device
```

*#include <WorkletReduceByKey.h>* `ExecutionSignature` tag for getting the device adapter tag.

This tag passes a device adapter tag object. This allows the worklet function to template on or overload itself based on the type of device that it is being executed on.

## 22.4.2 Key Objects

As specified in its documentation, the `InputDomain` of a `WorkletReducedByKey` has to be a `KeysIn` argument. Unlike simple mapping worklets, the control environment object passed as the `KeysIn` cannot be a simple `vtkm::cont::ArrayHandle`. Rather, this argument has to be given a `vtkm::worklet::Keys` object. This object manages an array of keys by reorganizing (i.e. sorting) the keys and finding duplicated keys that should be merged. A `vtkm::worklet::Keys` object can be constructed by simply providing a `vtkm::cont::ArrayHandle` to use as the keys.

```
template<typename T>
```

```
class Keys : public vtkm::worklet::internal::KeysBase
```

Manage keys for a `vtkm::worklet::WorkletReduceByKey`.

The `vtkm::worklet::WorkletReduceByKey` worklet takes an array of keys for its input domain, finds all identical keys, and runs a worklet that produces a single value for every key given all matching values. This class is used as the associated input for the keys input domain.

`Keys` is templated on the key array handle type and accepts an instance of this array handle as its constructor. It builds the internal structures needed to use the keys.

The same `Keys` structure can be used for multiple different invokes of different or the same worklets. When used in this way, the processing done in the `Keys` structure is reused for all the invokes. This is more efficient than creating a different `Keys` structure for each invoke.

## Public Functions

```
template<typename KeyStorage>
```

```
inline Keys(const vtkm::cont::ArrayHandle<KeyType, KeyStorage> &keys, vtkm::cont::DeviceAdapterId  
            device = vtkm::cont::DeviceAdapterTagAny())
```

Construct a `Keys` class from an array of keys.

Given an array of keys, construct a `Keys` class that will manage using these keys to perform reduce-by-key operations.

The input keys object is not modified and the result is not stable sorted. This is the equivalent of calling `BuildArrays(keys, KeysSortType::Unstable, device)`.

```
template<typename KeyArrayType>
```

```
void BuildArrays(const KeyArrayType &keys, KeysSortType sort, vtkm::cont::DeviceAdapterId device =  
                vtkm::cont::DeviceAdapterTagAny())
```

Build the internal arrays without modifying the input.

This is more efficient for stable sorted arrays, but requires an extra copy of the keys for unstable sorting.

```
template<typename KeyArrayType>
```

```
void BuildArraysInPlace(KeyArrayType &keys, KeysSortType sort, vtkm::cont::DeviceAdapterId device =  
                        vtkm::cont::DeviceAdapterTagAny())
```

Build the internal arrays and also sort the input keys.

This is more efficient for unstable sorting, but requires an extra copy for stable sorting.

```
inline KeyArrayHandleType GetUniqueKeys() const
```

Returns an array of unique keys.

The order of keys in this array describes the order that result values will be placed in a `vtkm::worklet::WorkletReduceByKey`.

```
vtkm::Id GetInputRange() const
```

Returns the input range of a keys object when used as an input domain.

This will be equal to the number of unique keys.

```
vtkm::cont::ArrayHandle<vtkm::Id> GetSortedValuesMap() const
```

Returns the array that maps each input value to an array of sorted keys.

This array is used internally as the indices to a `vtkm::cont::ArrayHandlePermutation` to order input values with the grouped keys so that they can then be grouped. This is an internal array that is seldom of use to code outside the `vtkm::worklet::WorkletReduceByKey` implementation.

`vtkm::cont::ArrayHandle<vtkm::Id> GetOffsets() const`

Returns an offsets array to group keys.

Given an array of sorted keys (or more frequently values permuted to the sorting of the keys), this array of indices can be used as offsets for a `vtkm::cont::ArrayHandleGroupVecVariable`. This is an internal array that is seldom of use to code outside the `vtkm::worklet::WorkletReduceByKey` implementation.

`vtkm::Id GetNumberOfValues() const`

Returns the number of input keys and values used to build this structure.

This is also the size of input arrays to a `vtkm::worklet::WorkletReduceByKey`.

### 22.4.3 Reduce by Key Examples

As stated earlier, the reduce by key worklet is useful for collecting like values. To demonstrate the reduce by key worklet, we will create a simple mechanism to generate a histogram in parallel. (VTK-m comes with its own histogram implementation, but we create our own version here for a simple example.) The way we can use the reduce by key worklet to compute a histogram is to first identify which bin of the histogram each value is in, and then use the bin identifiers as the keys to collect the information. To help with this example, we will first create a helper class named `BinScalars` that helps us manage the bins.

Example 8: A helper class to manage histogram bins.

```

1  class BinScalars
2  {
3  public:
4      VTKM_EXEC_CONT
5      BinScalars(const vtkm::Range& range, vtkm::Id numBins)
6          : Range(range)
7            , NumBins(numBins)
8      {
9      }
10
11     VTKM_EXEC_CONT
12     BinScalars(const vtkm::Range& range, vtkm::Float64 tolerance)
13         : Range(range)
14     {
15         this->NumBins = vtkm::Id(this->Range.Length() / tolerance) + 1;
16     }
17
18     VTKM_EXEC_CONT
19     vtkm::Id GetBin(vtkm::Float64 value) const
20     {
21         vtkm::Float64 ratio = (value - this->Range.Min) / this->Range.Length();
22         vtkm::Id bin = vtkm::Id(ratio * this->NumBins);
23         bin = vtkm::Max(bin, vtkm::Id(0));
24         bin = vtkm::Min(bin, this->NumBins - 1);
25         return bin;
26     }
27
28 private:
29     vtkm::Range Range;
30     vtkm::Id NumBins;
31 };

```

Using this helper class, we can easily create a simple map worklet that takes values, identifies a bin, and writes that result out to an array that can be used as keys.

Example 9: A simple map worklet to identify histogram bins, which will be used as keys.

```

1 struct IdentifyBins : vtkm::worklet::WorkletMapField
2 {
3     using ControlSignature = void(FieldIn data, FieldOut bins);
4     using ExecutionSignature = _2(_1);
5     using InputDomain = _1;
6
7     BinScalars Bins;
8
9     VTKM_CONT
10    IdentifyBins(const BinScalars& bins)
11        : Bins(bins)
12    {
13    }
14
15    VTKM_EXEC
16    vtkm::Id operator()(vtkm::Float64 value) const { return Bins.GetBin(value); }
17 };

```

Once you generate an array to be used as keys, you need to make a `vtkm::worklet::Keys` object. The `vtkm::worklet::Keys` object is what will be passed to the `vtkm::cont::Invoker` for the argument associated with the `KeysIn` ControlSignature tag. This of course happens in the control environment after calling the `vtkm::cont::Invoker` for our worklet for generating the keys.

Example 10: Creating a `vtkm::worklet::Keys` object.

```

1 vtkm::cont::ArrayHandle<vtkm::Id> binIds;
2 this->Invoke(IdentifyBins(bins), valuesArray, binIds);
3
4 vtkm::worklet::Keys<vtkm::Id> keys(binIds);

```

Now that we have our keys, we are finally ready for our reduce by key worklet. A histogram is simply a count of the number of elements in a bin. In this case, we do not really need any values for the keys. We just need the size of the bin, which can be identified with the internally calculated `ValueCount`.

A complication we run into with this histogram filter is that it is possible for a bin to be empty. If a bin is empty, there will be no key associated with that bin, and the `vtkm::cont::Invoker` will not call the worklet for that bin/key. To manage this case, we have to initialize an array with 0's and then fill in the non-zero entities with our reduce by key worklet. We can find the appropriate entry into the array by using the key, which is actually the bin identifier, which doubles as an index into the histogram. The following example gives the implementation for the reduce by key worklet that fills in positive values of the histogram.

Example 11: A reduce by key worklet to write histogram bin counts.

```

1 struct CountBins : vtkm::worklet::WorkletReduceByKey
2 {
3     using ControlSignature = void(KeysIn keys, WholeArrayOut binCounts);
4     using ExecutionSignature = void(_1, ValueCount, _2);
5     using InputDomain = _1;
6

```

(continues on next page)

(continued from previous page)

```

7  template<typename BinCountsPortalType>
8  VTKM_EXEC void operator()(vtkm::Id binId,
9                           vtkm::IdComponent numValuesInBin,
10                          BinCountsPortalType& binCounts) const
11  {
12      binCounts.Set(binId, numValuesInBin);
13  }
14  };

```

The previous example demonstrates the basic usage of the reduce by key worklet to count common keys. A more common use case is to collect values associated with those keys, do an operation on those values, and provide a “reduced” value for each unique key. The following example demonstrates such an operation by providing a worklet that finds the average of all values in a particular bin rather than counting them.

Example 12: A worklet that averages all values with a common key.

```

1  struct BinAverage : vtkm::worklet::WorkletReduceByKey
2  {
3      using ControlSignature = void(KeysIn keys,
4                                    ValuesIn originalValues,
5                                    ReducedValuesOut averages);
6      using ExecutionSignature = _3(_2);
7      using InputDomain = _1;
8
9      template<typename OriginalValuesVecType>
10     VTKM_EXEC typename OriginalValuesVecType::ComponentType operator()(
11         const OriginalValuesVecType& originalValues) const
12     {
13         typename OriginalValuesVecType::ComponentType sum = 0;
14         for (vtkm::IdComponent index = 0; index < originalValues.GetNumberOfComponents();
15             index++)
16         {
17             sum = sum + originalValues[index];
18         }
19         return sum / originalValues.GetNumberOfComponents();
20     }
21 };

```

To complete the code required to average all values that fall into the same bin, the following example shows the full code required to invoke such a worklet. Note that this example repeats much of the previous examples, but shows it in a more complete context.

Example 13: Using a reduce by key worklet to average values falling into the same bin.

```

1  struct IdentifyBins : vtkm::worklet::WorkletMapField
2  {
3      using ControlSignature = void(FieldIn data, FieldOut bins);
4      using ExecutionSignature = _2(_1);
5      using InputDomain = _1;
6
7      BinScalars Bins;
8

```

(continues on next page)

(continued from previous page)

```

9      VTKM_CONT
10      IdentifyBins(const BinScalars& bins)
11          : Bins(bins)
12      {
13      }
14
15      VTKM_EXEC
16      vtkm::Id operator()(vtkm::Float64 value) const { return Bins.GetBin(value); }
17  };
18
19  struct BinAverage : vtkm::worklet::WorkletReduceByKey
20  {
21      using ControlSignature = void(KeysIn keys,
22                                   ValuesIn originalValues,
23                                   ReducedValuesOut averages);
24      using ExecutionSignature = _3(_2);
25      using InputDomain = _1;
26
27      template<typename OriginalValuesVecType>
28      VTKM_EXEC typename OriginalValuesVecType::ComponentType operator()(
29          const OriginalValuesVecType& originalValues) const
30      {
31          typename OriginalValuesVecType::ComponentType sum = 0;
32          for (vtkm::IdComponent index = 0; index < originalValues.GetNumberOfComponents();
33              index++)
34          {
35              sum = sum + originalValues[index];
36          }
37          return sum / originalValues.GetNumberOfComponents();
38      }
39  };
40
41  //
42  // Later in the associated Filter class...
43  //
44
45  vtkm::Range range = vtkm::cont::ArrayRangeCompute(inField).ReadPortal().Get(0);
46  BinScalars bins(range, numBins);
47
48  vtkm::cont::ArrayHandle<vtkm::Id> binIds;
49  this->Invoke(IdentifyBins(bins), inField, binIds);
50
51  vtkm::worklet::Keys<vtkm::Id> keys(binIds);
52
53  vtkm::cont::ArrayHandle<T> combinedValues;
54
55  this->Invoke(BinAverage{}, keys, inField, combinedValues);

```

## EXTENDED FILTER IMPLEMENTATIONS

In [Chapter 18 \(Simple Worklets\)](#) and [Chapter 22 \(Worklet Types\)](#) we discuss how to implement an algorithm in the VTK-m framework by creating a worklet. For simplicity, worklet algorithms are wrapped in what are called filter objects for general usage. [Chapter 9 \(Running Filters\)](#) introduces the concept of filters, and [Chapter 10 \(Provided Filters\)](#) documents those that come with the VTK-m library. [Chapter 19 \(Basic Filter Implementation\)](#) gives a brief introduction on implementing filters. This chapter elaborates on building new filter objects by introducing new filter types. These will be used to wrap filters around the extended worklet examples in [Chapter 22 \(Worklet Types\)](#).

Unsurprisingly, the base filter objects are contained in the `vtkm::filter` package. In particular, all filter objects inherit from `vtkm::filter::Filter`, either directly or indirectly. The filter implementation must override the protected pure virtual method `vtkm::filter::Filter::DoExecute()`. The base class will call this method to run the operation of the filter.

The `vtkm::filter::Filter::DoExecute()` method has a single argument that is a `vtkm::cont::DataSet`. The `vtkm::cont::DataSet` contains the data on which the filter will operate. `vtkm::filter::Filter::DoExecute()` must then return a new `vtkm::cont::DataSet` containing the derived data. The `vtkm::cont::DataSet` should be created with one of the `vtkm::filter::Filter::CreateResult()` methods.

A filter implementation may also optionally override the `vtkm::filter::Filter::DoExecutePartitions()`. This method is similar to `vtkm::filter::Filter::DoExecute()` except that it takes and returns a `vtkm::cont::PartitionedDataSet` object. If a filter does not provide a `vtkm::filter::Filter::DoExecutePartitions()` method, then if given a `vtkm::cont::PartitionedDataSet`, the base class will call `vtkm::filter::Filter::DoExecute()` on each of the partitions and build a `vtkm::cont::PartitionedDataSet` with the results.

In addition to (or instead of) operating on the geometric structure of a `vtkm::cont::DataSet`, a filter will commonly take one or more fields from the input `vtkm::cont::DataSet` and write one or more fields to the result. For this reason, `vtkm::filter::Filter` provides convenience methods to select input fields and output field names.

It also provides a method named `vtkm::filter::Filter::GetFieldFromDataSet()` that can be used to get the input fields from the `vtkm::cont::DataSet` passed to `vtkm::filter::Filter::DoExecute()`. When getting a field with `vtkm::filter::Filter::GetFieldFromDataSet()`, you get a `vtkm::cont::Field` object. Before you can operate on the `vtkm::cont::Field`, you have to convert it to a `vtkm::cont::ArrayHandle`. `vtkm::filter::Filter::CastAndCallScalarField()` can be used to do this conversion. It takes the field object as the first argument and attempts to convert it to an `vtkm::cont::ArrayHandle` of different types. When it finds the correct type, it calls the provided functor with the appropriate `vtkm::cont::ArrayHandle`. The similar `vtkm::filter::Filter::CastAndCallVecField()` does the same thing to find an `vtkm::cont::ArrayHandle` with `vtkm::Vec`'s of a selected length, and `vtkm::filter::Filter::CastAndCallVariableVecField()` does the same thing but will find `vtkm::Vec`'s of any length.

The remainder of this chapter will provide some common patterns of filter operation based on the data they use and generate.

## 23.1 Deriving Fields from other Fields

A common type of filter is one that generates a new field that is derived from one or more existing fields or point coordinates on the data set. For example, mass, volume, and density are interrelated, and any one can be derived from the other two. Typically, you would use `vtkm::filter::Filter::GetFieldFromDataSet()` to retrieve the input fields, one of the `vtkm::filter::Filter::CastAndCall()` methods to resolve the array type of the field, and finally use `vtkm::filter::Filter::CreateResultField()` to produce the output.

In this section we provide an example implementation of a field filter that wraps the “magnitude” worklet provided in [Example 1](#). By C++ convention, object implementations are split into two files. The first file is a standard header file with a `.h` extension that contains the declaration of the filter class without the implementation. So we would expect the following code to be in a file named `FieldMagnitude.h`.

Example 1: Header declaration for a field filter.

```

1 namespace vtkm
2 {
3   namespace filter
4   {
5     namespace vector_calculus
6     {
7
8       class VTKM_FILTER_VECTOR_CALCULUS_EXPORT FieldMagnitude : public vtkm::filter::Filter
9       {
10      public:
11        VTKM_CONT FieldMagnitude();
12
13        VTKM_CONT vtkm::cont::DataSet DoExecute(const vtkm::cont::DataSet& inDataSet) override;
14      };
15
16    } // namespace vector_calculus
17  } // namespace filter
18 } // namespace vtkm

```

You may notice in [Example 1](#), line 8 there is a special macro names `VTKM_FILTER_VECTOR_CALCULUS_EXPORT`. This macro tells the C++ compiler that the class `FieldMagnitude` is going to be exported from a library. More specifically, the CMake for VTK-m’s build will generate a header file containing this export macro for the associated library. By VTK-m’s convention, a filter in the `vtkm::filter::vector_calculus` will be defined in the `vtkm/filter/vector_calculus` directory. When defining the targets for this library, CMake will create a header file named `vtkm_filter_vector_calculus.h` that contains the macro named `VTKM_FILTER_VECTOR_CALCULUS_EXPORT`. This macro will provide the correct modifiers for the particular C++ compiler being used to export the class from the library. If this macro is left out, then the library will work on some platforms, but on other platforms will produce a linker error for missing symbols.

Once the filter class is declared in the `.h` file, the implementation filter is by convention given in a separate `.cxx` file. So the continuation of our example that follows would be expected in a file named `FieldMagnitude.cxx`.

Example 2: Implementation of a field filter.

```

1 namespace vtkm
2 {
3   namespace filter
4   {
5     namespace vector_calculus

```

(continues on next page)



(continued from previous page)

```

6 {
7
8 VTKM_CONT
9 FieldMagnitude::FieldMagnitude()
10 {
11     this->SetOutputFieldName("");
12 }
13
14 VTKM_CONT vtkm::cont::DataSet FieldMagnitude::DoExecute(
15     const vtkm::cont::DataSet& inDataSet)
16 {
17     vtkm::cont::Field inField = this->GetFieldFromDataSet(inDataSet);
18
19     vtkm::cont::UnknownArrayHandle outField;
20
21     // Use a C++ lambda expression to provide a callback for CastAndCall. The lambda
22     // will capture references to local variables like outFieldArray (using ` [&] `)
23     // that it can read and write.
24     auto resolveType = [&](const auto& inFieldArray) {
25         using InArrayHandleType = std::decay_t<decltype(inFieldArray)>;
26         using ComponentType =
27             typename vtkm::VecTraits<typename InArrayHandleType::ValueType>::ComponentType;
28
29         vtkm::cont::ArrayHandle<ComponentType> outFieldArray;
30
31         this->Invoke(ComputeMagnitude{}, inFieldArray, outFieldArray);
32         outField = outFieldArray;
33     };
34
35     this->CastAndCallVecField<3>(inField, resolveType);
36
37     std::string outFieldName = this->GetOutputFieldName();
38     if (outFieldName == "")
39     {
40         outFieldName = inField.GetName() + "_magnitude";
41     }
42
43     return this->CreateResultField(
44         inDataSet, outFieldName, inField.GetAssociation(), outField);
45 }
46
47 } // namespace vector_calculus
48 } // namespace filter
49 } // namespace vtkm

```

The implementation of `vtkm::filter::Filter::DoExecute()` first pulls the input field from the provided `vtkm::cont::DataSet` using `vtkm::filter::Filter::GetFieldFromDataSet()`. It then uses `vtkm::filter::Filter::CastAndCallVecField()` to determine what type of `vtkm::cont::ArrayHandle` is contained in the input field. That calls a lambda function that invokes a worklet to create the output field.

```

template<vtkm::IdComponent VecSize, typename Functor, typename ...Args>
inline void vtkm::filter::Filter::CastAndCallVecField(const vtkm::cont::UnknownArrayHandle
&fieldArray, Functor &&functor, Args&&... args)
const

```

Convenience method to get the array from a filter's input vector field.

A field filter typically gets its input fields using the internal `GetFieldFromDataSet`. To use this field in a worklet, it eventually needs to be converted to an `vtkm::cont::ArrayHandle`. If the input field is limited to be a vector field with vectors of a specific size, then this method provides a convenient way to determine the correct array type. Like other `CastAndCall` methods, it takes as input a `vtkm::cont::Field` (or `vtkm::cont::UnknownArrayHandle`) and a function/functor to call with the appropriate `vtkm::cont::ArrayHandle` type. You also have to provide the vector size as the first template argument. For example `CastAndCallVecField<3>(field, functor);`.

```
template<vtkm::IdComponent VecSize, typename Functor, typename ...Args>
inline void vtkm::filter::Filter::CastAndCallVecField(const vtkm::cont::Field &field, Functor
                                                    &&functor, Args&&... args) const
```

Convenience method to get the array from a filter's input vector field.

A field filter typically gets its input fields using the internal `GetFieldFromDataSet`. To use this field in a worklet, it eventually needs to be converted to an `vtkm::cont::ArrayHandle`. If the input field is limited to be a vector field with vectors of a specific size, then this method provides a convenient way to determine the correct array type. Like other `CastAndCall` methods, it takes as input a `vtkm::cont::Field` (or `vtkm::cont::UnknownArrayHandle`) and a function/functor to call with the appropriate `vtkm::cont::ArrayHandle` type. You also have to provide the vector size as the first template argument. For example `CastAndCallVecField<3>(field, functor);`.

---

### Did You Know?

The filter implemented in [Example 2](#) is limited to only find the magnitude of `vtkm::Vec`'s with 3 components. It may be the case you wish to implement a filter that operates on `vtkm::Vec`'s of multiple sizes (or perhaps even any size). [Chapter ref{chap:UnknownArrayHandle}](#) discusses how you can use the `vtkm::cont::UnknownArrayHandle` contained in the `vtkm::cont::Field` to more expressively decide what types to check for.

---

```
template<typename Functor, typename ...Args>
inline void vtkm::filter::Filter::CastAndCallVariableVecField(const
                                                            vtkm::cont::UnknownArrayHandle
                                                            &fieldArray, Functor &&functor,
                                                            Args&&... args) const
```

This method is like `CastAndCallVecField` except that it can be used for a field of unknown vector size (or scalars).

This method will call the given functor with an `vtkm::cont::ArrayHandleRecombineVec`.

Note that there are limitations with using `vtkm::cont::ArrayHandleRecombineVec` within a worklet. Because the size of the vectors are not known at compile time, you cannot just create an intermediate `vtkm::Vec` of the correct size. Typically, you must allocate the output array (for example, with `vtkm::cont::ArrayHandleRuntimeVec`), and the worklet must iterate over the components and store them in the preallocated output.

```
template<typename Functor, typename ...Args>
inline void vtkm::filter::Filter::CastAndCallVariableVecField(const vtkm::cont::Field &field, Functor
                                                            &&functor, Args&&... args) const
```

This method is like `CastAndCallVecField` except that it can be used for a field of unknown vector size (or scalars).

This method will call the given functor with an `vtkm::cont::ArrayHandleRecombineVec`.

Note that there are limitations with using `vtkm::cont::ArrayHandleRecombineVec` within a worklet. Because the size of the vectors are not known at compile time, you cannot just create an intermedi-

ate `vtkm::Vec` of the correct size. Typically, you must allocate the output array (for example, with `vtkm::cont::ArrayHandleRuntimeVec`), and the worklet must iterate over the components and store them in the preallocated output.

Finally, `vtkm::filter::Filter::CreateResultField()` generates the output of the filter. Note that all fields need a unique name, which is the reason for the second argument to `vtkm::filter::Filter::CreateResult()`. The `vtkm::filter::Filter` base class contains a pair of methods named `vtkm::filter::Filter::SetOutputFieldName()` and `vtkm::filter::Filter::GetOutputFieldName()` to allow users to specify the name of output fields. The `vtkm::filter::Filter::DoExecute()` method should respect the given output field name. However, it is also good practice for the filter to have a default name if none is given. This might be simply specifying a name in the constructor, but it is worthwhile for many filters to derive a name based on the name of the input field.

## 23.2 Deriving Fields from Topology

The previous example performed a simple operation on each element of a field independently. However, it is also common for a “field” filter to take into account the topology of a data set. In this case, the implementation involves pulling a `vtkm::cont::CellSet` from the input `vtkm::cont::DataSet` and performing operations on fields associated with different topological elements. The steps involve calling `vtkm::cont::DataSet::GetCellSet()` to get access to the `vtkm::cont::CellSet` object and then using topology-based worklets, described in [Section 22.2 \(Topology Map\)](#), to operate on them.

In this section we provide an example implementation of a field filter on cells that wraps the “cell center” worklet provided in [Example 3](#).

Example 3: Header declaration for a field filter using cell topology.

```

1 namespace vtkm
2 {
3   namespace filter
4   {
5     namespace field_conversion
6     {
7
8       class VTKM_FILTER_FIELD_CONVERSION_EXPORT CellCenters : public vtkm::filter::Filter
9       {
10      public:
11        VTKM_CONT CellCenters();
12
13        VTKM_CONT vtkm::cont::DataSet DoExecute(const vtkm::cont::DataSet& inDataSet) override;
14      };
15
16    } // namespace field_conversion
17  } // namespace filter
18 } // namespace vtkm

```

As with any subclass of `vtkm::filter::Filter`, the filter implements `vtkm::filter::Filter::DoExecute()`, which in this case invokes a worklet to compute a new field array and then return a newly constructed `vtkm::cont::DataSet` object.

Example 4: Implementation of a field filter using cell topology.

```

1 namespace vtkm
2 {
3   namespace filter
4   {
5     namespace field_conversion
6     {
7
8       VTKM_CONT
9       CellCenters::CellCenters()
10      {
11        this->SetOutputFieldName("");
12      }
13
14       VTKM_CONT cont::DataSet CellCenters::DoExecute(const vtkm::cont::DataSet& inDataSet)
15      {
16        vtkm::cont::Field inField = this->GetFieldFromDataSet(inDataSet);
17
18        if (!inField.IsPointField())
19        {
20          throw vtkm::cont::ErrorBadType("Cell Centers filter operates on point data.");
21        }
22
23        vtkm::cont::UnknownArrayHandle outUnknownArray;
24
25        auto resolveType = [&](const auto& inArray) {
26          using InArrayHandleType = std::decay_t<decltype(inArray)>;
27          using ValueType = typename InArrayHandleType::ValueType;
28          vtkm::cont::ArrayHandle<ValueType> outArray;
29
30          this->Invoke(vtkm::worklet::CellCenter{}, inDataSet.GetCellSet(), inArray, outArray);
31
32          outUnknownArray = outArray;
33        };
34
35        vtkm::cont::UnknownArrayHandle inUnknownArray = inField.GetData();
36        inUnknownArray.CastAndCallForTypesWithFloatFallback<VTKM_DEFAULT_TYPE_LIST,
37                                                             VTKM_DEFAULT_STORAGE_LIST>(
38          resolveType);
39
40        std::string outFieldName = this->GetOutputFieldName();
41        if (outFieldName == "")
42        {
43          outFieldName = inField.GetName() + "_center";
44        }
45
46        return this->CreateResultFieldCell(inDataSet, outFieldName, outUnknownArray);
47      }
48
49    } // namespace field_conversion
50  } // namespace filter
51 } // namespace vtkm

```

## 23.3 Data Set Filters

Sometimes, a filter will generate a data set with a new cell set based off the cells of an input data set. For example, a data set can be significantly altered by adding, removing, or replacing cells.

As with any filter, data set filters can be implemented in classes that derive the `vtkm::filter::Filter` base class and implement its `vtkm::filter::Filter::DoExecute()` method.

In this section we provide an example implementation of a data set filter that wraps the functionality of extracting the edges from a data set as line elements. Many variations of implementing this functionality are given in Chapter~ref{chap:GeneratingCellSets}. Suffice it to say that a pair of worklets will be used to create a new `vtkm::cont::CellSet`, and this `vtkm::cont::CellSet` will be used to create the result `vtkm::cont::DataSet`. Details on how the worklets work are given in Section~ref{sec:GeneratingCellSets:SingleType}.

Because the operation of this edge extraction depends only on `vtkm::cont::CellSet` in a provided `vtkm::cont::DataSet`, the filter class is a simple subclass of `vtkm::filter::Filter`.

Example 5: Header declaration for a data set filter.

```

1 namespace vtkm
2 {
3   namespace filter
4   {
5     namespace entity_extraction
6     {
7
8       class VTKM_FILTER_ENTITY_EXTRACTION_EXPORT ExtractEdges : public vtkm::filter::Filter
9       {
10      public:
11        VTKM_CONT vtkm::cont::DataSet DoExecute(const vtkm::cont::DataSet& inData) override;
12      };
13
14    } // namespace entity_extraction
15  } // namespace filter
16 } // namespace vtkm

```

The implementation of `vtkm::filter::Filter::DoExecute()` first gets the `vtkm::cont::CellSet` and calls the worklet methods to generate a new `vtkm::cont::CellSet` class. It then uses a form of `vtkm::filter::Filter::CreateResult()` to generate the resulting `vtkm::cont::DataSet`.

Example 6: Implementation of the `vtkm::filter::Filter::DoExecute()` method of a data set filter.

```

1 inline VTKM_CONT vtkm::cont::DataSet ExtractEdges::DoExecute(
2   const vtkm::cont::DataSet& inData)
3 {
4   auto inCellSet = inData.GetCellSet();
5
6   // Count number of edges in each cell.
7   vtkm::cont::ArrayHandle<vtkm::IdComponent> edgeCounts;
8   this->Invoke(vtkm::worklet::CountEdgesWorklet{}, inCellSet, edgeCounts);
9
10  // Build the scatter object (for non 1-to-1 mapping of input to output)
11  vtkm::worklet::ScatterCounting scatter(edgeCounts);

```

(continues on next page)

(continued from previous page)

```

12  auto outputToInputCellMap = scatter.GetOutputToInputMap(inCellSet.GetNumberOfCells());
13
14  vtkm::cont::ArrayHandle<vtkm::Id> connectivityArray;
15  this->Invoke(vtkm::worklet::EdgeIndicesWorklet{},
16              scatter,
17              inCellSet,
18              vtkm::cont::make_ArrayHandleGroupVec<2>(connectivityArray));
19
20  vtkm::cont::CellSetSingleType<> outCellSet;
21  outCellSet.Fill(
22      inCellSet.GetNumberOfPoints(), vtkm::CELL_SHAPE_LINE, 2, connectivityArray);
23
24  // This lambda function maps an input field to the output data set. It is
25  // used with the CreateResult method.
26  auto fieldMapper = [&](vtkm::cont::DataSet& outData,
27                        const vtkm::cont::Field& inputField) {
28      if (inputField.IsCellField())
29      {
30          vtkm::filter::MapFieldPermutation(inputField, outputToInputCellMap, outData);
31      }
32      else
33      {
34          outData.AddField(inputField); // pass through
35      }
36  };
37
38  return this->CreateResult(inData, outCellSet, fieldMapper);
39  }

```

The form of `vtkm::filter::Filter::CreateResult()` used (Example 6, line 38) takes as input a `vtkm::cont::CellSet` to use in the generated data. In forms of `vtkm::filter::Filter::CreateResult()` used in previous examples of this chapter, the cell structure of the output was created from the cell structure of the input. Because these cell structures were the same, coordinate systems and fields needed to be changed. However, because we are providing a new `vtkm::cont::CellSet`, we need to also specify how the coordinate systems and fields change.

The last two arguments to `vtkm::filter::Filter::CreateResult()` are providing this information. The second-to-last argument is a `std::vector` of the `vtkm::cont::CoordinateSystem`'s to use. Because this filter does not actually change the points in the data set, the `vtkm::cont::CoordinateSystem`'s can just be copied over. The last argument provides a functor that maps a field from the input to the output. The functor takes two arguments: the output `vtkm::cont::DataSet` to modify and the input `vtkm::cont::Field` to map. In this example, the functor is defined as a lambda function (Example 6, line 26).

### Did You Know?

The field mapper in Example 5 uses a helper function named `vtkm::filter::MapFieldPermutation()`. In the case of this example, every cell in the output comes from one cell in the input. For this common case, the values in the field arrays just need to be permuted so that each input value gets to the right output value. `vtkm::filter::MapFieldPermutation()` will do this shuffling for you.

VTK-m also comes with a similar helper function `vtkm::filter::MapFieldMergeAverage()` that can be used when each output cell (or point) was constructed from multiple inputs. In this case, `vtkm::filter::MapFieldMergeAverage()` can do a simple average for each output value of all input values that contributed.

```
bool vtkm::filter::MapFieldPermutation(const vtkm::cont::Field &inputField, const
                                       vtkm::cont::ArrayHandle<vtkm::Id> &permutation,
                                       vtkm::cont::Field &outputField, vtkm::Float64 invalidValue =
                                       vtkm::Nan<vtkm::Float64>())
```

Maps a field by permuting it by a given index array.

This method will create a new field containing the data from the provided `inputField` but reordered by the given `permutation` index array. The value in the resulting field for index  $i$  will be a value from `inputField`, but comes from the index that comes from `permutation` at position  $i$ . The result is placed in `outputField`.

The intention of this method is to implement the mapping of fields from the input to the output in filters (many of which require this permutation of a field), but can be used in other places as well.

`outputField` is set to have the same metadata as the input. If the metadata needs to change (such as the name or the association) that should be done after the function returns.

This function returns whether the field was successfully permuted. If the returned result is `true`, then the results in `outputField` are valid. If it is `false`, then `outputField` should not be used.

If an invalid index is given in the permutation array (i.e. less than 0 or greater than the size of the array), then the resulting `outputField` will be given `invalidValue` (converted as best as possible to the correct data type).

```
bool vtkm::filter::MapFieldPermutation(const vtkm::cont::Field &inputField, const
                                       vtkm::cont::ArrayHandle<vtkm::Id> &permutation,
                                       vtkm::cont::DataSet &outputData, vtkm::Float64 invalidValue =
                                       vtkm::Nan<vtkm::Float64>())
```

Maps a field by permuting it by a given index array.

This method will create a new field containing the data from the provided `inputField` but reordered by the given `permutation` index array. The value in the resulting field for index  $i$  will be a value from `inputField`, but comes from the index that comes from `permutation` at position  $i$ .

The intention of this method is to implement the `MapFieldOntoOutput` methods in filters (many of which require this permutation of a field), but can be used in other places as well. The resulting field is put in the given `DataSet`.

The returned `Field` has the same metadata as the input. If the metadata needs to change (such as the name or the association), then a different form of `MapFieldPermutation` should be used.

This function returns whether the field was successfully permuted. If the returned result is `true`, then `outputData` has the permuted field. If it is `false`, then the field is not placed in `outputData`.

If an invalid index is given in the permutation array (i.e. less than 0 or greater than the size of the array), then the resulting `outputField` will be given `invalidValue` (converted as best as possible to the correct data type).

```
bool vtkm::filter::MapFieldMergeAverage(const vtkm::cont::Field &inputField, const
                                       vtkm::worklet::internal::KeysBase &keys, vtkm::cont::Field
                                       &outputField)
```

Maps a field by merging entries based on a keys object.

This method will create a new field containing the data from the provided `inputField` but with groups of entities merged together. The input `keys` object encapsulates which elements should be merged together. A group of elements merged together will be averaged. The result is placed in `outputField`.

The intention of this method is to implement the `MapFieldOntoOutput` methods in filters (many of which require this merge of a field), but can be used in other places as well.

`outputField` is set to have the same metadata as the input. If the metadata needs to change (such as the name or the association) that should be done after the function returns.



This function returns whether the field was successfully merged. If the returned result is `true`, then the results in `outputField` are valid. If it is `false`, then `outputField` should not be used.

```
bool vtkm::filter::MapFieldMergeAverage(const vtkm::cont::Field &inputField, const
                                       vtkm::worklet::internal::KeysBase &keys, vtkm::cont::DataSet
                                       &outputData)
```

Maps a field by merging entries based on a keys object.

This method will create a new field containing the data from the provided `inputField` but but with groups of entities merged together. The input `keys` object encapsulates which elements should be merged together. A group of elements merged together will be averaged.

The intention of this method is to implement the `MapFieldOntoOutput` methods in filters (many of which require this merge of a field), but can be used in other places as well. The resulting field is put in the given `DataSet`.

The returned `Field` has the same metadata as the input. If the metadata needs to change (such as the name or the association), then a different form of `MapFieldMergeAverage` should be used.

This function returns whether the field was successfully merged. If the returned result is `true`, then `outputData` has the merged field. If it is `false`, then the field is not placed in `outputData`.

---

### Did You Know?

Although not the case in this example, sometimes a filter creating a new cell set changes the points of the cells. As long as the field mapper you provide to `vtkm::filter::Filter::CreateResult()` properly converts points from the input to the output, all fields and coordinate systems will be automatically filled in the output. Sometimes when creating this new cell set you also create new point coordinates for it. This might be because the point coordinates are necessary for the computation or might be due to a faster way of computing the point coordinates. In either case, if the filter already has point coordinates computed, it can use `vtkm::filter::Filter::CreateResultCoordinateSystem()` to use the precomputed point coordinates.

---

## 23.4 Data Set with Field Filters

Sometimes, a filter will generate a data set with a new cell set based off the cells of an input data set along with the data in at least one field. For example, a field might determine how each cell is culled, clipped, or sliced.

In this section we provide an example implementation of a data set with field filter that blanks the cells in a data set based on a field that acts as a mask (or stencil). Any cell associated with a mask value of zero will be removed. For simplicity of this example, we will use the `vtkm::filter::entity_extraction::Threshold` filter internally for the implementation.

Example 7: Header declaration for a data set with field filter.

```
1 namespace vtkm
2 {
3   namespace filter
4   {
5     namespace entity_extraction
6     {
7
8     class VTKM_FILTER_ENTITY_EXTRACTION_EXPORT BlankCells : public vtkm::filter::Filter
9     {
10    public:
```

(continues on next page)



(continued from previous page)

```

11   VTKM_CONT vtkm::cont::DataSet DoExecute(const vtkm::cont::DataSet& inDataSet) override;
12 };
13
14
15 } // namespace entity_extraction
16 } // namespace filter
17 } // namespace vtkm

```

The implementation of `vtkm::filter::Filter::DoExecute()` first derives an array that contains a flag whether the input array value is zero or non-zero. This is simply to guarantee the range for the threshold filter. After that a threshold filter is set up and run to generate the result.

Example 8: Implementation of the `vtkm::filter::Filter::DoExecute()` method of a data set with field filter.

```

1  VTKM_CONT vtkm::cont::DataSet BlankCells::DoExecute(const vtkm::cont::DataSet& inData)
2  {
3      vtkm::cont::Field inField = this->GetFieldFromDataSet(inData);
4      if (!inField.IsCellField())
5      {
6          throw vtkm::cont::ErrorBadValue("Blanking field must be a cell field.");
7      }
8
9      // Set up this array to have a 0 for any cell to be removed and
10     // a 1 for any cell to keep.
11     vtkm::cont::ArrayHandle<vtkm::FloatDefault> blankingArray;
12
13     auto resolveType = [&](const auto& inFieldArray) {
14         auto transformArray =
15             vtkm::cont::make_ArrayHandleTransform(inFieldArray, vtkm::NotZeroInitialized{});
16         vtkm::cont::ArrayCopyDevice(transformArray, blankingArray);
17     };
18
19     this->CastAndCallScalarField(inField, resolveType);
20
21     // Make a temporary DataSet (shallow copy of the input) to pass blankingArray
22     // to threshold.
23     vtkm::cont::DataSet tempData = inData;
24     tempData.AddCellField("vtkm-blanking-array", blankingArray);
25
26     // Just use the Threshold filter to implement the actual cell removal.
27     vtkm::filter::entity_extraction::Threshold thresholdFilter;
28     thresholdFilter.SetLowerThreshold(0.5);
29     thresholdFilter.SetUpperThreshold(2.0);
30     thresholdFilter.SetActiveField("vtkm-blanking-array",
31                                   vtkm::cont::Field::Association::Cells);
32
33     // Make sure threshold filter passes all the fields requested, but not the
34     // blanking array.
35     thresholdFilter.SetFieldsToPass(this->GetFieldsToPass());
36     thresholdFilter.SetFieldsToPass("vtkm-blanking-array",

```

(continues on next page)

(continued from previous page)

```
37         vtkm::cont::Field::Association::Cells,  
38         vtkm::filter::FieldSelection::Mode::Exclude);  
39  
40     // Use the threshold filter to generate the actual output.  
41     return thresholdFilter.Execute(tempData);  
42 }
```

## WORKLET ERROR HANDLING

It is sometimes the case during the execution of an algorithm that an error condition can occur that causes the computation to become invalid. At such a time, it is important to raise an error to alert the calling code of the problem. Since VTK-m uses an exception mechanism to raise errors, we want an error in the execution environment to throw an exception.

However, throwing exceptions in a parallel algorithm is problematic. Some accelerator architectures, like CUDA, do not even support throwing exceptions. Even on architectures that do support exceptions, throwing them in a thread block can cause problems. An exception raised in one thread may or may not be thrown in another, which increases the potential for deadlocks, and it is unclear how uncaught exceptions progress through thread blocks.

VTK-m handles this problem by using a flag and check mechanism. When a worklet (or other subclass of `vtkm::exec::FunctorBase`) encounters an error, it can call its `vtkm::exec::FunctorBase::RaiseError()` method to flag the problem and record a message for the error. Once all the threads terminate, the scheduler checks for the error, and if one exists it throws a `vtkmcont{ErrorException}` exception in the control environment. Thus, calling `vtkm::exec::FunctorBase::RaiseError()` looks like an exception was thrown from the perspective of the control environment code that invoked it.

Example 1: Raising an error in the execution environment.

```
1 struct SquareRoot : vtkm::worklet::WorkletMapField
2 {
3 public:
4     using ControlSignature = void(FieldIn, FieldOut);
5     using ExecutionSignature = _2(_1);
6
7     template<typename T>
8     VTKM_EXEC T operator()(T x) const
9     {
10         if (x < 0)
11         {
12             this->RaiseError("Cannot take the square root of a negative number.");
13             return vtkm::Nan<T>();
14         }
15         return vtkm::Sqrt(x);
16     }
17 };
```

It is also worth noting that the `VTKM_ASSERT` macro described in [Section 12.2 \(Asserting Conditions\)](#) also works within worklets and other code running in the execution environment. Of course, a failed assert will terminate execution rather than just raise an error so is best for checking invalid conditions for debugging purposes.



## MATH

VTK-m comes with several math functions that tend to be useful for visualization algorithms. The implementation of basic math operations can vary subtly on different accelerators, and these functions provide cross platform support.

All math functions are located in the `vtkm` package. The functions are most useful in the execution environment, but they can also be used in the control environment when needed.

### 25.1 Basic Math

The `vtkm/Math.h` header file contains several math functions that replicate the behavior of the basic POSIX math functions as well as related functionality.

---

#### Did You Know?

When writing worklets, you should favor using these math functions provided by VTK-m over the standard math functions in `vtkm/Math.h`. VTK-m's implementation manages several compiling and efficiency issues when porting.

---

#### 25.1.1 Exponentials

```
inline vtkm::Float32 vtkm::Exp(vtkm::Float32 x)
```

Computes  $e^x$ , the base-e exponential of  $x$ .

```
inline vtkm::Float64 vtkm::Exp(vtkm::Float64 x)
```

Computes  $e^x$ , the base-e exponential of  $x$ .

```
template<typename T>
```

```
static inline detail::FloatingPointReturnType<T>::Type vtkm::Exp(const T &x)
```

Computes  $e^x$ , the base-e exponential of  $x$ .

```
template<typename T, vtkm::IdComponent N>
```

```
static inline vtkm::Vec<typename detail::FloatingPointReturnType<T>::Type, N> vtkm::Exp(const vtkm::Vec<T, N> &x)
```

Computes  $e^x$ , the base-e exponential of  $x$ .

```
static inline vtkm::Float32 vtkm::Exp10(vtkm::Float32 x)
```

Computes  $10^x$ , the base-10 exponential of  $x$ .

```
static inline vtkm::Float64 vtkm::Exp10(vtkm::Float64 x)
```

Computes  $10^x$ , the base-10 exponential of  $x$ .

```
template<typename T>
```

```
static inline vtkm::Float64 vtkm::Exp10(T x)
```

Computes  $10^x$ , the base-10 exponential of  $x$ .

```
template<typename T, vtkm::IdComponent N>
```

```
static inline vtkm::Vec<typename detail::FloatingPointReturnType<T>::Type, N> vtkm::Exp10(const vtkm::Vec<T, N> &x)
```

Computes  $10^x$ , the base-10 exponential of  $x$ .

```
inline vtkm::Float32 vtkm::Exp2(vtkm::Float32 x)
```

Computes  $2^x$ , the base-2 exponential of  $x$ .

```
inline vtkm::Float64 vtkm::Exp2(vtkm::Float64 x)
```

Computes  $2^x$ , the base-2 exponential of  $x$ .

```
template<typename T>
```

```
static inline detail::FloatingPointReturnType<T>::Type vtkm::Exp2(const T &x)
```

Computes  $2^x$ , the base-2 exponential of  $x$ .

```
template<typename T, vtkm::IdComponent N>
```

```
static inline vtkm::Vec<typename detail::FloatingPointReturnType<T>::Type, N> vtkm::Exp2(const vtkm::Vec<T, N> &x)
```

Computes  $2^x$ , the base-2 exponential of  $x$ .

```
inline vtkm::Float32 vtkm::ExpM1(vtkm::Float32 x)
```

Computes  $(e^x) - 1$ , the of base-e exponential of  $x$  then minus 1. The accuracy of this function is good even for very small values of  $x$ .

```
inline vtkm::Float64 vtkm::ExpM1(vtkm::Float64 x)
```

Computes  $(e^x) - 1$ , the of base-e exponential of  $x$  then minus 1. The accuracy of this function is good even for very small values of  $x$ .

```
template<typename T>
```

```
static inline detail::FloatingPointReturnType<T>::Type vtkm::ExpM1(const T &x)
```

Computes  $(e^x) - 1$ , the of base-e exponential of  $x$  then minus 1. The accuracy of this function is good even for very small values of  $x$ .

```
template<typename T, vtkm::IdComponent N>
```

```
static inline vtkm::Vec<typename detail::FloatingPointReturnType<T>::Type, N> vtkm::ExpM1(const vtkm::Vec<T, N> &x)
```

Computes  $(e^x) - 1$ , the of base-e exponential of  $x$  then minus 1. The accuracy of this function is good even for very small values of  $x$ .

```
inline vtkm::Float32 vtkm::Log(vtkm::Float32 x)
```

Computes the natural logarithm of  $x$ .

```
inline vtkm::Float64 vtkm::Log(vtkm::Float64 x)
```

Computes the natural logarithm of  $x$ .

```
template<typename T>
```

```
static inline detail::FloatingPointReturnType<T>::Type vtkm::Log(const T &x)
```

Computes the natural logarithm of  $x$ .

```
template<typename T, vtkm::IdComponent N>
```

```
static inline vtkm::Vec<typename detail::FloatingPointReturnType<T>::Type, N> vtkm::Log(const vtkm::Vec<T, N> &x)
```

Computes the natural logarithm of  $x$ .

```
inline vtkm::Float32 vtkm::Log10(vtkm::Float32 x)
```

Computes the logarithm base 10 of  $x$ .

```
inline vtkm::Float64 vtkm::Log10(vtkm::Float64 x)
```

Computes the logarithm base 10 of  $x$ .

```
template<typename T>
```

```
static inline detail::FloatingPointReturnType<T>::Type vtkm::Log10(const T &x)
```

Computes the logarithm base 10 of  $x$ .

```
template<typename T, vtkm::IdComponent N>
```

```
static inline vtkm::Vec<typename detail::FloatingPointReturnType<T>::Type, N> vtkm::Log10(const vtkm::Vec<T, N> &x)
```

Computes the logarithm base 10 of  $x$ .

```
inline vtkm::Float32 vtkm::Log1P(vtkm::Float32 x)
```

Computes the value of  $\log(1+x)$ . This method is more accurate for very small values of  $x$  than the `vtkm::Log` function.

```
inline vtkm::Float64 vtkm::Log1P(vtkm::Float64 x)
```

Computes the value of  $\log(1+x)$ . This method is more accurate for very small values of  $x$  than the `vtkm::Log` function.

```
template<typename T>
```

```
static inline detail::FloatingPointReturnType<T>::Type vtkm::Log1P(const T &x)
```

Computes the value of  $\log(1+x)$ . This method is more accurate for very small values of  $x$  than the `vtkm::Log` function.

```
template<typename T, vtkm::IdComponent N>
```

```
static inline vtkm::Vec<typename detail::FloatingPointReturnType<T>::Type, N> vtkm::Log1P(const vtkm::Vec<T, N> &x)
```

Computes the value of  $\log(1+x)$ . This method is more accurate for very small values of  $x$  than the `vtkm::Log` function.

```
inline vtkm::Float32 vtkm::Log2(vtkm::Float32 x)
```

Computes the logarithm base 2 of  $x$ .

```
inline vtkm::Float64 vtkm::Log2(vtkm::Float64 x)
```

Computes the logarithm base 2 of  $x$ .

```
template<typename T>
```

```
static inline detail::FloatingPointReturnType<T>::Type vtkm::Log2(const T &x)
```

Computes the logarithm base 2 of  $x$ .

```
template<typename T, vtkm::IdComponent N>
```

```
static inline vtkm::Vec<typename detail::FloatingPointReturnType<T>::Type, N> vtkm::Log2(const vtkm::Vec<T, N> &x)
```

Computes the logarithm base 2 of  $x$ .

```
static inline vtkm::Float32 vtkm::Pow(vtkm::Float32 x, vtkm::Float32 y)
```

Computes  $x$  raised to the power of  $y$ .

```
static inline vtkm::Float64 vtkm::Pow(vtkm::Float64 x, vtkm::Float64 y)
```

Computes  $x$  raised to the power of  $y$ .

## 25.1.2 Non-finites

```
template<typename T>
```

```
static inline T vtkm::Infinity()
```

Returns the representation for infinity. The result is greater than any other number except another infinity or NaN. When comparing two infinities or infinity to NaN, neither is greater than, less than, nor equal to the other. The `Infinity()` function is templated to specify either a 32 or 64 bit floating point number. The convenience functions `Infinity32()` and `Infinity64()` are non-templated versions that return the precision for a particular precision.

```
static inline vtkm::Float32 vtkm::Infinity32()
```

Returns the representation for infinity. The result is greater than any other number except another infinity or NaN. When comparing two infinities or infinity to NaN, neither is greater than, less than, nor equal to the other. The `Infinity()` function is templated to specify either a 32 or 64 bit floating point number. The convenience functions `Infinity32()` and `Infinity64()` are non-templated versions that return the precision for a particular precision.

```
static inline vtkm::Float64 vtkm::Infinity64()
```

Returns the representation for infinity. The result is greater than any other number except another infinity or NaN. When comparing two infinities or infinity to NaN, neither is greater than, less than, nor equal to the other. The `Infinity()` function is templated to specify either a 32 or 64 bit floating point number. The convenience functions `Infinity32()` and `Infinity64()` are non-templated versions that return the precision for a particular precision.

```
template<typename T>
```

```
static inline bool vtkm::IsFinite(T x)
```

Returns true if `x` is a normal number (not NaN or infinite).

```
template<typename T>
```

```
static inline bool vtkm::IsInf(T x)
```

Returns true if `x` is positive or negative infinity.

```
template<typename T>
```

```
static inline bool vtkm::IsNan(T x)
```

Returns true if `x` is not a number.

```
static inline bool vtkm::IsNegative(vtkm::Float32 x)
```

Returns true if `x` is less than zero, false otherwise.

```
static inline bool vtkm::IsNegative(vtkm::Float64 x)
```

Returns true if `x` is less than zero, false otherwise.

```
template<typename T>
```

```
static inline T vtkm::Nan()
```

Returns the representation for infinity. The result is greater than any other number except another infinity or NaN. When comparing two infinities or infinity to NaN, neither is greater than, less than, nor equal to the other. The `Nan()` function is templated to specify either a 32 or 64 bit floating point number. The convenience functions `Nan32()` and `Nan64()` are non-templated versions that return the precision for a particular precision.

```
static inline vtkm::Float32 vtkm::Nan32()
```

Returns the representation for infinity. The result is greater than any other number except another infinity or NaN. When comparing two infinities or infinity to NaN, neither is greater than, less than, nor equal to the other. The `Nan()` function is templated to specify either a 32 or 64 bit floating point number. The convenience functions `Nan32()` and `Nan64()` are non-templated versions that return the precision for a particular precision.



```
static inline vtkm::Float64 vtkm::Nan64()
```

Returns the representation for infinity. The result is greater than any other number except another infinity or NaN. When comparing two infinities or infinity to NaN, neither is greater than, less than, nor equal to the other. The Nan() function is templated to specify either a 32 or 64 bit floating point number. The convenience functions Nan32() and Nan64() are non-templated versions that return the precision for a particular precision.

```
template<typename T>
```

```
static inline T vtkm::NegativeInfinity()
```

Returns the representation for negative infinity. The result is less than any other number except another negative infinity or NaN. When comparing two negative infinities or negative infinity to NaN, neither is greater than, less than, nor equal to the other. The NegativeInfinity() function is templated to specify either a 32 or 64 bit floating point number. The convenience functions NegativeInfinity32() and NegativeInfinity64() are non-templated versions that return the precision for a particular precision.

```
static inline vtkm::Float32 vtkm::NegativeInfinity32()
```

Returns the representation for negative infinity. The result is less than any other number except another negative infinity or NaN. When comparing two negative infinities or negative infinity to NaN, neither is greater than, less than, nor equal to the other. The NegativeInfinity() function is templated to specify either a 32 or 64 bit floating point number. The convenience functions NegativeInfinity32() and NegativeInfinity64() are non-templated versions that return the precision for a particular precision.

```
static inline vtkm::Float64 vtkm::NegativeInfinity64()
```

Returns the representation for negative infinity. The result is less than any other number except another negative infinity or NaN. When comparing two negative infinities or negative infinity to NaN, neither is greater than, less than, nor equal to the other. The NegativeInfinity() function is templated to specify either a 32 or 64 bit floating point number. The convenience functions NegativeInfinity32() and NegativeInfinity64() are non-templated versions that return the precision for a particular precision.

### 25.1.3 Polynomials

```
inline vtkm::Float32 vtkm::Cbrt(vtkm::Float32 x)
```

Compute the cube root of  $x$ .

```
inline vtkm::Float64 vtkm::Cbrt(vtkm::Float64 x)
```

Compute the cube root of  $x$ .

```
template<typename T>
```

```
static inline detail::FloatingPointReturnType<T>::Type vtkm::Cbrt(const T &x)
```

Compute the cube root of  $x$ .

```
template<typename T, vtkm::IdComponent N>
```

```
static inline vtkm::Vec<typename detail::FloatingPointReturnType<T>::Type, N> vtkm::Cbrt(const vtkm::Vec<T, N> &x)
```

Compute the cube root of  $x$ .

```
template<typename T>
```

```
inline vtkm::Vec<T, 2> vtkm::QuadraticRoots(T a, T b, T c)
```

Solves  $ax^2 + bx + c = 0$ .

Only returns the real roots. If there are real roots, the first element of the pair is less than or equal to the second. If there are no real roots, both elements are NaNs. If VTK-m is compiled with FMA support, each root is accurate to 3 ulps; otherwise the discriminant is prone to catastrophic subtractive cancellation and no accuracy guarantees can be provided.

```
static inline vtkm::Float32 vtkm::RCbrt(vtkm::Float32 x)
```

Compute the reciprocal cube root of  $x$ . The result of this function is equivalent to  $1/\text{Cbrt}(x)$ . However, on some devices it is faster to compute the reciprocal cube root than the regular cube root. Thus, you should use this function whenever dividing by the cube root.

```
static inline vtkm::Float64 vtkm::RCbrt(vtkm::Float64 x)
```

Compute the reciprocal cube root of  $x$ . The result of this function is equivalent to  $1/\text{Cbrt}(x)$ . However, on some devices it is faster to compute the reciprocal cube root than the regular cube root. Thus, you should use this function whenever dividing by the cube root.

```
template<typename T>
```

```
static inline vtkm::Float64 vtkm::RCbrt(T x)
```

Compute the reciprocal cube root of  $x$ . The result of this function is equivalent to  $1/\text{Cbrt}(x)$ . However, on some devices it is faster to compute the reciprocal cube root than the regular cube root. Thus, you should use this function whenever dividing by the cube root.

```
template<typename T, vtkm::IdComponent N>
```

```
static inline vtkm::Vec<typename detail::FloatingPointReturnType<T>::Type, N> vtkm::RCbrt(const vtkm::Vec<T, N> &x)
```

Compute the reciprocal cube root of  $x$ . The result of this function is equivalent to  $1/\text{Cbrt}(x)$ . However, on some devices it is faster to compute the reciprocal cube root than the regular cube root. Thus, you should use this function whenever dividing by the cube root.

```
static inline vtkm::Float32 vtkm::RSqrt(vtkm::Float32 x)
```

Compute the reciprocal square root of  $x$ . The result of this function is equivalent to  $1/\text{Sqrt}(x)$ . However, on some devices it is faster to compute the reciprocal square root than the regular square root. Thus, you should use this function whenever dividing by the square root.

```
static inline vtkm::Float64 vtkm::RSqrt(vtkm::Float64 x)
```

Compute the reciprocal square root of  $x$ . The result of this function is equivalent to  $1/\text{Sqrt}(x)$ . However, on some devices it is faster to compute the reciprocal square root than the regular square root. Thus, you should use this function whenever dividing by the square root.

```
template<typename T>
```

```
static inline vtkm::Float64 vtkm::RSqrt(T x)
```

Compute the reciprocal square root of  $x$ . The result of this function is equivalent to  $1/\text{Sqrt}(x)$ . However, on some devices it is faster to compute the reciprocal square root than the regular square root. Thus, you should use this function whenever dividing by the square root.

```
template<typename T, vtkm::IdComponent N>
```

```
static inline vtkm::Vec<typename detail::FloatingPointReturnType<T>::Type, N> vtkm::RSqrt(const vtkm::Vec<T, N> &x)
```

Compute the reciprocal square root of  $x$ . The result of this function is equivalent to  $1/\text{Sqrt}(x)$ . However, on some devices it is faster to compute the reciprocal square root than the regular square root. Thus, you should use this function whenever dividing by the square root.

```
inline vtkm::Float32 vtkm::Sqrt(vtkm::Float32 x)
```

Compute the square root of  $x$ . On some hardware it is faster to find the reciprocal square root, so `RSqrt` should be used if you actually plan to divide by the square root.

```
inline vtkm::Float64 vtkm::Sqrt(vtkm::Float64 x)
```

Compute the square root of  $x$ . On some hardware it is faster to find the reciprocal square root, so `RSqrt` should be used if you actually plan to divide by the square root.

```
template<typename T>
```

```
static inline detail::FloatingPointReturnType<T>::Type vtkm::Sqrt(const T &x)
```

Compute the square root of *x*. On some hardware it is faster to find the reciprocal square root, so `RSqrt` should be used if you actually plan to divide by the square root.

```
template<typename T, vtkm::IdComponent N>
```

```
static inline vtkm::Vec<typename detail::FloatingPointReturnType<T>::Type, N> vtkm::Sqrt(const vtkm::Vec<T, N> &x)
```

Compute the square root of *x*. On some hardware it is faster to find the reciprocal square root, so `RSqrt` should be used if you actually plan to divide by the square root.

## 25.1.4 Remainders and Quotient

```
static inline vtkm::Float32 vtkm::ModF(vtkm::Float32 x, vtkm::Float32 &integral)
```

Gets the integral and fractional parts of *x*. The return value is the fractional part and *integral* is set to the integral part.

```
static inline vtkm::Float64 vtkm::ModF(vtkm::Float64 x, vtkm::Float64 &integral)
```

Gets the integral and fractional parts of *x*. The return value is the fractional part and *integral* is set to the integral part.

```
static inline vtkm::Float32 vtkm::Remainder(vtkm::Float32 x, vtkm::Float32 y)
```

Computes the remainder on division of 2 floating point numbers. The return value is *numerator* - *n denominator*, where *n* is the quotient of *numerator* divided by *denominator* rounded towards the nearest integer (instead of toward zero like `FMod`). For example, `FMod(6.5, 2.3)` returns -0.4, which is  $6.5 - 3 \cdot 2.3$ .

```
static inline vtkm::Float64 vtkm::Remainder(vtkm::Float64 x, vtkm::Float64 y)
```

Computes the remainder on division of 2 floating point numbers. The return value is *numerator* - *n denominator*, where *n* is the quotient of *numerator* divided by *denominator* rounded towards the nearest integer (instead of toward zero like `FMod`). For example, `FMod(6.5, 2.3)` returns -0.4, which is  $6.5 - 3 \cdot 2.3$ .

```
template<typename QType>
```

```
static inline vtkm::Float32 vtkm::RemainderQuotient(vtkm::Float32 numerator, vtkm::Float32 denominator, QType &quotient)
```

Returns the remainder on division of 2 floating point numbers just like `Remainder`. In addition, this function also returns the *quotient* used to get that remainder.

```
template<typename QType>
```

```
static inline vtkm::Float64 vtkm::RemainderQuotient(vtkm::Float64 numerator, vtkm::Float64 denominator, QType &quotient)
```

Returns the remainder on division of 2 floating point numbers just like `Remainder`. In addition, this function also returns the *quotient* used to get that remainder.

## 25.1.5 Rounding and Precision

```
inline vtkm::Float32 vtkm::Ceil(vtkm::Float32 x)
```

Round *x* to the smallest integer value not less than *x*.

```
inline vtkm::Float64 vtkm::Ceil(vtkm::Float64 x)
```

Round *x* to the smallest integer value not less than *x*.

```
template<typename T>
```

```
static inline detail::FloatingPointReturnType<T>::Type vtkm::Ceil(const T &x)
```

Round  $x$  to the smallest integer value not less than  $x$ .

```
template<typename T, vtkm::IdComponent N>
```

```
static inline vtkm::Vec<typename detail::FloatingPointReturnType<T>::Type, N> vtkm::Ceil(const vtkm::Vec<T, N> &x)
```

Round  $x$  to the smallest integer value not less than  $x$ .

```
static inline vtkm::Float32 vtkm::CopySign(vtkm::Float32 x, vtkm::Float32 y)
```

Copies the sign of  $y$  onto  $x$ . If  $y$  is positive, returns  $\text{Abs}(x)$ . If  $y$  is negative, returns  $-\text{Abs}(x)$ .

```
static inline vtkm::Float64 vtkm::CopySign(vtkm::Float64 x, vtkm::Float64 y)
```

Copies the sign of  $y$  onto  $x$ . If  $y$  is positive, returns  $\text{Abs}(x)$ . If  $y$  is negative, returns  $-\text{Abs}(x)$ .

```
template<typename T, vtkm::IdComponent N>
```

```
static inline vtkm::Vec<T, N> vtkm::CopySign(const vtkm::Vec<T, N> &x, const vtkm::Vec<T, N> &y)
```

Copies the sign of  $y$  onto  $x$ . If  $y$  is positive, returns  $\text{Abs}(x)$ . If  $y$  is negative, returns  $-\text{Abs}(x)$ .

```
template<typename T>
```

```
static inline T vtkm::Epsilon()
```

Returns the difference between 1 and the least value greater than 1 that is representable by a floating point number. Epsilon is useful for specifying the tolerance one should have when considering numerical error. The `Epsilon()` function is templated to specify either a 32 or 64 bit floating point number. The convenience functions `Epsilon32()` and `Epsilon64()` are non-templated versions that return the precision for a particular precision.

```
static inline vtkm::Float32 vtkm::Epsilon32()
```

Returns the difference between 1 and the least value greater than 1 that is representable by a floating point number. Epsilon is useful for specifying the tolerance one should have when considering numerical error. The `Epsilon()` function is templated to specify either a 32 or 64 bit floating point number. The convenience functions `Epsilon32()` and `Epsilon64()` are non-templated versions that return the precision for a particular precision.

```
static inline vtkm::Float64 vtkm::Epsilon64()
```

Returns the difference between 1 and the least value greater than 1 that is representable by a floating point number. Epsilon is useful for specifying the tolerance one should have when considering numerical error. The `Epsilon()` function is templated to specify either a 32 or 64 bit floating point number. The convenience functions `Epsilon32()` and `Epsilon64()` are non-templated versions that return the precision for a particular precision.

```
static inline vtkm::Float32 vtkm::FMod(vtkm::Float32 x, vtkm::Float32 y)
```

Computes the remainder on division of 2 floating point numbers. The return value is  $\text{numerator} - n \text{ denominator}$ , where  $n$  is the quotient of  $\text{numerator}$  divided by  $\text{denominator}$  rounded towards zero to an integer. For example, `FMod(6.5, 2.3)` returns 1.9, which is  $6.5 - 2 \cdot 2.3$ .

```
static inline vtkm::Float64 vtkm::FMod(vtkm::Float64 x, vtkm::Float64 y)
```

Computes the remainder on division of 2 floating point numbers. The return value is  $\text{numerator} - n \text{ denominator}$ , where  $n$  is the quotient of  $\text{numerator}$  divided by  $\text{denominator}$  rounded towards zero to an integer. For example, `FMod(6.5, 2.3)` returns 1.9, which is  $6.5 - 2 \cdot 2.3$ .

```
inline vtkm::Float32 vtkm::Round(vtkm::Float32 x)
```

Round  $x$  to the nearest integral value.

```
inline vtkm::Float64 vtkm::Round(vtkm::Float64 x)
```

Round  $x$  to the nearest integral value.

```
template<typename T>
```

static inline detail::FloatingPointReturnType<*T*>::Type vtkm::Round(const *T* &x)

Round *x* to the nearest integral value.

template<typename *T*, vtkm::IdComponent *N*>

static inline vtkm::Vec<typename detail::FloatingPointReturnType<*T*>::Type, *N*> vtkm::Round(const vtkm::Vec<*T*, *N*> &x)

Round *x* to the nearest integral value.

### 25.1.6 Sign

static inline vtkm::Int32 vtkm::Abs(vtkm::Int32 x)

Return the absolute value of *x*. That is, return *x* if it is positive or  $-x$  if it is negative.

static inline vtkm::Int64 vtkm::Abs(vtkm::Int64 x)

Return the absolute value of *x*. That is, return *x* if it is positive or  $-x$  if it is negative.

static inline vtkm::Float32 vtkm::Abs(vtkm::Float32 x)

Return the absolute value of *x*. That is, return *x* if it is positive or  $-x$  if it is negative.

static inline vtkm::Float64 vtkm::Abs(vtkm::Float64 x)

Return the absolute value of *x*. That is, return *x* if it is positive or  $-x$  if it is negative.

template<typename *T*>

static inline detail::FloatingPointReturnType<*T*>::Type vtkm::Abs(*T* x)

Return the absolute value of *x*. That is, return *x* if it is positive or  $-x$  if it is negative.

template<typename *T*, vtkm::IdComponent *N*>

static inline vtkm::Vec<*T*, *N*> vtkm::Abs(const vtkm::Vec<*T*, *N*> &x)

Return the absolute value of *x*. That is, return *x* if it is positive or  $-x$  if it is negative.

inline vtkm::Float32 vtkm::Floor(vtkm::Float32 x)

Round *x* to the largest integer value not greater than *x*.

inline vtkm::Float64 vtkm::Floor(vtkm::Float64 x)

Round *x* to the largest integer value not greater than *x*.

template<typename *T*>

static inline detail::FloatingPointReturnType<*T*>::Type vtkm::Floor(const *T* &x)

Round *x* to the largest integer value not greater than *x*.

template<typename *T*, vtkm::IdComponent *N*>

static inline vtkm::Vec<typename detail::FloatingPointReturnType<*T*>::Type, *N*> vtkm::Floor(const vtkm::Vec<*T*, *N*> &x)

Round *x* to the largest integer value not greater than *x*.

static inline vtkm::Int32 vtkm::SignBit(vtkm::Float32 x)

Returns a nonzero value if *x* is negative.

static inline vtkm::Int32 vtkm::SignBit(vtkm::Float64 x)

Returns a nonzero value if *x* is negative.

### 25.1.7 Trigonometry

inline vtkm::Float32 vtkm::ACos(vtkm::Float32 x)

Compute the arc cosine of  $x$ .

inline vtkm::Float64 vtkm::ACos(vtkm::Float64 x)

Compute the arc cosine of  $x$ .

template<typename T>

static inline detail::FloatingPointReturnType<T>::Type vtkm::ACos(const T &x)

Compute the arc cosine of  $x$ .

template<typename T, vtkm::IdComponent N>

static inline vtkm::Vec<typename detail::FloatingPointReturnType<T>::Type, N> vtkm::ACos(const vtkm::Vec<T, N> &x)

Compute the arc cosine of  $x$ .

inline vtkm::Float32 vtkm::ACosH(vtkm::Float32 x)

Compute the hyperbolic arc cosine of  $x$ .

inline vtkm::Float64 vtkm::ACosH(vtkm::Float64 x)

Compute the hyperbolic arc cosine of  $x$ .

template<typename T>

static inline detail::FloatingPointReturnType<T>::Type vtkm::ACosH(const T &x)

Compute the hyperbolic arc cosine of  $x$ .

template<typename T, vtkm::IdComponent N>

static inline vtkm::Vec<typename detail::FloatingPointReturnType<T>::Type, N> vtkm::ACosH(const vtkm::Vec<T, N> &x)

Compute the hyperbolic arc cosine of  $x$ .

inline vtkm::Float32 vtkm::ASin(vtkm::Float32 x)

Compute the arc sine of  $x$ .

inline vtkm::Float64 vtkm::ASin(vtkm::Float64 x)

Compute the arc sine of  $x$ .

template<typename T>

static inline detail::FloatingPointReturnType<T>::Type vtkm::ASin(const T &x)

Compute the arc sine of  $x$ .

template<typename T, vtkm::IdComponent N>

static inline vtkm::Vec<typename detail::FloatingPointReturnType<T>::Type, N> vtkm::ASin(const vtkm::Vec<T, N> &x)

Compute the arc sine of  $x$ .

inline vtkm::Float32 vtkm::ASinH(vtkm::Float32 x)

Compute the hyperbolic arc sine of  $x$ .

inline vtkm::Float64 vtkm::ASinH(vtkm::Float64 x)

Compute the hyperbolic arc sine of  $x$ .

template<typename T>

static inline detail::FloatingPointReturnType<T>::Type vtkm::ASinH(const T &x)

Compute the hyperbolic arc sine of  $x$ .

template<typename T, vtkm::IdComponent N>

```
static inline vtkm::Vec<typename detail::FloatingPointReturnType<T>::Type, N> vtkm::ASinh(const vtkm::Vec<T, N> &x)
```

Compute the hyperbolic arc sine of  $\mathbf{x}$ .

```
inline vtkm::Float32 vtkm::ATan(vtkm::Float32 x)
```

Compute the arc tangent of  $\mathbf{x}$ .

```
inline vtkm::Float64 vtkm::ATan(vtkm::Float64 x)
```

Compute the arc tangent of  $\mathbf{x}$ .

```
template<typename T>
```

```
static inline detail::FloatingPointReturnType<T>::Type vtkm::ATan(const T &x)
```

Compute the arc tangent of  $\mathbf{x}$ .

```
template<typename T, vtkm::IdComponent N>
```

```
static inline vtkm::Vec<typename detail::FloatingPointReturnType<T>::Type, N> vtkm::ATan(const vtkm::Vec<T, N> &x)
```

Compute the arc tangent of  $\mathbf{x}$ .

```
static inline vtkm::Float32 vtkm::ATan2(vtkm::Float32 x, vtkm::Float32 y)
```

Compute the arc tangent of  $\mathbf{x} / \mathbf{y}$  using the signs of both arguments to determine the quadrant of the return value.

```
static inline vtkm::Float64 vtkm::ATan2(vtkm::Float64 x, vtkm::Float64 y)
```

Compute the arc tangent of  $\mathbf{x} / \mathbf{y}$  using the signs of both arguments to determine the quadrant of the return value.

```
inline vtkm::Float32 vtkm::ATanh(vtkm::Float32 x)
```

Compute the hyperbolic arc tangent of  $\mathbf{x}$ .

```
inline vtkm::Float64 vtkm::ATanh(vtkm::Float64 x)
```

Compute the hyperbolic arc tangent of  $\mathbf{x}$ .

```
template<typename T>
```

```
static inline detail::FloatingPointReturnType<T>::Type vtkm::ATanh(const T &x)
```

Compute the hyperbolic arc tangent of  $\mathbf{x}$ .

```
template<typename T, vtkm::IdComponent N>
```

```
static inline vtkm::Vec<typename detail::FloatingPointReturnType<T>::Type, N> vtkm::ATanh(const vtkm::Vec<T, N> &x)
```

Compute the hyperbolic arc tangent of  $\mathbf{x}$ .

```
inline vtkm::Float32 vtkm::Cos(vtkm::Float32 x)
```

Compute the cosine of  $\mathbf{x}$ .

```
inline vtkm::Float64 vtkm::Cos(vtkm::Float64 x)
```

Compute the cosine of  $\mathbf{x}$ .

```
template<typename T>
```

```
static inline detail::FloatingPointReturnType<T>::Type vtkm::Cos(const T &x)
```

Compute the cosine of  $\mathbf{x}$ .

```
template<typename T, vtkm::IdComponent N>
```

```
static inline vtkm::Vec<typename detail::FloatingPointReturnType<T>::Type, N> vtkm::Cos(const vtkm::Vec<T, N> &x)
```

Compute the cosine of  $\mathbf{x}$ .

```
inline vtkm::Float32 vtkm::Cosh(vtkm::Float32 x)
```

Compute the hyperbolic cosine of  $\mathbf{x}$ .



```
inline vtkm::Float64 vtkm::CosH(vtkm::Float64 x)
```

Compute the hyperbolic cosine of  $x$ .

```
template<typename T>
```

```
static inline detail::FloatingPointReturnType<T>::Type vtkm::CosH(const T &x)
```

Compute the hyperbolic cosine of  $x$ .

```
template<typename T, vtkm::IdComponent N>
```

```
static inline vtkm::Vec<typename detail::FloatingPointReturnType<T>::Type, N> vtkm::CosH(const vtkm::Vec<T, N> &x)
```

Compute the hyperbolic cosine of  $x$ .

```
template<typename T = vtkm::Float64>
```

```
static inline constexpr detail::FloatingPointReturnType<T>::Type vtkm::Pi()
```

Returns the constant Pi.

```
template<typename T = vtkm::Float64>
```

```
static inline constexpr detail::FloatingPointReturnType<T>::Type vtkm::Pi_2()
```

Returns the constant Pi halves.

```
template<typename T = vtkm::Float64>
```

```
static inline constexpr detail::FloatingPointReturnType<T>::Type vtkm::Pi_3()
```

Returns the constant Pi thirds.

```
template<typename T = vtkm::Float64>
```

```
static inline constexpr detail::FloatingPointReturnType<T>::Type vtkm::Pi_4()
```

Returns the constant Pi fourths.

```
template<typename T = vtkm::Float64>
```

```
static inline constexpr detail::FloatingPointReturnType<T>::Type vtkm::Pi_180()
```

Returns the constant Pi one hundred and eightieth.

```
inline vtkm::Float32 vtkm::Sin(vtkm::Float32 x)
```

Compute the sine of  $x$ .

```
inline vtkm::Float64 vtkm::Sin(vtkm::Float64 x)
```

Compute the sine of  $x$ .

```
template<typename T>
```

```
static inline detail::FloatingPointReturnType<T>::Type vtkm::Sin(const T &x)
```

Compute the sine of  $x$ .

```
template<typename T, vtkm::IdComponent N>
```

```
static inline vtkm::Vec<typename detail::FloatingPointReturnType<T>::Type, N> vtkm::Sin(const vtkm::Vec<T, N> &x)
```

Compute the sine of  $x$ .

```
inline vtkm::Float32 vtkm::SinH(vtkm::Float32 x)
```

Compute the hyperbolic sine of  $x$ .

```
inline vtkm::Float64 vtkm::SinH(vtkm::Float64 x)
```

Compute the hyperbolic sine of  $x$ .

```
template<typename T>
```

```
static inline detail::FloatingPointReturnType<T>::Type vtkm::SinH(const T &x)
```

Compute the hyperbolic sine of  $x$ .

```
template<typename T, vtkm::IdComponent N>
```



```
static inline vtkm::Vec<typename detail::FloatingPointReturnType<T>::Type, N> vtkm::SinH(const vtkm::Vec<T, N> &x)
```

Compute the hyperbolic sine of  $x$ .

```
inline vtkm::Float32 vtkm::Tan(vtkm::Float32 x)
```

Compute the tangent of  $x$ .

```
inline vtkm::Float64 vtkm::Tan(vtkm::Float64 x)
```

Compute the tangent of  $x$ .

```
template<typename T>
```

```
static inline detail::FloatingPointReturnType<T>::Type vtkm::Tan(const T &x)
```

Compute the tangent of  $x$ .

```
template<typename T, vtkm::IdComponent N>
```

```
static inline vtkm::Vec<typename detail::FloatingPointReturnType<T>::Type, N> vtkm::Tan(const vtkm::Vec<T, N> &x)
```

Compute the tangent of  $x$ .

```
inline vtkm::Float32 vtkm::TanH(vtkm::Float32 x)
```

Compute the hyperbolic tangent of  $x$ .

```
inline vtkm::Float64 vtkm::TanH(vtkm::Float64 x)
```

Compute the hyperbolic tangent of  $x$ .

```
template<typename T>
```

```
static inline detail::FloatingPointReturnType<T>::Type vtkm::TanH(const T &x)
```

Compute the hyperbolic tangent of  $x$ .

```
template<typename T, vtkm::IdComponent N>
```

```
static inline vtkm::Vec<typename detail::FloatingPointReturnType<T>::Type, N> vtkm::TanH(const vtkm::Vec<T, N> &x)
```

Compute the hyperbolic tangent of  $x$ .

```
template<typename T = vtkm::Float64>
```

```
static inline constexpr detail::FloatingPointReturnType<T>::Type vtkm::TwoPi()
```

Returns the constant 2 times Pi.

## 25.1.8 Miscellaneous

```
inline vtkm::UInt64 vtkm::FloatDistance(vtkm::Float64 x, vtkm::Float64 y)
```

Computes the number of representables between two floating point numbers.

This function is non-negative and symmetric in its arguments. If either argument is non-finite, the value returned is the maximum value allowed by 64-bit unsigned integers:  $2^{64}-1$ .

```
inline vtkm::UInt64 vtkm::FloatDistance(vtkm::Float32 x, vtkm::Float32 y)
```

Computes the number of representables between two floating point numbers.

This function is non-negative and symmetric in its arguments. If either argument is non-finite, the value returned is the maximum value allowed by 64-bit unsigned integers:  $2^{64}-1$ .

```
template<typename T>
```

```
static inline T vtkm::Max(const T &x, const T &y)
```

Returns  $x$  or  $y$ , whichever is larger.

Returns  $x$  or  $y$ , whichever is larger.

```
template<typename T>
static inline T vtkm::Min(const T &x, const T &y)
```

Returns  $x$  or  $y$ , whichever is smaller.

Returns  $x$  or  $y$ , whichever is smaller.

## 25.2 Vector Analysis

Visualization and computational geometry algorithms often perform vector analysis operations. The `vtkm/VectorAnalysis.h` header file provides functions that perform the basic common vector analysis operations.

```
template<typename T>
vtkm::Vec<typename detail::FloatingPointReturnType<T>::Type, 3> vtkm::Cross(const vtkm::Vec<T, 3> &x,
                                                                              const vtkm::Vec<T, 3> &y)
```

Find the cross product of two vectors.

If VTK-m is compiled with FMA support, it uses Kahan's difference of products algorithm to achieve a maximum error of 1.5 ulps in each component.

```
template<typename ValueType, typename WeightType>
inline ValueType vtkm::Lerp(const ValueType &value0, const ValueType &value1, const WeightType &weight)
```

Returns the linear interpolation of two values based on weight.

`Lerp` returns the linear interpolation of two values based on a weight. If `weight` is outside  $[0,1]$  then `Lerp` extrapolates. If `weight=0` then `value0` is returned. If `weight=1` then `value1` is returned.

```
template<typename T>
detail::FloatingPointReturnType<T>::Type vtkm::Magnitude(const T &x)
```

Returns the magnitude of a vector.

It is usually much faster to compute `MagnitudeSquared`, so that should be substituted when possible (unless you are just going to take the square root, which would be besides the point). On some hardware it is also faster to find the reciprocal magnitude, so `RMagnitude` should be used if you actually plan to divide by the magnitude.

```
template<typename T>
detail::FloatingPointReturnType<T>::Type vtkm::MagnitudeSquared(const T &x)
```

Returns the square of the magnitude of a vector.

It is usually much faster to compute the square of the magnitude than the magnitude, so you should use this function in place of `Magnitude` or `RMagnitude` when possible.

```
template<typename T>
T vtkm::Normal(const T &x)
```

Returns a normalized version of the given vector.

The resulting vector points in the same direction but has unit length.

```
template<typename T>
void vtkm::Normalize(T &x)
```

Changes a vector to be normal.

The given vector is scaled to be unit length.

```
template<typename T, int N>
int vtkm::Orthonormalize(const vtkm::Vec<vtkm::Vec<T, N>, N> &inputs, vtkm::Vec<vtkm::Vec<T, N>, N>
                        &outputs, T tol = static_cast<T>(1e-6))
```

Convert a set of vectors to an orthonormal basis.

This function performs Gram-Schmidt orthonormalization for 3-D vectors. The first output vector will always be parallel to the first input vector. The remaining output vectors will be orthogonal and unit length and have the same handedness as their corresponding input vectors.

This method is geometric. It does not require a matrix solver. However, unlike the algebraic eigensolver techniques which do use matrix inversion, this method may return zero-length output vectors if some input vectors are collinear. The number of non-zero (to within the specified tolerance, `tol`) output vectors is the return value.

See [https://en.wikipedia.org/wiki/Gram%E2%80%93Schmidt\\_process](https://en.wikipedia.org/wiki/Gram%E2%80%93Schmidt_process) for details.

```
template<typename T, int N>
vtkm::Vec<T, N> vtkm::Project(const vtkm::Vec<T, N> &v, const vtkm::Vec<T, N> &u)
```

Project a vector onto another vector.

This method computes the orthogonal projection of the vector `v` onto `u`; that is, it projects its first argument onto its second.

Note that if the vector `u` has zero length, the output vector will have all its entries equal to NaN.

```
template<typename T, int N>
T vtkm::ProjectedDistance(const vtkm::Vec<T, N> &v, const vtkm::Vec<T, N> &u)
```

Project a vector onto another vector, returning only the projected distance.

This method computes the orthogonal projection of the vector `v` onto `u`; that is, it projects its first argument onto its second.

Note that if the vector `u` has zero length, the output will be NaN.

```
template<typename T>
detail::FloatingPointReturnType<T>::Type vtkm::RMagnitude(const T &x)
```

Returns the reciprocal magnitude of a vector.

On some hardware `RMagnitude` is faster than `Magnitude`, but neither is as fast as `MagnitudeSquared`. This function works on scalars as well as vectors, in which case it just returns the reciprocal of the scalar.

```
template<typename T>
vtkm::Vec<typename detail::FloatingPointReturnType<T>::Type, 3> vtkm::TriangleNormal(const vtkm::Vec<T,
                                                                                          3> &a, const
                                                                                          vtkm::Vec<T, 3>
                                                                                          &b, const
                                                                                          vtkm::Vec<T, 3>
                                                                                          &c)
```

Find the normal of a triangle.

Given three coordinates in space, which, unless degenerate, uniquely define a triangle and the plane the triangle is on, returns a vector perpendicular to that triangle/plane.

Note that the returned vector might not be a unit vector. In fact, the length is equal to twice the area of the triangle. If you want a unit vector, send the result through the `vtkm::Normal()` or `vtkm::Normalize()` function.

## 25.3 Matrices

Linear algebra operations on small matrices that are done on a single thread are located in `vtkm/Matrix.h`.

This header defines the `vtkm::Matrix` templated class. The template parameters are first the type of component, then the number of rows, then the number of columns. The overloaded parentheses operator can be used to retrieve values based on row and column indices. Likewise, the bracket operators can be used to reference the `vtkm::Matrix` as a 2D array (indexed by row first).

```
template<typename T, vtkm::IdComponent NumRow, vtkm::IdComponent NumCol>
```

```
class Matrix
```

Basic *Matrix* type.

The *Matrix* class holds a small two dimensional array for simple linear algebra and vector operations. VTK-m provides several Matrix-based operations to assist in visualization computations.

A *Matrix* is not intended to hold very large arrays. Rather, they are a per-thread data structure to hold information like geometric transforms and tensors.

### Public Functions

```
inline Matrix()
```

Creates an uninitialized matrix. The values in the matrix are not determined.

```
inline explicit Matrix(const ComponentType &value)
```

Creates a matrix initialized with all values set to the provided value.

```
inline const vtkm::Vec<ComponentType, NUM_COLUMNS> &operator[] (vtkm::IdComponent rowIndex)  
const
```

Brackets are used to reference a matrix like a 2D array (i.e.

`matrix[row][column]`).

```
inline vtkm::Vec<ComponentType, NUM_COLUMNS> &operator[] (vtkm::IdComponent rowIndex)
```

Brackets are used to referens a matrix like a 2D array i.e.

`matrix[row][column]`.

```
inline const ComponentType &operator() (vtkm::IdComponent rowIndex, vtkm::IdComponent colIndex)  
const
```

Parentheses are used to reference a matrix using mathematical tuple notation i.e.

`matrix(row,column)`.

```
inline ComponentType &operator() (vtkm::IdComponent rowIndex, vtkm::IdComponent colIndex)
```

Parentheses are used to reference a matrix using mathematical tuple notation i.e.

`matrix(row,column)`.

The following example builds a `vtkm::Matrix` that contains the values

$$\begin{vmatrix} 0 & 1 & 2 \\ 10 & 11 & 12 \end{vmatrix}$$

Example 1: Creating a `vtkm::Matrix`.

```

1  vtkm::Matrix<vtkm::Float32, 2, 3> matrix;
2
3  // Using parenthesis notation.
4  matrix(0, 0) = 0.0f;
5  matrix(0, 1) = 1.0f;
6  matrix(0, 2) = 2.0f;
7
8  // Using bracket notation.
9  matrix[1][0] = 10.0f;
10 matrix[1][1] = 11.0f;
11 matrix[1][2] = 12.0f;

```

The `vtkm/Matrix.h` header also defines the following functions that operate on matrices.

template<typename **T**, vtkm::IdComponent **Size**>

**T** vtkm::MatrixDeterminant(const vtkm::Matrix<**T**, **Size**, **Size**> &A)

Compute the determinant of a matrix.

template<typename **T**, vtkm::IdComponent **NumRow**, vtkm::IdComponent **NumCol**>

vtkm::Vec<**T**, **NumRow**> vtkm::MatrixGetColumn(const vtkm::Matrix<**T**, **NumRow**, **NumCol**> &matrix,  
vtkm::IdComponent columnIndex)

Returns a tuple containing the given column (indexed from 0) of the given matrix.

Might not be as efficient as the `MatrixGetRow()` function.

template<typename **T**, vtkm::IdComponent **NumRow**, vtkm::IdComponent **NumCol**>

const vtkm::Vec<**T**, **NumCol**> &vtkm::MatrixGetRow(const vtkm::Matrix<**T**, **NumRow**, **NumCol**> &matrix,  
vtkm::IdComponent rowIndex)

Returns a tuple containing the given row (indexed from 0) of the given matrix.

template<typename **T**, vtkm::IdComponent **Size**>

vtkm::Matrix<**T**, **Size**, **Size**> vtkm::MatrixIdentity()

Returns the identity matrix.

template<typename **T**, vtkm::IdComponent **Size**>

void vtkm::MatrixIdentity(vtkm::Matrix<**T**, **Size**, **Size**> &matrix)

Fills the given matrix with the identity matrix.

template<typename **T**, vtkm::IdComponent **Size**>

vtkm::Matrix<**T**, **Size**, **Size**> vtkm::MatrixInverse(const vtkm::Matrix<**T**, **Size**, **Size**> &A, bool &valid)

Find and return the inverse of the given matrix.

If the matrix is singular, the inverse will not be correct and valid will be set to false.

template<typename **T**, vtkm::IdComponent **NumRow**, vtkm::IdComponent **NumCol**, vtkm::IdComponent  
**NumInternal**>

vtkm::Matrix<**T**, **NumRow**, **NumCol**> vtkm::MatrixMultiply(const vtkm::Matrix<**T**, **NumRow**, **NumInternal**>  
&leftFactor, const vtkm::Matrix<**T**, **NumInternal**,  
**NumCol**> &rightFactor)

Standard matrix multiplication.

template<typename **T**, vtkm::IdComponent **NumRow**, vtkm::IdComponent **NumCol**>

vtkm::Vec<**T**, **NumRow**> vtkm::MatrixMultiply(const vtkm::Matrix<**T**, **NumRow**, **NumCol**> &leftFactor, const  
vtkm::Vec<**T**, **NumCol**> &rightFactor)

Standard matrix-vector multiplication.

```
template<typename T, vtkm::IdComponent NumRow, vtkm::IdComponent NumCol>
vtkm::Vec<T, NumCol> vtkm::MatrixMultiply(const vtkm::Vec<T, NumRow> &leftFactor, const
                                         vtkm::Matrix<T, NumRow, NumCol> &rightFactor)
```

Standard vector-matrix multiplication.

```
template<typename T, vtkm::IdComponent NumRow, vtkm::IdComponent NumCol>
void vtkm::MatrixSetColumn(vtkm::Matrix<T, NumRow, NumCol> &matrix, vtkm::IdComponent columnIndex,
                          const vtkm::Vec<T, NumRow> &columnValues)
```

Convenience function for setting a column of a matrix.

```
template<typename T, vtkm::IdComponent NumRow, vtkm::IdComponent NumCol>
void vtkm::MatrixSetRow(vtkm::Matrix<T, NumRow, NumCol> &matrix, vtkm::IdComponent rowIndex, const
                       vtkm::Vec<T, NumCol> &rowValues)
```

Convenience function for setting a row of a matrix.

```
template<typename T, vtkm::IdComponent NumRows, vtkm::IdComponent NumCols>
vtkm::Matrix<T, NumCols, NumRows> vtkm::MatrixTranspose(const vtkm::Matrix<T, NumRows, NumCols>
                                                         &matrix)
```

Returns the transpose of the given matrix.

```
template<typename T, vtkm::IdComponent Size>
vtkm::Vec<T, Size> vtkm::SolveLinearSystem(const vtkm::Matrix<T, Size, Size> &A, const vtkm::Vec<T,
                                                    Size> &b, bool &valid)
```

Solve the linear system  $Ax = b$  for  $x$ .

If a single solution is found, `valid` is set to true, false otherwise.

## 25.4 Newton's Method

VTK-m's matrix methods (documented in [Section 25.3 \(Matrices\)](#)) provide a method to solve a small linear system of equations. However, sometimes it is necessary to solve a small nonlinear system of equations. This can be done with the `vtkm::NewtonsMethod()` function defined in the `vtkm/NewtonMethod.h` header.

The `vtkm::NewtonsMethod()` function assumes that the number of variables equals the number of equations. Newton's method operates on an iterative evaluate and search. Evaluations are performed using the functors passed into the `vtkm::NewtonsMethod()`.

```
template<typename ScalarType, vtkm::IdComponent Size, typename JacobianFunctor, typename
FunctionFunctor>
NewtonMethodResult<ScalarType, Size> vtkm::NewtonsMethod(JacobianFunctor jacobianEvaluator,
                                                         FunctionFunctor functionEvaluator,
                                                         vtkm::Vec<ScalarType, Size>
desiredFunctionOutput, vtkm::Vec<ScalarType,
Size> initialGuess = vtkm::Vec<ScalarType,
Size>(ScalarType(0)), ScalarType
convergeDifference = ScalarType(1e-3),
                                                         vtkm::IdComponent maxIterations = 10)
```

Uses Newton's method (a.k.a.

Newton-Raphson method) to solve a nonlinear system of equations. This function assumes that the number of variables equals the number of equations. Newton's method operates on an iterative evaluate and search. Evaluations are performed using the functors passed into the `NewtonMethod`. The first functor returns the  $N \times N$  matrix of the Jacobian at a given input point. The second functor returns the  $N$  tuple that is the function evaluation at the given input point. The input point that evaluates to the desired output, or the closest point found, is returned.

### Parameters

- **jacobianEvaluator** – [in] A functor whose operation takes a `vtkm::Vec` and returns a `vtkm::Matrix` containing the math function's Jacobian vector at that point.
- **functionEvaluator** – [in] A functor whose operation takes a `vtkm::Vec` and returns the evaluation of the math function at that point as another `vtkm::Vec`.
- **desiredFunctionOutput** – [in] The desired output of the function.
- **initialGuess** – [in] The initial guess to search from. If not specified, the origin is used.
- **convergeDifference** – [in] The convergence distance. If the iterative method changes all values less than this amount. Once all values change less, it considers the solution found. If not specified, set to 0.001.
- **maxIterations** – [in] The maximum amount of iterations to run before giving up and returning the best solution found. If not specified, set to 10.

### Returns

A `vtkm::NewtonsMethodResult` containing the best found result and state about its validity.

The `vtkm::NewtonsMethod()` function returns a `vtkm{NewtonsMethodResult}` object. `textidentifier{NewtonsMethodResult}` is a `textcode{struct}` templated on the type and number of input values of the nonlinear system. `textidentifier{NewtonsMethodResult}` contains the following items.

```
template<typename ScalarType, vtkm::IdComponent Size>
```

```
struct NewtonsMethodResult
```

An object returned from `NewtonsMethod()` that contains the result and other information about the final state.

### Public Members

```
bool Valid
```

True if Newton's method ran into a singularity.

```
bool Converged
```

True if Newton's method converted to below the convergence value.

```
vtkm::Vec<ScalarType, Size> Solution
```

The solution found by Newton's method.

If Converged is false, then this value is likely inaccurate. If Valid is false, then this value is undefined.

Example 2: Using `vtkm::NewtonsMethod()` to solve a small system of nonlinear equations.

```
1 // A functor for the mathematical function f(x) = [dot(x,x), x[0]*x[1]]
2 struct FunctionFunctor
3 {
4     template<typename T>
5     VTKM_EXEC_CONT vtkm::Vec<T, 2> operator()(const vtkm::Vec<T, 2>& x) const
6     {
7         return vtkm::make_Vec(vtkm::Dot(x, x), x[0] * x[1]);
8     }
9 };
```

(continues on next page)

(continued from previous page)

```

10
11 // A functor for the Jacobian of the mathematical function
12 //  $f(x) = [\text{dot}(x,x), x[0]*x[1]]$ , which is
13 //  $\begin{vmatrix} 2*x[0] & 2*x[1] \\ x[1] & x[0] \end{vmatrix}$ 
14 //  $\begin{vmatrix} x[1] & x[0] \end{vmatrix}$ 
15 struct JacobianFunctor
16 {
17     template<typename T>
18     VTKM_EXEC_CONT vtkm::Matrix<T, 2, 2> operator()(const vtkm::Vec<T, 2>& x) const
19     {
20         vtkm::Matrix<T, 2, 2> jacobian;
21         jacobian(0, 0) = 2 * x[0];
22         jacobian(0, 1) = 2 * x[1];
23         jacobian(1, 0) = x[1];
24         jacobian(1, 1) = x[0];
25
26         return jacobian;
27     }
28 };
29
30 VTKM_EXEC
31 void SolveNonlinear()
32 {
33     // Use Newton's method to solve the nonlinear system of equations:
34     //
35     //  $x^2 + y^2 = 2$ 
36     //  $x*y = 1$ 
37     //
38     // There are two possible solutions, which are  $(x=1,y=1)$  and  $(x=-1,y=-1)$ .
39     // The one found depends on the starting value.
40     vtkm::NewtonMethodResult<vtkm::Float32, 2> answer1 =
41         vtkm::NewtonMethod(JacobianFunctor(),
42                             FunctionFunctor(),
43                             vtkm::make_Vec(2.0f, 1.0f),
44                             vtkm::make_Vec(1.0f, 0.0f));
45     if (!answer1.Valid || !answer1.Converged)
46     {
47         // Failed to find solution
48     }
49     // answer1.Solution is [1,1]
50
51     vtkm::NewtonMethodResult<vtkm::Float32, 2> answer2 =
52         vtkm::NewtonMethod(JacobianFunctor(),
53                             FunctionFunctor(),
54                             vtkm::make_Vec(2.0f, 1.0f),
55                             vtkm::make_Vec(0.0f, -2.0f));
56     if (!answer2.Valid || !answer2.Converged)
57     {
58         // Failed to find solution
59     }
60     // answer2 is [-1,-1]
61 }

```



## WORKING WITH CELLS

In the control environment, data is defined in mesh structures that comprise a set of finite cells. (See [Section 7.2 \(Cell Sets\)](#) for information on defining cell sets in the control environment.) When worklets that operate on cells are scheduled, these grid structures are broken into their independent cells, and that data is handed to the worklet. Thus, cell-based operations in the execution environment exclusively operate on independent cells.

Unlike some other libraries such as VTK, VTK-m does not have a cell class that holds all the information pertaining to a cell of a particular type. Instead, VTK-m provides tags or identifiers defining the cell shape, and companion data like coordinate and field information are held in separate structures. This organization is designed so a worklet may specify exactly what information it needs, and only that information will be loaded.

### 26.1 Cell Shape Tags and Ids

Cell shapes can be specified with either a tag (defined with a struct with a name like `CellShapeTag*`) or an enumerated identifier (defined with a constant number with a name like `CELL_SHAPE_*`). These shape tags and identifiers are defined in `vtkm/CellShape.h` and declared in the `vtkm` namespace (because they can be used in either the control or the execution environment). [Figure 1](#) gives both the identifier and the tag names.

```
struct CellShapeTagVertex
```

```
enumerator CELL_SHAPE_VERTEX
```

Vertex cells of a single point.

```
struct CellShapeTagLine
```

```
enumerator CELL_SHAPE_LINE
```

A line cell connecting two points.

```
struct CellShapeTagPolyLine
```

```
enumerator CELL_SHAPE_POLY_LINE
```

A sequence of line segments.

A polyline has 2 or more points, and the points are connected in order by line segments forming a piecewise linear curve.

```
struct CellShapeTagTriangle
```

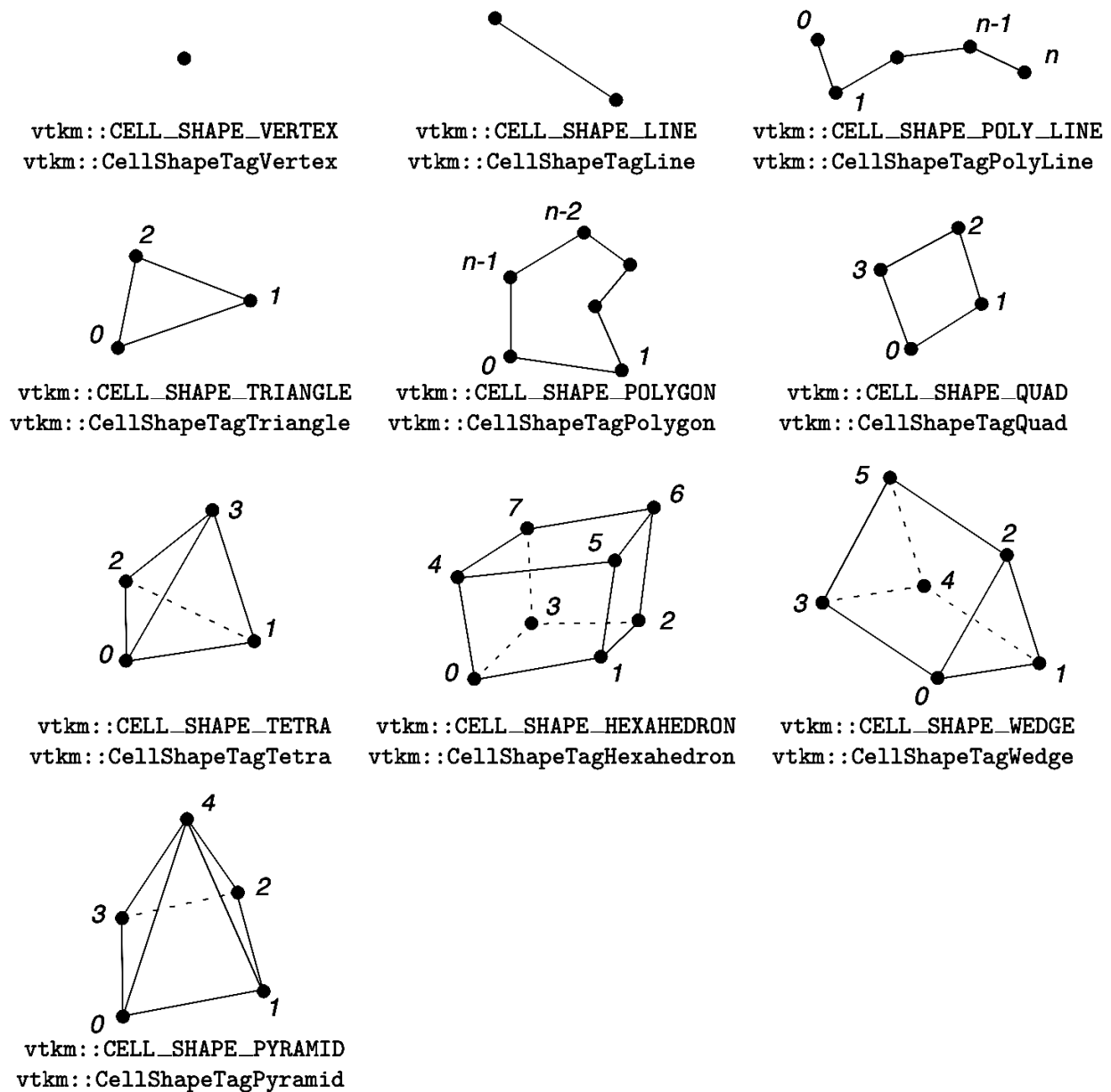


Figure 1: Basic Cell Shapes.

enumerator **CELL\_SHAPE\_TRIANGLE**

A triangle.

struct **CellShapeTagPolygon**

enumerator **CELL\_SHAPE\_POLYGON**

A general polygon shape.

All polygons have points ordered in counterclockwise order around the front side. Some operations may be invalid if the polygon is not a convex shape.

struct **CellShapeTagQuad**

enumerator **CELL\_SHAPE\_QUAD**

A four-sided polygon.

struct **CellShapeTagTetra**

enumerator **CELL\_SHAPE\_TETRA**

A tetrahedron.

A tetrahedron is a 3D polyhedron with 4 points and 4 triangular faces.

struct **CellShapeTagHexahedron**

enumerator **CELL\_SHAPE\_HEXAHEDRON**

A hexahedron.

struct **CellShapeTagWedge**

enumerator **CELL\_SHAPE\_WEDGE**

A wedge.

Wedges are simple prisms that can be formed by extruding a triangle. They have 2 triangular faces and 3 quadrilateral faces.

struct **CellShapeTagPyramid**

enumerator **CELL\_SHAPE\_PYRAMID**

A pyramid with a quadrilateral base and four triangular faces.0.

In addition to the basic cell shapes, there is a special “empty” cell with the identifier `vtkm::CELL_SHAPE_EMPTY` and tag `vtkm::CellShapeTagEmpty`. This type of cell has no points, edges, or faces and can be thought of as a placeholder for a null or void cell.

struct **CellShapeTagEmpty**

enumerator **CELL\_SHAPE\_EMPTY**

Placeholder for empty or invalid cells.

There is also a special cell shape “tag” named `vtkm::CellShapeTagGeneric` that is used when the actual cell shape is not known at compile time. `vtkm::CellShapeTagGeneric` actually has a member variable named `vtkm::CellShapeTagGeneric::Id` that stores the identifier for the cell shape. There is no equivalent identifier for a generic cell; cell shape identifiers can be placed in a `vtkm::IdComponent` at runtime.

struct **CellShapeTagGeneric**

A special cell shape tag that holds a cell shape that is not known at compile time.

Unlike other cell set tags, the Id field is set at runtime so its value cannot be used in template parameters. You need to use `vtkmGenericCellShapeMacro` to specialize on the cell type.

### Public Members

`vtkm::UInt8 Id`

An identifier that corresponds to one of the `CELL_SHAPE_*` identifiers.

This value is used to detect the proper shape at runtime.

When using cell shapes in templated classes and functions, you can use the `VTKM_IS_CELL_SHAPE_TAG` to ensure a type is a valid cell shape tag. This macro takes one argument and will produce a compile error if the argument is not a cell shape tag type.

**VTKM\_IS\_CELL\_SHAPE\_TAG**(tag)

Checks that the argument is a proper cell shape tag.

This is a handy concept check to make sure that a template argument is a proper cell shape tag.

## 26.1.1 Converting Between Tags and Identifiers

Every cell shape tag has a member variable named `Id` that contains the identifier for the cell shape. This provides a convenient mechanism for converting a cell shape tag to an identifier. Most cell shape tags have their `Id` member as a compile-time constant, but `vtkm::CellShapeTagGeneric::Id` is set at run time.

The `vtkm/CellShape.h` header also declares a templated class named `vtkm::CellShapeIdToTag` that converts a cell shape identifier to a cell shape tag. `vtkm::CellShapeIdToTag` has a single template argument that is the identifier. Inside the class is a type named `vtkm::CellShapeIdToTag::Tag` that is the type of the correct tag.

template<`vtkm::IdComponent Id`>

struct **CellShapeIdToTag**

A traits-like class to get an `CellShapeId` known at compile time to a tag.

Example 1: Using `vtkm::CellShapeIdToTag`.

```
1 void CellFunction(vtkm::CellShapeTagTriangle)
2 {
3     std::cout << "In CellFunction for triangles." << std::endl;
4 }
5
6 void DoSomethingWithACell()
```

(continues on next page)

(continued from previous page)

```

7 {
8   // Calls CellFunction overloaded with a vtkm::CellShapeTagTriangle.
9   CellFunction(vtkm::CellShapeIdToTag<vtkm::CELL_SHAPE_TRIANGLE>::Tag());
10 }

```

However, `vtkm::CellShapeIdToTag` is only viable if the identifier can be resolved at compile time. In the case where a cell identifier is stored in a variable or an array or the code is using a `vtkm::CellShapeTagGeneric`, the correct cell shape is not known until run time. In this case, the `vtkmGenericCellShapeMacro` macro can be used to check all possible conditions. This macro is embedded in a switch statement where the condition is the cell shape identifier.

#### `vtkmGenericCellShapeMacro`(call)

A macro used in a `switch` statement to determine cell shape.

The `vtkmGenericCellShapeMacro` is a series of case statements for all of the cell shapes supported by VTK-m. This macro is intended to be used inside of a `switch` statement on a cell type. For each cell shape condition, a `CellShapeTag` typedef is created and the given `call` is executed.

A typical use case of this class is to create the specialization of a function overloaded on a cell shape tag for the generic cell shape like as following.

```

template<typename WorkletType>
VTM_EXEC
void MyCellOperation(vtkm::CellShapeTagGeneric cellShape,
                    const vtkm::exec::FunctorBase &worklet)
{
    switch(cellShape.CellShapeId)
    {
        vtkmGenericCellShapeMacro(
            MyCellOperation(CellShapeTag())
        );
        default: worklet.RaiseError("Encountered unknown cell shape."); break
    }
}

```

Note that `vtkmGenericCellShapeMacro` does not have a default case. You should consider adding one that gives a

Often this method is used to implement the condition for a `vtkm::CellShapeTagGeneric` in a function overloaded for cell types. A demonstration of `vtkmGenericCellShapeMacro` is given in [Example 2](#).

## 26.1.2 Cell Traits

The `vtkm/CellTraits.h` header file contains a traits class named `vtkm::CellTraits` that provides information about a cell.

```
template<class CellTag>
```

```
struct CellTraits
```

Information about a cell based on its tag.

The templated `CellTraits` struct provides the basic high level information about cells (like the number of vertices in the cell or its dimensionality).

## Public Types

using **TopologicalDimensionsTag** =

`vtkm::CellTopologicalDimensionsTag<TOPOLOGICAL_DIMENSIONS>`

This tag is typedef'ed to `vtkm::CellTopologicalDimensionsTag<TOPOLOGICAL_DIMENSIONS>`.

This provides a convenient way to overload a function based on topological dimensions (which is usually more efficient than conditionals).

using **IsSizeFixed** = `vtkm::CellTraitsTagSizeFixed`

A tag specifying whether the number of points is fixed.

If set to `vtkm::CellTraitsTagSizeFixed`, then `NUM_POINTS` is set. If set to `vtkm::CellTraitsTagSizeVariable`, then the number of points is not known at compile time.

## Public Static Attributes

static const `vtkm::IdComponent` **TOPOLOGICAL\_DIMENSIONS** = 3

This defines the topological dimensions of the cell type.

3 for polyhedra, 2 for polygons, 1 for lines, 0 for points.

static constexpr `vtkm::IdComponent` **NUM\_POINTS** = 3

Number of points in the cell.

This is only defined for cell shapes of a fixed number of points (i.e., `IsSizeFixed` is set to `vtkm::CellTraitsTagSizeFixed`).

template<`vtkm::IdComponent` **dimension**>

struct **CellTopologicalDimensionsTag**

`vtkm::CellTraits::TopologyDimensionType` is typedef to this with the template parameter set to `TOPOLOGICAL_DIMENSIONS`.

See `vtkm::CellTraits` for more information.

struct **CellTraitsTagSizeFixed**

Tag for cell shapes with a fixed number of points.

struct **CellTraitsTagSizeVariable**

Tag for cell shapes that can have a variable number of points.

Example 2: Using `vtkm::CellTraits` to implement a polygon normal estimator.

```

1 namespace detail
2 {
3
4 template<typename PointCoordinatesVector, typename WorkletType>
5 VTKM_EXEC_CONT typename PointCoordinatesVector::ComponentType CellNormalImpl(
6     const PointCoordinatesVector& pointCoordinates,
7     vtkm::CellTopologicalDimensionsTag<2>,

```

(continues on next page)

(continued from previous page)

```

8   const WorkletType& worklet)
9   {
10  if (pointCoordinates.GetNumberOfComponents() >= 3)
11  {
12      return vtkm::TriangleNormal(
13          pointCoordinates[0], pointCoordinates[1], pointCoordinates[2]);
14  }
15  else
16  {
17      worklet.RaiseError("Degenerate polygon.");
18      return typename PointCoordinatesVector::ComponentType();
19  }
20  }
21
22  template<typename PointCoordinatesVector,
23          vtkm::IdComponent Dimensions,
24          typename WorkletType>
25  VTKM_EXEC_CONT typename PointCoordinatesVector::ComponentType CellNormalImpl(
26      const PointCoordinatesVector&,
27      vtkm::CellTopologicalDimensionsTag<Dimensions>,
28      const WorkletType& worklet)
29  {
30      worklet.RaiseError("Only polygons supported for cell normals.");
31      return typename PointCoordinatesVector::ComponentType();
32  }
33
34  } // namespace detail
35
36  template<typename CellShape, typename PointCoordinatesVector, typename WorkletType>
37  VTKM_EXEC_CONT typename PointCoordinatesVector::ComponentType CellNormal(
38      CellShape,
39      const PointCoordinatesVector& pointCoordinates,
40      const WorkletType& worklet)
41  {
42      return detail::CellNormalImpl(
43          pointCoordinates,
44          typename vtkm::CellTraits<CellShape>::TopologicalDimensionsTag(),
45          worklet);
46  }
47
48  template<typename PointCoordinatesVector, typename WorkletType>
49  VTKM_EXEC_CONT typename PointCoordinatesVector::ComponentType CellNormal(
50      vtkm::CellShapeTagGeneric shape,
51      const PointCoordinatesVector& pointCoordinates,
52      const WorkletType& worklet)
53  {
54      switch (shape.Id)
55      {
56          vtkmGenericCellShapeMacro(
57              return CellNormal(CellShapeTag(), pointCoordinates, worklet));
58          default:
59              worklet.RaiseError("Unknown cell type.");

```

(continues on next page)

(continued from previous page)

```

60     return typename PointCoordinatesVector::ComponentType();
61 }
62 }

```

## 26.2 Parametric and World Coordinates

Each cell type supports a one-to-one mapping between a set of parametric coordinates in the unit cube (or some subset of it) and the points in 3D space that are the locus contained in the cell. Parametric coordinates are useful because certain features of the cell, such as vertex location and center, are at a consistent location in parametric space irrespective of the location and distortion of the cell in world space. Also, many field operations are much easier with parametric coordinates.

The `vtkm/exec/ParametricCoordinates.h` header file contains the following functions for working with parametric coordinates. These functions contain several overloads for using different cell shape tags.

```

template<typename ParametricCoordType>
static inline vtkm::ErrorCode vtkm::exec::ParametricCoordinatesCenter(vtkm::IdComponent numPoints,
                                                                    vtkm::CellShapeTagGeneric
                                                                    shape,
                                                                    vtkm::Vec<ParametricCoordType,
                                                                    3> &pcoords)

```

Returns the parametric center of the given cell shape with the given number of points.

### Parameters

- **numPoints** – [in] The number of points in the cell.
- **shape** – [in] A tag of type `CellShapeTag*` to identify the shape of the cell. This method is overloaded for different shape types.
- **pcoords** – [out] `vtkm::Vec` to store the parametric center.

```

template<typename ParametricCoordType>
static inline vtkm::ErrorCode vtkm::exec::ParametricCoordinatesPoint(vtkm::IdComponent numPoints,
                                                                    vtkm::IdComponent pointIndex,
                                                                    vtkm::CellShapeTagGeneric shape,
                                                                    vtkm::Vec<ParametricCoordType,
                                                                    3> &pcoords)

```

Returns the parametric coordinate of a cell point of the given shape with the given number of points.

### Parameters

- **numPoints** – [in] The number of points in the cell.
- **pointIndex** – [in] The local index for the point to get the parametric coordinates of. This index is between 0 and  $n-1$  where  $n$  is the number of points in the cell.
- **shape** – [in] A tag of type `CellShapeTag*` to identify the shape of the cell. This method is overloaded for different shape types.
- **pcoords** – [out] `vtkm::Vec` to store the parametric center.

```

template<typename WorldCoordVector, typename PCoordType>

```



```
static inline vtkm::ErrorCode vtkm::exec::ParametricCoordinatesToWorldCoordinates(const WorldCoordVector
&pointWCoords,
const
vtkm::Vec<PCoordType,
3> &pcoords,
vtkm::CellShapeTagGeneric
shape, typename
WorldCoordVector::ComponentType
&result)
```

Converts parametric coordinates (coordinates relative to the cell) to world coordinates (coordinates in the global system).

#### Parameters

- **pointWCoords** – [in] A list of world coordinates for each point in the cell. This usually comes from a `FieldInPoint` argument in a `vtkm::worklet::WorkletVisitCellsWithPoints` where the coordinate system is passed into that argument.
- **pcoords** – [in] The parametric coordinates where you want to get world coordinates for.
- **shape** – [in] A tag of type `CellShapeTag*` to identify the shape of the cell. This method is overloaded for different shape types.
- **result** – [out] `vtkm::Vec` to store the interpolated world coordinates.

```
template<typename WorldCoordVector>
static inline vtkm::ErrorCode vtkm::exec::WorldCoordinatesToParametricCoordinates(const WorldCoordVector
&pointWCoords,
const typename
WorldCoordVector::ComponentType
&wcoords,
vtkm::CellShapeTagGeneric
shape, typename
WorldCoordVector::ComponentType
&result)
```

Converts world coordinates (coordinates in the global system) to parametric coordinates (coordinates relative to the cell).

This function can be slow for cell types with nonlinear interpolation (which is anything that is not a simplex).

#### Parameters

- **pointWCoords** – [in] A list of world coordinates for each point in the cell. This usually comes from a `FieldInPoint` argument in a `vtkm::worklet::WorkletVisitCellsWithPoints` where the coordinate system is passed into that argument.
- **wcoords** – [in] The world coordinates where you want to get parametric coordinates for.
- **shape** – [in] A tag of type `CellShapeTag*` to identify the shape of the cell. This method is overloaded for different shape types.
- **result** – [out] `vtkm::Vec` to store the associated parametric coordinates.

## 26.3 Interpolation

The shape of every cell is defined by the connections of some finite set of points. Field values defined on those points can be interpolated to any point within the cell to estimate a continuous field.

The `vtkm/exec/CellInterpolate.h` header contains the function `vtkm::exec::CellInterpolate()` to do this interpolation.

```
template<typename FieldVecType, typename ParametricCoordType>
vtkm::ErrorCode vtkm::exec::CellInterpolate(const FieldVecType &pointFieldValues, const
                                             vtkm::Vec<ParametricCoordType, 3> &parametricCoords,
                                             vtkm::CellShapeTagGeneric shape, typename
                                             FieldVecType::ComponentType &result)
```

Interpolate a point field in a cell.

Given the point field values for each node and the parametric coordinates of a point within the cell, interpolates the field to that point.

### Parameters

- **pointFieldValues** – [in] A list of field values for each point in the cell. This usually comes from a `FieldInPoint` argument in a `vtkm::worklet::WorkletVisitCellsWithPoints`.
- **parametricCoords** – [in] The parametric coordinates where you want to get the interpolated field value for.
- **shape** – [in] A tag of type `CellShapeTag*` to identify the shape of the cell. This method is overloaded for different shape types.
- **result** – [out] Value to store the interpolated field.

Example 3: Interpolating field values to a cell's center.

```
1 struct CellCenters : vtkm::worklet::WorkletVisitCellsWithPoints
2 {
3     using ControlSignature = void(CellSetIn,
4                                   FieldInPoint inputField,
5                                   FieldOutCell outputField);
6     using ExecutionSignature = void(CellShape, PointCount, _2, _3);
7     using InputDomain = _1;
8
9     template<typename CellShapeTag, typename FieldInVecType, typename FieldOutType>
10     VTKM_EXEC void operator()(CellShapeTag shape,
11                               vtkm::IdComponent pointCount,
12                               const FieldInVecType& inputField,
13                               FieldOutType& outputField) const
14     {
15         vtkm::Vec3f center;
16         vtkm::ErrorCode status =
17             vtkm::exec::ParametricCoordinatesCenter(pointCount, shape, center);
18         if (status != vtkm::ErrorCode::Success)
19         {
20             this->RaiseError(vtkm::ErrorString(status));
21             return;
22         }
23         vtkm::exec::CellInterpolate(inputField, center, shape, outputField);
```

(continues on next page)

(continued from previous page)

```

24 }
25 };

```

## 26.4 Derivatives

Since interpolations provide a continuous field function over a cell, it is reasonable to consider the derivative of this function. The `vtkm/exec/CellDerivative.h` header contains the function `vtkm::exec::CellDerivative()` to compute derivatives. The derivative is returned in a `vtkm::Vec` of size 3 corresponding to the partial derivatives in the  $x$ ,  $y$ , and  $z$  directions. This derivative is equivalent to the gradient of the field.

Example 4: Computing the derivative of the field at cell centers.

```

1 struct CellDerivatives : vtkm::worklet::WorkletVisitCellsWithPoints
2 {
3     using ControlSignature = void(CellSetIn,
4                                   FieldInPoint inputField,
5                                   FieldInPoint pointCoordinates,
6                                   FieldOutCell outputField);
7     using ExecutionSignature = void(CellShape, PointCount, _2, _3, _4);
8     using InputDomain = _1;
9
10    template<typename CellShapeTag,
11             typename FieldInVecType,
12             typename PointCoordVecType,
13             typename FieldOutType>
14    VTKM_EXEC void operator()(CellShapeTag shape,
15                             vtkm::IdComponent pointCount,
16                             const FieldInVecType& inputField,
17                             const PointCoordVecType& pointCoordinates,
18                             FieldOutType& outputField) const
19    {
20        vtkm::Vec3f center;
21        vtkm::ErrorCode status =
22            vtkm::exec::ParametricCoordinatesCenter(pointCount, shape, center);
23        if (status != vtkm::ErrorCode::Success)
24        {
25            this->RaiseError(vtkm::ErrorString(status));
26            return;
27        }
28        vtkm::exec::CellDerivative(inputField, pointCoordinates, center, shape, outputField);
29    }
30 };

```

## 26.5 Edges and Faces

As explained earlier in this chapter, a cell is defined by a collection of points and a shape identifier that describes how the points come together to form the structure of the cell. The cell shapes supported by VTK-m are documented in [Section 26.1 \(Cell Shape Tags and Ids\)](#). It contains [Figure 1](#), which shows how the points for each shape form the structure of the cell.

Most cell shapes can be broken into subelements. 2D and 3D cells have pairs of points that form *edges* at the boundaries of the cell. Likewise, 3D cells have loops of edges that form *faces* that encase the cell. [Figure 2](#) demonstrates the relationship of these constituent elements for some example cell shapes.



Figure 2: The constituent elements (points, edges, and faces) of cells..

The header file `vtkm/exec/CellEdge.h` contains a collection of functions to help identify the edges of a cell.

```
static inline vtkm::ErrorCode vtkm::exec::CellEdgeNumberOfEdges(vtkm::IdComponent numPoints,
                                                                vtkm::CellShapeTagGeneric shape,
                                                                vtkm::IdComponent &numEdges)
```

Get the number of edges in a cell.

### Parameters

- **numPoints** – [in] The number of points in the cell.
- **shape** – [in] A tag of type `CellShapeTag*` to identify the shape of the cell. This method is overloaded for different shape types.
- **numEdges** – [out] A reference to return the number of edges.

```
static inline vtkm::ErrorCode vtkm::exec::CellEdgeLocalIndex(vtkm::IdComponent numPoints,
                                                            vtkm::IdComponent pointIndex,
                                                            vtkm::IdComponent edgeIndex,
                                                            vtkm::CellShapeTagGeneric shape,
                                                            vtkm::IdComponent &result)
```

Given the index for an edge of a cell and one of the points on that edge, this function returns the point index for the cell.

To get the point indices relative to the data set, the returned index should be used to reference a `PointIndices` list.

### Parameters

- **numPoints** – [in] The number of points in the cell.
- **pointIndex** – [in] The index of the edge within the cell.
- **edgeIndex** – [in] The index of the point on the edge (either 0 or 1).
- **shape** – [in] A tag of type `CellShapeTag*` to identify the shape of the cell. This method is overloaded for different shape types.
- **result** – [out] Reference to put the index of the point relative to the cell (between 0 and the number of points in the cell).

```
template<typename CellShapeTag, typename GlobalPointIndicesVecType>
```

```
static inline vtkm::ErrorCode vtkm::exec::CellEdgeCanonicalId(vtkm::IdComponent numPoints,
                                                             vtkm::IdComponent edgeIndex,
                                                             CellShapeTag shape, const
                                                             GlobalPointIndicesVecType
                                                             &globalPointIndicesVec, vtkm::Id2 &result)
```

Returns a canonical identifier for a cell edge.

Given information about a cell edge and the global point indices for that cell, returns a *vtkm::Id2* that contains values that are unique to that edge. The values for two edges will be the same if and only if the edges contain the same points.

The following example demonstrates a pair of worklets that use the cell edge functions. As is typical for operations of this nature, one worklet counts the number of edges in each cell and another uses this count to generate the data.

### Did You Know?

**Example 5** demonstrates one of many techniques for creating cell sets in a worklet. Chapter~ref{chap:GeneratingCellSets} describes this and many more such techniques.

Example 5: Using cell edge functions.

```
1 struct EdgesCount : vtkm::worklet::WorkletVisitCellsWithPoints
2 {
3     using ControlSignature = void(CellSetIn, FieldOutCell numEdgesInCell);
4     using ExecutionSignature = void(CellShape, PointCount, _2);
5     using InputDomain = _1;
6
7     template<typename CellShapeTag>
8     VTKM_EXEC void operator()(CellShapeTag cellShape,
9                             vtkm::IdComponent numPointsInCell,
10                            vtkm::IdComponent& numEdges) const
11     {
12         vtkm::ErrorCode status =
13             vtkm::exec::CellEdgeNumberOfEdges(numPointsInCell, cellShape, numEdges);
14         if (status != vtkm::ErrorCode::Success)
15         {
16             this->RaiseError(vtkm::ErrorString(status));
17         }
18     }
19 };
20
21 struct EdgesExtract : vtkm::worklet::WorkletVisitCellsWithPoints
22 {
23     using ControlSignature = void(CellSetIn, FieldOutCell edgeIndices);
24     using ExecutionSignature = void(CellShape, PointIndices, VisitIndex, _2);
25     using InputDomain = _1;
26
27     using ScatterType = vtkm::worklet::ScatterCounting;
28
29     template<typename CellShapeTag,
30             typename PointIndexVecType,
31             typename EdgeIndexVecType>
32     VTKM_EXEC void operator()(CellShapeTag cellShape,
```

(continues on next page)

(continued from previous page)

```

33         const PointIndexVecType& globalPointIndicesForCell,
34         vtkm::IdComponent edgeIndex,
35         EdgeIndexVecType& edgeIndices) const
36     {
37         vtkm::IdComponent numPointsInCell =
38             globalPointIndicesForCell.GetNumberOfComponents();
39
40         vtkm::ErrorCode error;
41
42         vtkm::IdComponent pointInCellIndex0;
43         error = vtkm::exec::CellEdgeLocalIndex(
44             numPointsInCell, 0, edgeIndex, cellShape, pointInCellIndex0);
45         if (error != vtkm::ErrorCode::Success)
46         {
47             this->RaiseError(vtkm::ErrorString(error));
48             return;
49         }
50
51         vtkm::IdComponent pointInCellIndex1;
52         error = vtkm::exec::CellEdgeLocalIndex(
53             numPointsInCell, 1, edgeIndex, cellShape, pointInCellIndex1);
54         if (error != vtkm::ErrorCode::Success)
55         {
56             this->RaiseError(vtkm::ErrorString(error));
57             return;
58         }
59
60         edgeIndices[0] = globalPointIndicesForCell[pointInCellIndex0];
61         edgeIndices[1] = globalPointIndicesForCell[pointInCellIndex1];
62     }
63 };

```

The header file `vtkm/exec/CellFace.h` contains a collection of functions to help identify the faces of a cell.

```

template<typename CellShapeTag>
static inline vtkm::ErrorCode vtkm::exec::CellFaceNumberOfFaces(CellShapeTag shape, vtkm::IdComponent
&result)

```

Get the number of faces in a cell.

#### Parameters

- **shape** – [in] A tag of type `CellShapeTag*` to identify the shape of the cell. This method is overloaded for different shape types.
- **result** – [out] A reference to return the number of faces.

```

template<typename CellShapeTag>
static inline vtkm::ErrorCode vtkm::exec::CellFaceNumberOfPoints(vtkm::IdComponent faceIndex,
CellShapeTag shape,
vtkm::IdComponent &result)

```

Get the number of points in a face.

Given a local index to the face and a shape of the cell, this method returns the number of points in that particular face.

#### Parameters

- **faceIndex** – [in] The index of the face within the cell.
- **shape** – [in] A tag of type `CellShapeTag*` to identify the shape of the cell. This method is overloaded for different shape types.
- **result** – [out] A reference to return the number of points in the selected face.

```
template<typename CellShapeTag>
static inline vtkm::ErrorCode vtkm::exec::CellFaceShape(vtkm::IdComponent faceIndex, CellShapeTag shape,
                                                         vtkm::UInt8 &result)
```

Get the shape of a face.

Given a local index to the face and a shape of the cell, this method returns the identifier for the shape of that face. Faces are always polygons, so it is valid to just to treat the face as a `CELL_SHAPE_POLYGON`. However, the face will be checked to see if it can be further specialized to `CELL_SHAPE_TRIANGLE` or `CELL_SHAPE_QUAD`.

#### Parameters

- **faceIndex** – [in] The index of the face within the cell.
- **shape** – [in] A tag of type `CellShapeTag*` to identify the shape of the cell. This method is overloaded for different shape types.
- **result** – [out] A reference to return the number of points in the selected face.

```
template<typename CellShapeTag>
static inline vtkm::ErrorCode vtkm::exec::CellFaceLocalIndex(vtkm::IdComponent pointIndex,
                                                             vtkm::IdComponent faceIndex, CellShapeTag
                                                             shape, vtkm::IdComponent &result)
```

Given the index for a face of a cell and one of the points on that face, this function returns the point index for the cell.

To get the point indices relative to the data set, the returned index should be used to reference a `PointIndices` list.

#### Parameters

- **pointIndex** – [in] The index of the edge within the cell.
- **faceIndex** – [in] The index of the point on the face.
- **shape** – [in] A tag of type `CellShapeTag*` to identify the shape of the cell. This method is overloaded for different shape types.
- **result** – [out] Reference to put the index of the point relative to the cell (between 0 and the number of points in the cell).

```
template<typename CellShapeTag, typename GlobalPointIndicesVecType>
static inline vtkm::ErrorCode vtkm::exec::CellFaceCanonicalId(vtkm::IdComponent faceIndex,
                                                             CellShapeTag shape, const
                                                             GlobalPointIndicesVecType
                                                             &globalPointIndicesVec, vtkm::Id3 &result)
```

Returns a canonical identifier for a cell face.

Given information about a cell face and the global point indices for that cell, returns a `vtkm::Id3` that contains values that are unique to that face. The values for two faces will be the same if and only if the faces contain the same points.

Note that this property is only true if the mesh is conforming. That is, any two neighboring cells that share a face have the same points on that face. This precludes 2 faces sharing more than a single point or single edge.

The following example demonstrates a triple of worklets that use the cell face functions. As is typical for operations of this nature, the worklets are used in steps to first count entities and then generate new entities. In this case, the first worklet counts the number of faces and the second worklet counts the points in each face. The third worklet generates cells for each face.

Example 6: Using cell face functions.

```

1  struct FacesCount : vtkm::worklet::WorkletVisitCellsWithPoints
2  {
3      using ControlSignature = void(CellSetIn, FieldOutCell numFacesInCell);
4      using ExecutionSignature = void(CellShape, _2);
5      using InputDomain = _1;
6
7      template<typename CellShapeTag>
8      VTKM_EXEC void operator()(CellShapeTag cellShape, vtkm::IdComponent& numFaces) const
9      {
10         vtkm::ErrorCode status = vtkm::exec::CellFaceNumberOfFaces(cellShape, numFaces);
11         if (status != vtkm::ErrorCode::Success)
12         {
13             this->RaiseError(vtkm::ErrorString(status));
14         }
15     }
16 };
17
18 struct FacesCountPoints : vtkm::worklet::WorkletVisitCellsWithPoints
19 {
20     using ControlSignature = void(CellSetIn,
21                                 FieldOutCell numPointsInFace,
22                                 FieldOutCell faceShape);
23     using ExecutionSignature = void(CellShape, VisitIndex, _2, _3);
24     using InputDomain = _1;
25
26     using ScatterType = vtkm::worklet::ScatterCounting;
27
28     template<typename CellShapeTag>
29     VTKM_EXEC void operator()(CellShapeTag cellShape,
30                             vtkm::IdComponent faceIndex,
31                             vtkm::IdComponent& numPointsInFace,
32                             vtkm::UInt8& faceShape) const
33     {
34         vtkm::exec::CellFaceNumberOfPoints(faceIndex, cellShape, numPointsInFace);
35         switch (numPointsInFace)
36         {
37             case 3:
38                 faceShape = vtkm::CELL_SHAPE_TRIANGLE;
39                 break;
40             case 4:
41                 faceShape = vtkm::CELL_SHAPE_QUAD;
42                 break;
43             default:
44                 faceShape = vtkm::CELL_SHAPE_POLYGON;
45                 break;
46         }
47     }

```

(continues on next page)



(continued from previous page)

```

48 };
49
50 struct FacesExtract : vtkm::worklet::WorkletVisitCellsWithPoints
51 {
52     using ControlSignature = void(CellSetIn, FieldOutCell faceIndices);
53     using ExecutionSignature = void(CellShape, PointIndices, VisitIndex, _2);
54     using InputDomain = _1;
55
56     using ScatterType = vtkm::worklet::ScatterCounting;
57
58     template<typename CellShapeTag,
59             typename PointIndexVecType,
60             typename FaceIndexVecType>
61     VTKM_EXEC void operator()(CellShapeTag cellShape,
62                             const PointIndexVecType& globalPointIndicesForCell,
63                             vtkm::IdComponent faceIndex,
64                             FaceIndexVecType& faceIndices) const
65     {
66         vtkm::IdComponent numPointsInFace = faceIndices.GetNumberOfComponents();
67         for (vtkm::IdComponent pointInFaceIndex = 0; pointInFaceIndex < numPointsInFace;
68             pointInFaceIndex++)
69         {
70             vtkm::IdComponent pointInCellIndex;
71             vtkm::ErrorCode error = vtkm::exec::CellFaceLocalIndex(
72                 pointInFaceIndex, faceIndex, cellShape, pointInCellIndex);
73             if (error != vtkm::ErrorCode::Success)
74             {
75                 this->RaiseError(vtkm::ErrorString(error));
76                 return;
77             }
78             faceIndices[pointInFaceIndex] = globalPointIndicesForCell[pointInCellIndex];
79         }
80     }
81 };

```



## MEMORY LAYOUT OF ARRAY HANDLES

Chapter 17 (Basic Array Handles) describes the basics of the `vtkm::cont::ArrayHandle` class, which is the interface to the arrays of data that VTK-m operates on. Recall that `vtkm::cont::ArrayHandle` is a templated class with two template parameters. The first template argument is the type of each item in the array. The second parameter, which is optional, determines how the array is stored in memory. This can be used in a variety of different ways, but its primary purpose is to provide a strategy for laying the data out in memory. This chapter documents the ways in which VTK-m can store and access arrays of data in different layouts.

### 27.1 Basic Memory Layout

If the second storage template parameter of `vtkm::cont::ArrayHandle` is not specified, it defaults to the basic memory layout. This is roughly synonymous with a wrapper around a standard C array, much like `std::vector`. In fact, Section 17.1 (Creating Array Handles) provides examples of wrapping a default `vtkm::cont::ArrayHandle` around either a basic C array or a `std::vector`.

VTK-m provides `vtkm::cont::ArrayHandleBasic` as a convenience class for working with basic array handles. `vtkm::cont::ArrayHandleBasic` is a simple subclass of `vtkm::cont::ArrayHandle` with the default storage in the second template argument (which is `vtkm::cont::StorageTagBasic`). `vtkm::cont::ArrayHandleBasic` and its superclass can be used more or less interchangeably.

```
template<typename T>
```

```
class ArrayHandleBasic : public vtkm::cont::ArrayHandle<T, vtkm::cont::StorageTagBasic>
```

Basic array storage for an array handle.

This array handle references a standard C array. It provides a level of safety and management across devices. This is the default used when no storage is specified. Using this subclass allows access to the underlying raw array.

#### Public Functions

```
inline const T *GetReadPointer() const
```

Gets raw access to the `ArrayHandle`'s data.

Note that the returned array may become invalidated by other operations on the `ArrayHandle`.

```
inline const T *GetReadPointer(vtkm::cont::Token &token) const
```

Gets raw access to the `ArrayHandle`'s data.

#### Parameters

**token** – When a `vtkm::cont::Token` is provided, the array is locked from being used by any write operations until the token goes out of scope.

```
inline T *GetWritePointer() const
```

Gets raw write access to the [ArrayHandle](#)'s data.

Note that the returned array may become invalidated by other operations on the [ArrayHandle](#).

```
inline T *GetWritePointer(vtkm::cont::Token &token) const
```

Gets raw write access to the [ArrayHandle](#)'s data.

#### Parameters

**token** – When a `vtkm::cont::Token` is provided, the array is locked from being used by any read or write operations until the token goes out of scope.

```
inline const T *GetReadPointer(vtkm::cont::DeviceAdapterId device) const
```

Gets raw access to the [ArrayHandle](#)'s data on a particular device.

Note that the returned array may become invalidated by other operations on the [ArrayHandle](#).

#### Parameters

**device** – The device ID or device tag specifying on which device the array will be valid on.

```
inline const T *GetReadPointer(vtkm::cont::DeviceAdapterId device, vtkm::cont::Token &token) const
```

Gets raw access to the [ArrayHandle](#)'s data.

#### Parameters

- **device** – The device ID or device tag specifying on which device the array will be valid on.
- **token** – When a `vtkm::cont::Token` is provided, the array is locked from being used by any write operations until the token goes out of scope.

```
inline T *GetWritePointer(vtkm::cont::DeviceAdapterId device) const
```

Gets raw write access to the [ArrayHandle](#)'s data.

Note that the returned array may become invalidated by other operations on the [ArrayHandle](#).

#### Parameters

**device** – The device ID or device tag specifying on which device the array will be valid on.

```
inline T *GetWritePointer(vtkm::cont::DeviceAdapterId device, vtkm::cont::Token &token) const
```

Gets raw write access to the [ArrayHandle](#)'s data.

#### Parameters

- **device** – The device ID or device tag specifying on which device the array will be valid on.
- **token** – When a `vtkm::cont::Token` is provided, the array is locked from being used by any read or write operations until the token goes out of scope.

Because a `vtkm::cont::ArrayHandleBasic` represents arrays as a standard C array, it is possible to get a pointer to this array using either `vtkm::cont::ArrayHandleBasic::GetReadPointer()` or `vtkm::cont::ArrayHandleBasic::GetWritePointer()`.

Example 1: Getting a standard C array from a basic array handle.

```
1 void LegacyFunction(const int* data);  
2  
3 void UseArrayWithLegacy(const vtkm::cont::ArrayHandle<vtkm::Int32> array)  
4 {  
5     vtkm::cont::ArrayHandleBasic<vtkm::Int32> basicArray = array;
```

(continues on next page)

(continued from previous page)

```

6   vtkm::cont::Token token; // Token prevents array from changing while in scope.
7   const int* cArray = basicArray.GetReadPointer(token);
8   LegacyFunction(cArray);
9   // When function returns, token goes out of scope and array can be modified.
10  }

```

### Did You Know?

When you get an array pointer this way, the `vtkm::cont::ArrayHandle` still has a reference to it. If using multiple threads, you can use a `vtkm::cont::Token` object to lock the array. When the token is used to get a pointer, it will lock the array as long as the token exists. [Example 1](#) demonstrates using a `vtkm::cont::Token`.

## 27.2 Structure of Arrays

The basic `vtkm::cont::ArrayHandle` stores `vtkm::Vec` objects in sequence. In this sense, a basic array is an *Array of Structures* (AOS). Another approach is to store each component of the structure (i.e., the `vtkm::Vec`) in a separate array. This is known as a *Structure of Arrays* (SOA). There are advantages to this approach including potentially better cache performance and the ability to combine arrays already represented as separate components without copying them. Arrays of this nature are represented with a `vtkm::cont::ArrayHandleSOA`, which is a subclass of `vtkm::cont::StorageTagSOA`.

template<typename T>

class **ArrayHandleSOA** : public `vtkm::cont::ArrayHandle<T>`, `vtkm::cont::StorageTagSOA`>

An `ArrayHandle` that for Vecs stores each component in a separate physical array.

`ArrayHandleSOA` behaves like a regular `ArrayHandle` (with a basic storage) except that if you specify a `ValueType` of a `Vec` or a `Vec`-like, it will actually store each component in a separate physical array. When data are retrieved from the array, they are reconstructed into `Vec` objects as expected.

The intention of this array type is to help cover the most common ways data is laid out in memory. Typically, arrays of data are either an “array of structures” like the basic storage where you have a single array of structures (like `Vec`) or a “structure of arrays” where you have an array of a basic type (like `float`) for each component of the data being represented. The `ArrayHandleSOA` makes it easy to cover this second case without creating special types.

`ArrayHandleSOA` can be constructed from a collection of `ArrayHandle` with basic storage. This allows you to construct `Vec` arrays from components without deep copies.

### Public Functions

inline **ArrayHandleSOA**(const std::array<ComponentArrayType, NUM\_COMPONENTS>&componentArrays)

Construct an `ArrayHandleSOA` from a collection of component arrays.

```

vtkm::cont::ArrayHandle<T> components1;
vtkm::cont::ArrayHandle<T> components2;
vtkm::cont::ArrayHandle<T> components3;
// Fill arrays...

```

(continues on next page)

(continued from previous page)

```
std::array<T, 3> allComponents{ components1, components2, components3 };
vtkm::cont::make_ArrayHandleSOA<vtkm::Vec<T, 3>>vecarray(allComponents);
```

inline **ArrayHandleSOA**(const std::vector<ComponentArrayType> &componentArrays)

Construct an *ArrayHandleSOA* from a collection of component arrays.

```
vtkm::cont::ArrayHandle<T> components1;
vtkm::cont::ArrayHandle<T> components2;
vtkm::cont::ArrayHandle<T> components3;
// Fill arrays...

std::vector<T> allComponents{ components1, components2, components3 };
vtkm::cont::make_ArrayHandleSOA<vtkm::Vec<T, 3>>vecarray(allComponents);
```

inline **ArrayHandleSOA**(std::initializer\_list<ComponentArrayType> &&componentArrays)

Construct an *ArrayHandleSOA* from a collection of component arrays.

```
vtkm::cont::ArrayHandle<T> components1;
vtkm::cont::ArrayHandle<T> components2;
vtkm::cont::ArrayHandle<T> components3;
// Fill arrays...

vtkm::cont::make_ArrayHandleSOA<vtkm::Vec<T, 3>> vecarray(
    { components1, components2, components3 });
```

inline **ArrayHandleSOA**(std::initializer\_list<std::vector<ComponentType>> &&componentVectors)

Construct an *ArrayHandleSOA* from a collection of component arrays.

The data is copied from the std::vectors to the array handle.

```
std::vector<T> components1;
std::vector<T> components2;
std::vector<T> components3;
// Fill arrays...

vtkm::cont::ArrayHandleSOA<vtkm::Vec<T, 3>> vecarray(
    { components1, components2, components3 });
```

template<typename **Allocator**, typename ...**RemainingVectors**>

inline **ArrayHandleSOA**(vtkm::CopyFlag copy, const std::vector<ComponentType, *Allocator*> &vector0, *RemainingVectors*&&... componentVectors)

Construct an *ArrayHandleSOA* from a collection of component arrays.

The first argument is a *vtkm::CopyFlag* to determine whether the input arrays should be copied. The component arrays are listed as arguments. This only works if all the templated arguments are of type std::vector<ComponentType>.

```
std::vector<T> components1;
std::vector<T> components2;
std::vector<T> components3;
// Fill arrays...
```

(continues on next page)

(continued from previous page)

```

vtkm::cont::ArrayHandleSOA<vtkm::Vec<T, 3>> vecarray(
    vtkm::CopyFlag::On, components1, components2, components3);

```

template<typename ...**RemainingVectors**>

inline **ArrayHandleSOA**(vtkm::CopyFlag copy, std::vector<ComponentType> &&vector0,  
*RemainingVectors*&&... componentVectors)

Construct an *ArrayHandleSOA* from a collection of component arrays.

The first argument is a *vtkm::CopyFlag* to determine whether the input arrays should be copied. The component arrays are listed as arguments. This only works if all the templated arguments are rvalues of type `std::vector<ComponentType>`.

```

std::vector<T> components1;
std::vector<T> components2;
std::vector<T> components3;
// Fill arrays...

vtkm::cont::ArrayHandleSOA<vtkm::Vec<T, N> vecarray(vtkm::CopyFlag::Off,
                                                    std::move(components1),
                                                    std::move(components2),
                                                    std::move(components3);

```

inline **ArrayHandleSOA**(std::initializer\_list<const ComponentType\*> componentArrays, vtkm::Id length,  
*vtkm::CopyFlag* copy)

Construct an *ArrayHandleSOA* from a collection of component arrays.

```

T* components1;
T* components2;
T* components3;
// Fill arrays...

vtkm::cont::ArrayHandleSOA<vtkm::Vec<T, 3>>(
    { components1, components2, components3 }, size, vtkm::CopyFlag::On);

```

template<typename ...**RemainingArrays**>

inline **ArrayHandleSOA**(vtkm::Id length, vtkm::CopyFlag copy, const ComponentType \*array0, const  
*RemainingArrays*&&... componentArrays)

Construct an *ArrayHandleSOA* from a collection of component arrays.

The component arrays are listed as arguments. This only works if all the templated arguments are of type `ComponentType*`.

```

T* components1;
T* components2;
T* components3;
// Fill arrays...

vtkm::cont::ArrayHandleSOA<vtkm::Vec<T, 3>> vecarray(
    size, vtkm::CopyFlag::On, components1, components2, components3);

```

inline vtkm::cont::ArrayHandleBasic<ComponentType> **GetArray**(vtkm::IdComponent index) const

Get a basic array representing the component for the given index.

```
inline void SetArray(vtkm::IdComponent index, const ComponentArrayType &array)
```

Replace a component array.

`vtkm::cont::ArrayHandleSOA` can be constructed and allocated just as a basic array handle. Additionally, you can use its constructors or the `vtkm::cont::make_ArrayHandleSOA()` functions to build a `vtkm::cont::ArrayHandleSOA` from basic `vtkm::cont::ArrayHandle`'s that hold the components.

```
template<typename ValueType>
```

```
ArrayHandleSOA<ValueType> vtkm::cont::make_ArrayHandleSOA(std::initializer_list<vtkm::cont::ArrayHandle<typename  
vtkm::VecTraits<ValueType>::ComponentType,  
vtkm::cont::StorageTagBasic>>  
&&componentArrays)
```

Create a `vtkm::cont::ArrayHandleSOA` with an initializer list of array handles.

```
vtkm::cont::ArrayHandle<T> components1;  
vtkm::cont::ArrayHandle<T> components2;  
vtkm::cont::ArrayHandle<T> components3;  
// Fill arrays...  
  
auto vecarray = vtkm::cont::make_ArrayHandleSOA<vtkm::Vec<T, 3>>(  
    { components1, components2, components3 });
```

```
template<typename ComponentType, typename ...RemainingArrays>
```

```
ArrayHandleSOA<vtkm::Vec<ComponentType, internal::VecSizeFromRemaining<RemainingArrays...>::value>> vtkm::cont::make_
```

Create a `vtkm::cont::ArrayHandleSOA` with a number of array handles.

This only works if all the templated arguments are of type `vtkm::cont::ArrayHandle<ComponentType>`.

```
vtkm::cont::ArrayHandle<T> components1;  
vtkm::cont::ArrayHandle<T> components2;  
vtkm::cont::ArrayHandle<T> components3;  
// Fill arrays...  
  
auto vecarray =  
    vtkm::cont::make_ArrayHandleSOA(components1, components2, components3);
```

```
template<typename ValueType>
```



```
ArrayHandleSOA<ValueType> vtkm::cont::make_ArrayHandleSOA(std::initializer_list<std::vector<typename
                                                         vtkm::VecTraits<ValueType>::ComponentType>>
                                                         &&componentVectors)
```

Create a `vtkm::cont::ArrayHandleSOA` with an initializer list of `std::vector`.

The data is copied from the `std::vectors` to the array handle.

```
std::vector<T> components1;
std::vector<T> components2;
std::vector<T> components3;
// Fill arrays...

auto vecarray = vtkm::cont::make_ArrayHandleSOA<vtkm::Vec<T, 3>>(
    { components1, components2, components3 });
```

```
template<typename ComponentType, typename ...RemainingVectors>
ArrayHandleSOA<vtkm::Vec<ComponentType, internal::VecSizeFromRemaining<RemainingVectors...>::value>> vtkm::cont::make
```

Create a `vtkm::cont::ArrayHandleSOA` with a number of `std::vector`.

The first argument is a `vtkm::CopyFlag` to determine whether the input arrays should be copied. The component arrays are listed as arguments. This only works if all the templated arguments are of type `std::vector<ComponentType>`.

```
std::vector<T> components1;
std::vector<T> components2;
std::vector<T> components3;
// Fill arrays...

auto vecarray = vtkm::cont::make_ArrayHandleSOA(
    vtkm::CopyFlag::On, components1, components2, components3);
```

```
template<typename ComponentType, typename ...RemainingVectors>
```

*ArrayHandleSOA*<vtkm::Vec<*ComponentType*, internal::VecSizeFromRemaining<*RemainingVectors...*>::value>> vtkm::cont::make

Create a *vtkm::cont::ArrayHandleSOA* with a number of *std::vector*.

The first argument is a *vtkm::CopyFlag* to determine whether the input arrays should be copied. The component arrays are listed as arguments. This only works if all the templated arguments are rvalues of type *std::vector<ComponentType>*.

```
std::vector<T> components1;
std::vector<T> components2;
std::vector<T> components3;
// Fill arrays...

auto vecarray = vtkm::cont::make_ArrayHandleSOA(vtkm::CopyFlag::Off,
                                                std::move(components1),
                                                std::move(components2),
                                                std::move(components3));
```

template<typename **ComponentType**, typename ...**RemainingVectors**>  
*ArrayHandleSOA*<vtkm::Vec<*ComponentType*, internal::VecSizeFromRemaining<*RemainingVectors...*>::value>> vtkm::cont::make

Create a *vtkm::cont::ArrayHandleSOA* with a number of *std::vector*.

This only works if all the templated arguments are rvalues of type *std::vector<ComponentType>*.

```
std::vector<T> components1;
std::vector<T> components2;
std::vector<T> components3;
// Fill arrays...

auto vecarray = vtkm::cont::make_ArrayHandleSOAMove(
    std::move(components1), std::move(components2), std::move(components3));
```

template<typename **ValueType**>

```
ArrayHandleSOA<ValueType> vtkm::cont::make_ArrayHandleSOA(std::initializer_list<const typename
                                                         vtkm::VecTraits<ValueType>::ComponentType*>
                                                         &&componentVectors, vtkm::Id length,
                                                         vtkm::CopyFlag copy)
```

Create a `vtkm::cont::ArrayHandleSOA` with an initializer list of C arrays.

```
T* components1;
T* components2;
T* components3;
// Fill arrays...

auto vecarray = vtkm::cont::make_ArrayHandleSOA<vtkm::Vec<T, 3>>(
    { components1, components2, components3 }, size, vtkm::CopyFlag::On);
```

```
template<typename ComponentType, typename ...RemainingArrays>
```

```
ArrayHandleSOA<vtkm::Vec<ComponentType, internal::VecSizeFromRemaining<RemainingArrays...>::value>> vtkm::cont::make_
```

Create a `vtkm::cont::ArrayHandleSOA` with a number of C arrays.

This only works if all the templated arguments are of type `ComponentType*`.

```
T* components1;
T* components2;
T* components3;
// Fill arrays...

auto vecarray = vtkm::cont::make_ArrayHandleSOA(
    size, vtkm::CopyFlag::On, components1, components2, components3);
```

Example 2: Creating an SOA array handle from component arrays.

```
1 vtkm::cont::ArrayHandle<vtkm::FloatDefault> component1;
2 vtkm::cont::ArrayHandle<vtkm::FloatDefault> component2;
3 vtkm::cont::ArrayHandle<vtkm::FloatDefault> component3;
```

(continues on next page)

(continued from previous page)

```
4 // Fill component arrays...
5
6 vtkm::cont::ArrayHandleSOA<vtkm::Vec3f> soaArray =
7   vtkm::cont::make_ArrayHandleSOA(component1, component2, component3);
```

---

### Did You Know?

In addition to constructing a `vtkm::cont::ArrayHandleSOA` from its component arrays, you can get the component arrays back out using the `vtkm::cont::ArrayHandleSOA::GetArray()` method.

---

## 27.3 Strided Arrays

`vtkm::cont::ArrayHandleBasic` operates on a tightly packed array. That is, each value follows immediately after the proceeding value in memory. However, it is often convenient to access values at different strides or offsets. This allows representations of data that are not tightly packed in memory. The `vtkm::cont::ArrayHandleStride` class allows arrays with different data packing.

template<typename T>

class **ArrayHandleStride** : public vtkm::cont::ArrayHandle<T, vtkm::cont::StorageTagStride>

An `ArrayHandle` that accesses a basic array with strides and offsets.

`ArrayHandleStride` is a simple `ArrayHandle` that accesses data with a prescribed stride and offset. You specify the stride and offset at construction. So when a portal for this `ArrayHandle` Gets or Sets a value at a specific index, the value accessed in the underlying C array is:

$(\text{index} * \text{stride}) + \text{offset}$

Optionally, you can also specify a modulo and divisor. If they are specified, the index mangling becomes:

$((\text{index} / \text{divisor}) \% \text{modulo}) * \text{stride} + \text{offset}$

You can “disable” any of the aforementioned operations by setting them to the following values (most of which are arithmetic identities):

- stride: 1
- offset: 0
- modulo: 0
- divisor: 1

Note that all of these indices are referenced by the `ValueType` of the array. So, an `ArrayHandleStride<vtkm::Float32>` with an offset of 1 will actually offset by 4 bytes (the size of a `vtkm::Float32`).

`ArrayHandleStride` is used to provide a unified type for pulling a component out of an `ArrayHandle`. This way, you can iterate over multiple components in an array without having to implement a template instance for each vector size or representation.

## Public Functions

```
inline ArrayHandleStride(const vtkm::cont::ArrayHandle<T>, vtkm::cont::StorageTagBasic> &array,
                        vtkm::Id numValues, vtkm::Id stride, vtkm::Id offset, vtkm::Id modulo = 0,
                        vtkm::Id divisor = 1)
```

Construct an *ArrayHandleStride* from a basic array with specified access patterns.

```
inline vtkm::Id GetStride() const
```

Get the stride that values are accessed.

The stride is the spacing between consecutive values. The stride is measured in terms of the number of values. A stride of 1 means a fully packed array. A stride of 2 means selecting every other values.

```
inline vtkm::Id GetOffset() const
```

Get the offset to start reading values.

The offset is the number of values to skip before the first value. The offset is measured in terms of the number of values. An offset of 0 means the first value at the beginning of the array.

The offset is unaffected by the stride and dictates where the strides starts counting. For example, given an array with size 3 vectors packed into an array, a strided array referencing the middle component will have offset 1 and stride 3.

```
inline vtkm::Id GetModulo() const
```

Get the modulus of the array index.

When the index is modulo a value, it becomes the remainder after dividing by that value. The effect of the modulus is to cause the index to repeat over the values in the array.

If the modulo is set to 0, then it is ignored.

```
inline vtkm::Id GetDivisor() const
```

Get the divisor of the array index.

The index is divided by the divisor before the other effects. The default divisor of 1 will have no effect on the indexing. Setting the divisor to a value greater than 1 has the effect of repeating each value that many times.

```
inline vtkm::cont::ArrayHandleBasic<T> GetBasicArray() const
```

Return the underlying data as a basic array handle.

It is common for the same basic array to be shared among multiple *vtkm::cont::ArrayHandleStride* objects.

The most common use of *vtkm::cont::ArrayHandleStride* is to pull components out of arrays. *vtkm::cont::ArrayHandleStride* is seldom constructed directly. Rather, VTK-m has mechanisms to extract a component from an array. To extract a component directly from a *vtkm::cont::ArrayHandle*, use *vtkm::cont::ArrayExtractComponent()*.

```
template<typename T, typename S>
```

```
vtkm::cont::ArrayHandleStride<typename vtkm::VecTraits<T>::BaseComponentType> vtkm::cont::ArrayExtractComponent (con
```

Pulls a component out of an *ArrayHandle*.

Given an *ArrayHandle* of any type, *ArrayExtractComponent* returns an *ArrayHandleStride* of the base component type that contains the data for the specified array component. This function can be used to apply an operation on an *ArrayHandle* one component at a time. Because the array type is always *ArrayHandleStride*, you can drastically cut down on the number of templates to instantiate (at a possible cost to performance).

Note that *ArrayExtractComponent* will flatten out the indices of any vec value type and return an *ArrayExtractComponent* of the base component type. For example, if you call *ArrayExtractComponent* on an *ArrayHandle* with a value type of *vtkm::Vec<vtkm::Vec<vtkm::Float32, 2>, 3>*, you will get an *ArrayExtractComponent<vtkm::Float32>* returned. The *componentIndex* provided will be applied to the nested vector in depth first order. So in the previous example, a *componentIndex* of 0 gets the values at [0][0], *componentIndex* of 1 gets [0][1], *componentIndex* of 2 gets [1][0], and so on.

Some *ArrayHandles* allow this method to return an *ArrayHandleStride* that shares the same memory as the the original *ArrayHandle*. This form will be used if possible. In this case, if data are written into the *ArrayHandleStride*, they are also written into the original *ArrayHandle*. However, other forms will require copies into a new array. In this case, writes into *ArrayHandleStride* will not affect the original *ArrayHandle*.

For some operations, such as writing into an output array, this behavior of shared arrays is necessary. For this case, the optional argument *allowCopy* can be set to *vtkm::CopyFlag::Off* to prevent the copying behavior into the return *ArrayHandleStride*. If this is the case, an *ErrorBadValue* is thrown. If the arrays can be shared, they always will be regardless of the value of *allowCopy*.

Many forms of *ArrayHandle* have optimized versions to pull out a component. Some, however, do not. In these cases, a fallback array copy, done in serial, will be performed. A warning will be logged to alert users of this likely performance bottleneck.

As an implementation note, this function should not be overloaded directly. Instead, *ArrayHandle* implementations should provide a specialization of *vtkm::cont::internal::ArrayExtractComponentImpl*.

The main advantage of extracting components this way is to convert data represented in different types of arrays into an array of a single type. For example, *vtkm::cont::ArrayHandleStride* can represent a component from either a *vtkm::cont::ArrayHandleBasic* or a *vtkm::cont::ArrayHandleSOA* by just using different stride values. This is used by *vtkm::cont::UnknownArrayHandle::ExtractComponent()* and elsewhere to create a concrete array handle class without knowing the actual class.

---

## Common Errors

Many, but not all, of VTK-m's arrays can be represented by a *vtkm::cont::ArrayHandleStride* directly without copying. If VTK-m cannot easily create a *vtkm::cont::ArrayHandleStride* when attempting such an operation, it

will use a slow copying fallback. A warning will be issued whenever this happens. Be on the lookout for such warnings and consider changing the data representation when that happens.

## 27.4 Runtime Vec Arrays

Because many of the devices VTK-m runs on cannot efficiently allocate memory while an algorithm is running, the data held in `vtkm::cont::ArrayHandle`'s are usually required to be a static size. For example, the `vtkm::Vec` object often used as the value type for `vtkm::cont::ArrayHandle` has a number of components that must be defined at compile time.

This is a problem in cases where the size of a vector object cannot be determined at compile time. One class to help alleviate this problem is `vtkm::cont::ArrayHandleRuntimeVec`. This array handle stores data in the same way as `vtkm::cont::ArrayHandleBasic` with a `vtkm::Vec` value type, but the size of the `Vec` can be set at runtime.

```
template<typename ComponentType>
```

```
class ArrayHandleRuntimeVec : public
```

```
vtkm::cont::ArrayHandle<vtkm::VecFromPortal<ArrayHandleBasic<ComponentType>::WritePortalType>,
```

```
vtkm::cont::StorageTagRuntimeVec>
```

Fancy array handle for a basic array with runtime selected vec size.

It is sometimes the case that you need to create an array of `Vectors` where the number of components is not known until runtime. This is problematic for normal `ArrayHandles` because you have to specify the size of the `Vectors` as a template parameter at compile time. `ArrayHandleRuntimeVec` can be used in this case.

Note that caution should be used with `ArrayHandleRuntimeVec` because the size of the `Vec` values is not known at compile time. Thus, the value type of this array is forced to a special `VecFromPortal` class that can cause surprises if treated as a `Vec`. In particular, the static `NUM_COMPONENTS` expression does not exist. Furthermore, new variables of type `VecFromPortal` cannot be created. This means that simple operators like `+` will not work because they require an intermediate object to be created. (Equal operators like `+=` do work because they are given an existing variable to place the output.)

It is possible to provide an `ArrayHandleBasic` of the same component type as the underlying storage for this array. In this case, the array will be accessed much in the same manner as `ArrayHandleGroupVec`.

`ArrayHandleRuntimeVec` also allows you to convert the array to an `ArrayHandleBasic` of the appropriate `Vec` type (or component type). A runtime check will be performed to make sure the number of components matches.

### Public Functions

```
inline ArrayHandleRuntimeVec(vtkm::IdComponent numComponents, const ComponentsArrayType
                             &componentsArray = ComponentsArrayType{ })
```

Construct an `ArrayHandleRuntimeVec` with a given number of components.

#### Parameters

- **numComponents** – The size of the `Vectors` stored in the array. This must be specified at the time of construction.
- **componentsArray** – This optional parameter allows you to supply a basic array that holds the components. This provides a mechanism to group consecutive values into vectors.

```
inline vtkm::IdComponent GetNumberOfComponents() const
```

Return the number of components in each vec value.

```
inline vtkm::cont::ArrayHandleBasic<ComponentType> GetComponentsArray() const
```

Return a basic array containing the components stored in this array.

The returned array is shared with this object. Modifying the contents of one array will modify the other.

```
template<typename ValueType>
```

```
inline void AsArrayHandleBasic(vtkm::cont::ArrayHandle<ValueType> &array) const
```

Converts the array to that of a basic array handle.

This method converts the *ArrayHandleRuntimeVec* to a simple *ArrayHandleBasic*. This is useful if the *ArrayHandleRuntimeVec* is passed to a routine that works on an array of a specific *Vec* size (or scalars).

After a runtime check, the array can be converted to a typical array and used as such.

```
template<typename ArrayType>
```

```
inline ArrayType AsArrayHandleBasic() const
```

Converts the array to that of a basic array handle.

This method converts the *ArrayHandleRuntimeVec* to a simple *ArrayHandleBasic*. This is useful if the *ArrayHandleRuntimeVec* is passed to a routine that works on an array of a specific *Vec* size (or scalars).

After a runtime check, the array can be converted to a typical array and used as such.

A *vtkm::cont::ArrayHandleRuntimeVec* is easily created from existing data using one of the *vtkm::cont::make\_ArrayHandleRuntimeVec()* functions.

```
template<typename T>
```

```
auto vtkm::cont::make_ArrayHandleRuntimeVec(vtkm::IdComponent numComponents, const  
                                             vtkm::cont::ArrayHandle<T, vtkm::cont::StorageTagBasic>  
                                             &componentsArray = vtkm::cont::ArrayHandle<T,  
                                             vtkm::cont::StorageTagBasic>{ })
```

*make\_ArrayHandleRuntimeVec* is convenience function to generate an *ArrayHandleRuntimeVec*.

It takes the number of components stored in each value's *Vec*, which must be specified on the construction of the *ArrayHandleRuntimeVec*. If not specified, the number of components is set to 1. *make\_ArrayHandleRuntimeVec* can also optionally take an existing array of components, which will be grouped into *Vec* values based on the specified number of components.

```
template<typename T>
```

```
auto vtkm::cont::make_ArrayHandleRuntimeVec(const vtkm::cont::ArrayHandle<T,  
                                             vtkm::cont::StorageTagBasic> &componentsArray)
```

Converts a basic array handle into an *ArrayHandleRuntimeVec* with 1 component.

The constructed array is essentially equivalent but of a different type.

VTK-m also provides several convenience functions to convert a basic C array or *std::vector* to a *vtkm::cont::ArrayHandleRuntimeVec*.

```
template<typename T>
```

```
auto vtkm::cont::make_ArrayHandleRuntimeVec(vtkm::IdComponent numComponents, const T *array,  
                                             vtkm::Id numberOfValues, vtkm::CopyFlag copy)
```

A convenience function for creating an *ArrayHandleRuntimeVec* from a standard C array.

```
template<typename T>
```

```
auto vtkm::cont::make_ArrayHandleRuntimeVecMove(vtkm::IdComponent numComponents, T *&array,  
                                             vtkm::Id numberOfValues,  
                                             vtkm::cont::internal::BufferInfo::Deleter deleter =  
                                             internal::SimpleArrayDeleter<T>,  
                                             vtkm::cont::internal::BufferInfo::Reallocator reallocator  
                                             = internal::SimpleArrayReallocator<T>)
```



A convenience function to move a user-allocated array into an `ArrayHandleRuntimeVec`.

The provided array pointer will be reset to `nullptr`. If the array was not allocated with the `new[]` operator, then deleter and reallocator functions must be provided.

```
template<typename T, typename Allocator>
auto vtkm::cont::make_ArrayHandleRuntimeVec(vtkm::IdComponent numComponents, const std::vector<T,
                                           Allocator> &array, vtkm::CopyFlag copy)
```

A convenience function for creating an `ArrayHandleRuntimeVec` from an `std::vector`.

```
template<typename T, typename Allocator>
auto vtkm::cont::make_ArrayHandleRuntimeVecMove(vtkm::IdComponent numComponents, std::vector<T,
                                                Allocator> &&array)
```

Move an `std::vector` into an `ArrayHandleRuntimeVec`.

The advantage of this class is that a `vtkm::cont::ArrayHandleRuntimeVec` can be created in a routine that does not know the number of components at runtime and then later retrieved as a basic `vtkm::cont::ArrayHandle` with a `vtkm::Vec` of the correct size. This often consists of a file reader or other data ingestion creating `vtkm::cont::ArrayHandleRuntimeVec` objects and storing them in `vtkm::cont::UnknownArrayHandle`, which is used as an array container for `vtkm::cont::DataSet`. Filters that then subsequently operate on the `vtkm::cont::DataSet` can retrieve the data as a `vtkm::cont::ArrayHandle` of the appropriate `vtkm::Vec` size.

Example 3: Loading a data with runtime component size and using with a static sized filter.

```
1 void ReadArray(std::vector<float>& data, int& numComponents);
2
3 vtkm::cont::UnknownArrayHandle LoadData()
4 {
5     // Read data from some external source where the vector size is determined at runtime.
6     std::vector<vtkm::Float32> data;
7     int numComponents;
8     ReadArray(data, numComponents);
9
10    // Resulting ArrayHandleRuntimeVec gets wrapped in an UnknownArrayHandle
11    return vtkm::cont::make_ArrayHandleRuntimeVecMove(
12        static_cast<vtkm::IdComponent>(numComponents), std::move(data));
13 }
14
15 void UseVecArray(const vtkm::cont::UnknownArrayHandle& array)
16 {
17     using ExpectedArrayType = vtkm::cont::ArrayHandle<vtkm::Vec3f_32>;
18     if (!array.CanConvert<ExpectedArrayType>())
19     {
20         throw vtkm::cont::ErrorBadType("Array unexpected type.");
21     }
22
23     ExpectedArrayType concreteArray = array.AsArrayHandle<ExpectedArrayType>();
24     // Do something with concreteArray...
25 }
26
27 void LoadAndRun()
28 {
29     // Load data in a routine that does not know component size until runtime.
```

(continues on next page)

(continued from previous page)

```

30  vtkm::cont::UnknownArrayHandle array = LoadData();
31
32  // Use the data in a method that requires an array of static size.
33  // This will work as long as the `Vec` size matches correctly (3 in this case).
34  UseVecArray(array);
35  }

```

### Did You Know?

Wrapping a basic array in a `vtkm::cont::ArrayHandleRuntimeVec` has a similar effect as wrapping the array in a `vtkm::cont::ArrayHandleGroupVec`. The difference is in the context in which they are used. If the size of the Vec is known at compile time *and* the array is going to immediately be used (such as operated on by a worklet), then `vtkm::cont::ArrayHandleGroupVec` should be used. However, if the Vec size is not known or the array will be stored in an object like `vtkm::cont::UnknownArrayHandle`, then `vtkm::cont::ArrayHandleRuntimeVec` is a better choice.

It is also possible to get a `vtkm::cont::ArrayHandleRuntimeVec` from a `vtkm::cont::UnknownArrayHandle` that was originally stored as a basic array. This is convenient for operations that want to operate on arrays with an unknown Vec size.

Example 4: Using `vtkm::cont::ArrayHandleRuntimeVec` to get an array regardless of the size of the contained `vtkm::Vec` values.

```

1  template<typename T>
2  void WriteData(const T* data, std::size_t size, int numComponents);
3
4  void WriteVTKmArray(const vtkm::cont::UnknownArrayHandle& array)
5  {
6      bool writeSuccess = false;
7      auto doWrite = [&](auto componentType) {
8          using ComponentType = decltype(componentType);
9          using VecArrayType = vtkm::cont::ArrayHandleRuntimeVec<ComponentType>;
10         if (array.CanConvert<VecArrayType>())
11         {
12             // Get the array as a runtime Vec.
13             VecArrayType runtimeVecArray = array.AsArrayHandle<VecArrayType>();
14
15             // Get the component array.
16             vtkm::cont::ArrayHandleBasic<ComponentType> componentArray =
17                 runtimeVecArray.GetComponentsArray();
18
19             // Use the general function to write the data.
20             WriteData(componentArray.GetReadPointer(),
21                     componentArray.GetNumberOfValues(),
22                     runtimeVecArray.GetNumberOfComponentsFlat());
23
24             writeSuccess = true;
25         }
26     };
27
28     // Figure out the base component type, retrieve the data (regardless

```

(continues on next page)

(continued from previous page)

```
29  // of vec size), and write out the data.  
30  vtkm::ListForEach(doWrite, vtkm::TypeListBaseC{});  
31 }
```



# **Part V**

## **Core Development**



# **Part VI**

## **Appendix**





## ACKNOWLEDGEMENTS

### 28.1 Contributors

This book includes contributions from the VTK-m community including the VTK-m development team and the user community. We would like to thank the following people for their significant contributions to this text:

**Vicente Bolea**, **Nickolas Davis**, **Matthew Letter**, and **Nick Thompson** for their help keeping the user's guide up to date with the VTK-m source code.

**Sujin Philip**, **Robert Maynard**, **James Kress**, **Abhishek Yenpure**, **Mark Kim**, and **Hank Childs** for their descriptions of numerous filters.

**Allison Vacanti** for her documentation of..

**David Pugmire** for his documentation of..

**Abhishek Yenpure** and **Li-Ta Lo** for their documentation of locator structures..

**Li-Ta Lo** for his documentation of random array handles and particle density filters.

**James Kress** for his documentation on VTK-m's testing classes.

**Manish Mathai** for his documentation of rendering features..

### 28.2 Funding



This project has been funded in whole or in part with Federal funds from the Department of Energy, including from Oak Ridge National Laboratory, Los Alamos National Laboratory, and Sandia National Laboratories.

This manuscript has been authored in part by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The US government retains and the publisher, by accepting the article for publication, acknowledges that the US government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for US government purposes.

Sandia National Laboratories is a multitechnology laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

This research was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation's exascale computing imperative.

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program.

## **LICENSE**

Copyright (c) 2014-2023 Kitware Inc., National Technology & Engineering Solutions of Sandia, LLC (NTESS), UT-Battelle, LLC., Los Alamos National Security, LLC., All rights reserved.

Under the terms of Contract DE-NA0003525 with NTESS, the U.S. Government retains certain rights in this software.

Under the terms of Contract DE-AC52-06NA25396 with Los Alamos National Laboratory (LANL), the U.S. Government retains certain rights in this software.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of Kitware nor the names of any contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHORS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

All product names mentioned herein are the trademarks of their respective owners.



---

CHAPTER  
**THIRTY**

---

**INDEX**



## Symbols

- : implicit function
  - box, 225
- \_\_device\_\_, 233
- \_\_host\_\_, 233
- 2D
  - camera rendering, 186
- 3D
  - camera rendering, 187

## A

- actor, 19
  - rendering, 163
- AMR
  - arrays, 147
- AMR arrays
  - filter, 147
- AOS, 409
- argc, 35
- argv, 35
- array handle
  - basic, 235, 407
  - const, 245
  - deep copy, 243
  - divisor, 416
  - memory layout, 407
  - modulo, 416
  - offset, 416
  - storage, 244
  - stride, 416
- assert
  - errors, 205
  - static, 206
- azimuth
  - camera rendering, 188

## B

- background
  - color, 174
- basic array handle, 407
- blanked cell
  - remove, 111

- box
  - : implicit function, 225
- BUILD\_SHARED\_LIBS
  - variable, 11

## C

- camera
  - interactive rendering, 191
  - pinhole, 187
  - rendering, 176
  - rendering 2D, 186
  - rendering 3D, 187
  - rendering azimuth, 188
  - rendering clipping range, 187
  - rendering elevation, 188
  - rendering far clip plane, 187
  - rendering field of view, 187
  - rendering focal point, 187
  - rendering look at, 187
  - rendering mouse, 191
  - rendering near clip plane, 187
  - rendering pan, 185
  - rendering position, 187
  - rendering reset, 189
  - rendering up, 187
  - rendering view range, 186
  - rendering view up, 187
  - rendering zoom, 186
- canvas, 19
  - ray tracer, 165
  - rendering, 165
- cell, 389
  - derivative, 399
  - edge, 400
  - face, 402
  - gradient, 399
  - interpolation, 398
  - parametric coordinates, 396
  - point, 400
  - shape, 400
  - shape edge, 57
  - shape face, 57

- shape point, 57
  - world coordinates, 396
  - cell average
    - filter, 114
  - cell gradients, 155
  - cell measures
    - filter, 137
  - cell neighborhood
    - worklet, 335
  - cell set, 39, 57
    - explicit, 61
    - permutation, 65
    - single type, 64
    - structured, 59
  - cell shape
    - tag, 389
  - cell traits, 393
  - CELL\_SHAPE\_EMPTY (C++ enumerator), 391
  - CELL\_SHAPE\_HEXAHEDRON (C++ enumerator), 391
  - CELL\_SHAPE\_LINE (C++ enumerator), 389
  - CELL\_SHAPE\_POLY\_LINE (C++ enumerator), 389
  - CELL\_SHAPE\_POLYGON (C++ enumerator), 391
  - CELL\_SHAPE\_PYRAMID (C++ enumerator), 391
  - CELL\_SHAPE\_QUAD (C++ enumerator), 391
  - CELL\_SHAPE\_TETRA (C++ enumerator), 391
  - CELL\_SHAPE\_TRIANGLE (C++ enumerator), 389
  - CELL\_SHAPE\_VERTEX (C++ enumerator), 389
  - CELL\_SHAPE\_WEDGE (C++ enumerator), 391
  - clean grid
    - filter, 93
  - clip
    - field, 98
    - filter, 98, 100
    - implicit function, 100
  - clipping range
    - camera rendering, 187
  - cloud in cell
    - particle density, 104
  - CMake, 9
    - configuration, 9
    - VTK-m package, 13
    - VTK-m package libraries, 13
    - VTK-m package variables, 14
    - VTK-m package version, 33
  - CMAKE\_BUILD\_TYPE
    - variable, 11, 12
  - CMAKE\_INSTALL\_PREFIX
    - variable, 11
  - CMAKE\_PREFIX\_PATH
    - envvar, 13
    - variable, 13
  - CMakeLists.txt, 21
  - color
    - background, 174
    - foreground, 174
  - color tables
    - rendering, 193
  - command
    - find\_package, 13, 14, 33
    - project, 21
    - target\_link\_libraries, 13
  - composite vectors
    - filter, 115
  - compression
    - zfp, 160
  - configuration
    - CMake, 9
  - connected components
    - cell, 95
    - field, 95
    - filter, 95
    - image, 95
  - contour
    - filter, 96
  - contouring
    - filter, 96
  - control
    - environment, 231, 232
    - modifier, 233
  - control environment
    - errors, 203
  - control signature, 247, 248
  - convert to point cloud
    - filter, 133
  - coordinate system, 39, 71
  - coordinate system transform
    - cylindrical, 116
    - spherical, 124
  - cross product
    - filter, 151
  - CUDA, 11, 233
  - cuda, 207
  - cylinder
    - implicit function, 223
  - cylindrical coordinate system transform
    - filter, 116
- ## D
- data set, 39
    - building, 39
    - cell set, 57
    - clean, 93
    - coordinate system, 71
    - field, 69
    - filter, 361
    - partitioned, 71
  - data set with field
    - filter, 364



- Debug, 11
- deep copy
  - array handle, 243
- density
  - cloud in cell, particle, 104
  - filter, 101
  - histogram, 101
  - nearest grid point, particle, 103
- derivative
  - cell, 399
- determinant, 385
- device adapter, 207
  - any, 209
  - id, 208
  - runtime tracker, 209
  - scoped runtime tracker, 211
  - tag, 207
  - undefined, 209
- dimensionality
  - tag type, 281
- divisor
  - array handle, 416
- dot product
  - filter, 153

## E

- edge, 400
  - cell shape, 57
- elevation
  - camera rendering, 188
  - filter, 121
- entity extraction
  - filter, 106
- environment, 231, 232
  - control, 231, 232
  - execution, 231, 232
- envvar
  - CMAKE\_PREFIX\_PATH, 13
- error codes, 306
- error handling
  - worklet, 367
- errors
  - assert, 205
  - control environment, 203
  - execution environment, 367
- execution
  - environment, 231, 232
  - modifier, 233
- execution environment
  - errors, 367
- execution signature, 247, 248
- explicit cell set, 61
  - single type, 64
- explicit mesh, 45

- connectivity, 45
  - offsets, 45
  - shapes, 45
- exploded view, 133
- export macro
  - filter, 356
- external faces
  - filter, 106
- extract geometry
  - filter, 107
- extract points
  - filter, 108
- extract structured
  - filter, 110

## F

- face, 402
  - cell shape, 57
  - external, 106
- far clip plane
  - camera rendering, 187
- field, 39, 69, 84
  - clip, 98
  - connected components, 95
  - filter, 356
  - range, 70
- field conversion
  - filter, 114
- field map
  - worklet, 317, 318
- field of view
  - camera rendering, 187
- field to colors
  - filter, 116
- field transform
  - filter, 115
- fields
  - filter, 84
  - filter input, 85
  - filter passing, 86
- file I/O, 77
  - read, 18, 77
  - write, 79
- filter, 18, 83, 231
  - AMR arrays, 147
  - cell average, 114
  - cell measures, 137
  - clean grid, 93
  - clip, 98, 100
  - composite vectors, 115
  - compression, 160
  - connected components, 95
  - contour, 96
  - contouring, 96

- convert to point cloud, 133
- cross product, 151
- cylindrical coordinate system transform, 116
- data set, 361
- data set with field, 364
- density, 101
- dot product, 153
- elevation, 121
- entity extraction, 106
- export macro, 356
- external faces, 106
- extract geometry, 107
- extract points, 108
- extract structured, 110
- field, 356
- field conversion, 114
- field to colors, 116
- field transform, 115
- fields, 84
- FTLE, 130
- generate ids, 119
- ghost cell, 111
- ghost cell classification, 139
- gradients, 155
- histogram, 101
- histogram sampling, 149
- implementation, 253, 355
- input fields, 85
- isosurface, 96
- Lagrangian coherent structures, 130
- log, 120
- merge data sets, 148
- mesh quality, 139
- passing fields, 86
- pathlines, 128
- point average, 115
- point elevation, 121
- point transform, 122
- probe, 150
- shrink, 133
- slice, 98
- spherical coordinate system transform, 124
- split sharp edges, 134
- stream surface, 129
- streamlines, 127
- surface normals, 157
- surface simplification, 136
- tetrahedralize, 135
- threshold, 112
- transform, 122
- triangulate, 135
- tube, 135
- using cells, 359
- vector magnitude, 159
- vertex clustering, 136
- warp, 124
- zfp, 160
- find\_package
  - command, 13, 14, 33
- finite time Lyapunov exponent, *see* FTLE
- flow, 126
  - pathlines, 128
  - stream surface, 129
  - streamlines, 127
- focal point
  - camera rendering, 187
- foreground
  - color, 174
- frustum
  - implicit function, 226
- FTLE
  - filter, 130
- function modifier, 233
- function types, 248
- functions
  - implicit, 219
- functor, 231
- G**
  - general
    - implicit function, 227
  - generate ids
    - filter, 119
  - geometry refinement, 132
  - ghost cell
    - classify, 139
    - filter, 111
    - remove, 111
  - ghost cell classification
    - filter, 139
  - git, 9
  - gradient
    - cell, 399
  - gradients
    - filter, 155
- H**
  - histogram, 351
    - density, 101
    - filter, 101
  - histogram sampling
    - filter, 149
- I**
  - I/O, 77
  - id

- device adapter, 208
- identity
  - matrix, 385
- image, 40
  - connected components, 95
- implementation
  - filter, 355
- implicit function, 219
  - clip, 100
  - cylinder, 223
  - frustum, 226
  - general, 227
  - plane, 219
  - sphere, 222
- initialization, 17, 35
- input
  - fields, filter, 85
- input domain, 247, 249
- Intel Threading Building Blocks, *see* TBB, 207
- interactive
  - rendering, 190
  - rendering camera, 191
  - rendering OpenGL, 190
- interpolation
  - cell, 398
- interval volume, 98
- inverse
  - matrix, 385
- isosurface
  - filter, 96
- isovolume, 98

## K

- kernel, 231
- Kokkos, 208
- kokkos, 11

## L

- Lagrangian coherent structures
  - filter, 130
- LCS, *see* Lagrangian coherent structures
- less, 286
- level of detail, 136
- libraries
  - CMake VTK-m package, 13
- linear system, 386
- lists, 288
  - type, 289
- LOD, 136
- log
  - filter, 120
- logging, 309
  - initialization, 309
  - levels, 309

- loguru, 309
- look at
  - camera rendering, 187

## M

- magnitude, 159
- map, 317
- map field, 318
- mapper, 19
  - rendering, 167
- math, 369
- matrix, 384
  - determinant, 385
  - identity, 385
  - inverse, 385
  - transpose, 386
- merge data sets
  - filter, 148
- mesh information, 137
  - quality, 139
- mesh quality
  - filter, 139
- meshless data, 133
- metaprogramming, 288
- method modifier, 233
- modifier
  - control, 233
  - execution, 233
- modulo
  - array handle, 416
- mouse
  - camera rendering, 191
  - pan, 192
  - rotation, 191
  - zoom, 193
- MPI, 11
- multi-block, 147
- multiple components
  - tag vector, 285

## N

- namespace, 232
- near clip plane
  - camera rendering, 187
- nearest grid point
  - particle density, 103
- neighborhood
  - worklet, 331
- Newton's method, 386
- normals, 157
- numeric
  - tag type, 281

## O

- offset
  - array handle, 416
- OpenGL
  - interactive rendering, 190
- OpenMP, 11, 207

## P

- packages, 232
- pan
  - camera rendering, 185
  - mouse, 192
  - rendering, 192
- parametric coordinates
  - cell, 396
- particle
  - density cloud in cell, 104
  - densitynearest grid point, 103
- partitioned data set, 71
- passing
  - fields, filter, 86
- pathlines
  - filter, 128
- permutation cell set, 65
- pinhole
  - camera, 187
- plane
  - implicit function, 219
- point, 400
  - cell shape, 57
- point average
  - filter, 115
- point elevation
  - filter, 121
- point gradients, 155
- point neighborhood
  - worklet, 317, 332
- point transform
  - filter, 122
- position
  - camera rendering, 187
- probe
  - filter, 150
- project
  - command, 21
- pseudocolor, 193

## R

- range
  - field, 70
- ray tracer
  - canvas, 165
- read file, 18, 77

- rectilinear grid, 42
- reduce by key
  - worklet, 317, 345
- regular grid, 40
- Release, 11
- rendering, 19, 163
  - 2D, camera, 186
  - 3D, camera, 187
  - actor, 163
  - azimuth, camera, 188
  - camera, 176
  - camera, interactive, 191
  - canvas, 165
  - clipping range, camera, 187
  - color tables, 193
  - elevation, camera, 188
  - far clip plane, camera, 187
  - field of view, camera, 187
  - focal point, camera, 187
  - interactive, 190
  - look at, camera, 187
  - mapper, 167
  - mouse, camera, 191
  - near clip plane, camera, 187
  - OpenGL, interactive, 190
  - pan, 192
  - pan, camera, 185
  - position, camera, 187
  - reset, camera, 189
  - rotation, 191
  - scene, 164
  - up, camera, 187
  - view, 172
  - view range, camera, 186
  - view up, camera, 187
  - wireframe, 176
  - zoom, 193
  - zoom, camera, 186
- reset
  - camera rendering, 189
- rotation
  - mouse, 191
  - rendering, 191
- runtime device tracker, 209
  - scoped, 211

## S

- scene, 19
  - rendering, 164
- scoped device adapter, 211
- serial, 207
- shape
  - cell, 400
  - edge, 400

- edge, cell, 57
- face, 402
- face, cell, 57
- point, cell, 57
- tag, 389
- shrink
  - filter, 133
- signature, 248
  - control, 247, 248
  - execution, 247, 248
  - tags, 248
- single type cell set, 64
- size\_t, 26
- slice
  - filter, 98
- SOA, 409
- sphere
  - implicit function, 222
- spherical coordinate system transform
  - filter, 124
- split sharp edges
  - filter, 134
- static
  - tag vector, 285
- static assert, 206
- std::size\_t, 26
- std::vector, 241
- storage
  - array handle, 244
- stream surface
  - filter, 129
- streamlines
  - filter, 127
- stride
  - array handle, 416
- structured cell set, 59
- surface normals
  - filter, 157
- surface simplification
  - filter, 136

## T

- tag, 280
  - cell shape, 389
  - device adapter, 207
  - shape, 389
  - type dimensionality, 281
  - type numeric, 281
  - vector multiple components, 285
  - vector static, 285
- target\_link\_libraries
  - command, 13
- TBB, 11, 207
- template metaprogramming, 288

- tetrahedralize
  - filter, 135
- thread name, 309
- threshold
  - filter, 112
- timer, 215
- topology map
  - worklet, 317
- traits, 280
  - type, 280
  - vector, 283
- transform
  - filter, 122
- transpose
  - matrix, 386
- triangulate
  - filter, 135
- tube
  - filter, 135
- type
  - dimensionality, tag, 281
  - lists, 289
  - numeric, tag, 281
  - traits, 280

## U

- uniform grid, 40
- unstructured grid, 45
- up
  - camera rendering, 187
- using cells
  - filter, 359

## V

- variable
  - BUILD\_SHARED\_LIBS, 11
  - CMAKE\_BUILD\_TYPE, 11, 12
  - CMAKE\_INSTALL\_PREFIX, 11
  - CMAKE\_PREFIX\_PATH, 13
  - vtkm::cont, 13
  - vtkm::filter, 13, 14
  - vtkm::filter\_contour, 14
  - vtkm::filter\_field\_transform, 14
  - vtkm::io, 14
  - vtkm::rendering, 14
  - vtkm::source, 14
  - VTKm\_ENABLE\_BENCHMARKS, 11
  - VTKm\_ENABLE\_CUDA, 11, 14
  - VTKm\_ENABLE\_EXAMPLES, 11
  - VTKm\_ENABLE\_KOKKOS, 11
  - VTKm\_ENABLE\_Kokkos, 15
  - VTKm\_ENABLE\_MPI, 11, 15
  - VTKm\_ENABLE\_OPENMP, 11, 15
  - VTKm\_ENABLE\_RENDERING, 11, 14, 15

- VTKm\_ENABLE\_TBB, [11](#), [15](#)
- VTKm\_ENABLE\_TESTING, [11](#)
- VTKm\_ENABLE\_TUTORIALS, [11](#)
- VTKm\_FOUND, [14](#)
- VTKm\_USE\_64BIT\_IDS, [11](#), [26](#)
- VTKm\_USE\_DOUBLE\_PRECISION, [11](#), [26](#), [27](#)
- VTKm\_VERSION, [14](#), [33](#)
- VTKm\_VERSION\_FULL, [14](#), [33](#)
- VTKm\_VERSION\_MAJOR, [14](#), [33](#)
- VTKm\_VERSION\_MINOR, [14](#), [33](#)
- VTKm\_VERSION\_PATCH, [14](#), [33](#)
- variables
  - CMake VTK-m package, [14](#)
- Vec-like, [267](#), [283](#)
- vector, [241](#)
  - multiple components, tag, [285](#)
  - static, tag, [285](#)
  - traits, [283](#)
- vector analysis, [151](#), [382](#)
- vector magnitude
  - filter, [159](#)
- version, [33](#)
  - CMake VTK-m package, [33](#)
  - macro, [33](#)
- vertex clustering
  - filter, [136](#)
- view, [19](#)
  - rendering, [172](#)
- view range
  - camera rendering, [186](#)
- view up
  - camera rendering, [187](#)
- visit cells
  - worklet, [317](#), [322](#)
- visit points
  - worklet, [317](#), [327](#)
- VTK-m package
  - CMake, [13](#)
  - libraries, CMake, [13](#)
  - variables, CMake, [14](#)
  - version, CMake, [33](#)
- vtkm::Abs (C++ function), [377](#)
- vtkm::ACos (C++ function), [378](#)
- vtkm::ACosH (C++ function), [378](#)
- vtkm::Apply (C++ function), [304](#)
- vtkm::ASin (C++ function), [378](#)
- vtkm::ASinH (C++ function), [378](#)
- vtkm::ATan (C++ function), [379](#)
- vtkm::ATan2 (C++ function), [379](#)
- vtkm::ATanH (C++ function), [379](#)
- vtkm::Bounds (C++ struct), [273](#)
- vtkm::Bounds::Area (C++ function), [274](#)
- vtkm::Bounds::Bounds (C++ function), [273](#)
- vtkm::Bounds::Center (C++ function), [274](#)
- vtkm::Bounds::Contains (C++ function), [273](#)
- vtkm::Bounds::Include (C++ function), [274](#)
- vtkm::Bounds::Intersection (C++ function), [274](#)
- vtkm::Bounds::IsEmpty (C++ function), [273](#)
- vtkm::Bounds::MaxCorner (C++ function), [274](#)
- vtkm::Bounds::MinCorner (C++ function), [274](#)
- vtkm::Bounds::operator+ (C++ function), [274](#)
- vtkm::Bounds::Union (C++ function), [274](#)
- vtkm::Bounds::Volume (C++ function), [273](#)
- vtkm::Bounds::X (C++ member), [275](#)
- vtkm::Bounds::Y (C++ member), [275](#)
- vtkm::Bounds::Z (C++ member), [275](#)
- vtkm::Box (C++ class), [225](#)
- vtkm::Box::Box (C++ function), [225](#)
- vtkm::Box::GetBounds (C++ function), [226](#)
- vtkm::Box::GetMaxPoint (C++ function), [225](#)
- vtkm::Box::GetMinPoint (C++ function), [225](#)
- vtkm::Box::Gradient (C++ function), [226](#)
- vtkm::Box::SetBounds (C++ function), [226](#)
- vtkm::Box::SetMaxPoint (C++ function), [225](#)
- vtkm::Box::SetMinPoint (C++ function), [225](#)
- vtkm::Box::Value (C++ function), [226](#)
- vtkm::Cbrt (C++ function), [373](#)
- vtkm::Ceil (C++ function), [375](#), [376](#)
- vtkm::CellShapeIdToTag (C++ struct), [392](#)
- vtkm::CellShapeTagEmpty (C++ struct), [391](#)
- vtkm::CellShapeTagGeneric (C++ struct), [392](#)
- vtkm::CellShapeTagGeneric::Id (C++ member), [392](#)
- vtkm::CellShapeTagHexahedron (C++ struct), [391](#)
- vtkm::CellShapeTagLine (C++ struct), [389](#)
- vtkm::CellShapeTagPolygon (C++ struct), [391](#)
- vtkm::CellShapeTagPolyLine (C++ struct), [389](#)
- vtkm::CellShapeTagPyramid (C++ struct), [391](#)
- vtkm::CellShapeTagQuad (C++ struct), [391](#)
- vtkm::CellShapeTagTetra (C++ struct), [391](#)
- vtkm::CellShapeTagTriangle (C++ struct), [389](#)
- vtkm::CellShapeTagVertex (C++ struct), [389](#)
- vtkm::CellShapeTagWedge (C++ struct), [391](#)
- vtkm::CellTopologicalDimensionsTag (C++ struct), [394](#)
- vtkm::CellTraits (C++ struct), [393](#)
- vtkm::CellTraits::IsSizeFixed (C++ type), [394](#)
- vtkm::CellTraits::NUM\_POINTS (C++ member), [394](#)
- vtkm::CellTraits::TOPOLOGICAL\_DIMENSIONS (C++ member), [394](#)
- vtkm::CellTraits::TopologicalDimensionsTag (C++ type), [394](#)
- vtkm::CellTraitsTagSizeFixed (C++ struct), [394](#)
- vtkm::CellTraitsTagSizeVariable (C++ struct), [394](#)
- vtkm::cont
  - variable, [13](#)
- vtkm::cont::ArrayCopy (C++ function), [244](#)

vtkm::cont::ArrayExtractComponent (C++ function), 417  
 vtkm::cont::ArrayHandle (C++ class), 235  
 vtkm::cont::ArrayHandle::~~ArrayHandle (C++ function), 236  
 vtkm::cont::ArrayHandle::Allocate (C++ function), 237  
 vtkm::cont::ArrayHandle::AllocateAndFill (C++ function), 238  
 vtkm::cont::ArrayHandle::ArrayHandle (C++ function), 236  
 vtkm::cont::ArrayHandle::DeepCopyFrom (C++ function), 240  
 vtkm::cont::ArrayHandle::Enqueue (C++ function), 240  
 vtkm::cont::ArrayHandle::Fill (C++ function), 238  
 vtkm::cont::ArrayHandle::GetBuffers (C++ function), 240  
 vtkm::cont::ArrayHandle::GetNumberOfComponentsFlat (C++ function), 237  
 vtkm::cont::ArrayHandle::GetNumberOfValues (C++ function), 237  
 vtkm::cont::ArrayHandle::GetStorage (C++ function), 236  
 vtkm::cont::ArrayHandle::IsOnDevice (C++ function), 239  
 vtkm::cont::ArrayHandle::IsOnHost (C++ function), 239  
 vtkm::cont::ArrayHandle::operator= (C++ function), 236  
 vtkm::cont::ArrayHandle::operator== (C++ function), 236  
 vtkm::cont::ArrayHandle::PrepareForInPlace (C++ function), 239  
 vtkm::cont::ArrayHandle::PrepareForInput (C++ function), 239  
 vtkm::cont::ArrayHandle::PrepareForOutput (C++ function), 239  
 vtkm::cont::ArrayHandle::ReadPortal (C++ function), 236  
 vtkm::cont::ArrayHandle::ReleaseResources (C++ function), 239  
 vtkm::cont::ArrayHandle::ReleaseResourcesExecution (C++ function), 238  
 vtkm::cont::ArrayHandle::SyncControlArray (C++ function), 239  
 vtkm::cont::ArrayHandle::WritePortal (C++ function), 237  
 vtkm::cont::ArrayHandleBasic (C++ class), 407  
 vtkm::cont::ArrayHandleBasic::GetReadPointer (C++ function), 407, 408  
 vtkm::cont::ArrayHandleBasic::GetWritePointer (C++ function), 407, 408  
 vtkm::cont::ArrayHandleRuntimeVec (C++ class), 419  
 vtkm::cont::ArrayHandleRuntimeVec::ArrayHandleRuntimeVec (C++ function), 419  
 vtkm::cont::ArrayHandleRuntimeVec::AsArrayHandleBasic (C++ function), 420  
 vtkm::cont::ArrayHandleRuntimeVec::GetComponentsArray (C++ function), 419  
 vtkm::cont::ArrayHandleRuntimeVec::GetNumberOfComponents (C++ function), 419  
 vtkm::cont::ArrayHandleSOA (C++ class), 409  
 vtkm::cont::ArrayHandleSOA::ArrayHandleSOA (C++ function), 409–411  
 vtkm::cont::ArrayHandleSOA::GetArray (C++ function), 411  
 vtkm::cont::ArrayHandleSOA::SetArray (C++ function), 411  
 vtkm::cont::ArrayHandleStride (C++ class), 416  
 vtkm::cont::ArrayHandleStride::ArrayHandleStride (C++ function), 417  
 vtkm::cont::ArrayHandleStride::GetBasicArray (C++ function), 417  
 vtkm::cont::ArrayHandleStride::GetDivisor (C++ function), 417  
 vtkm::cont::ArrayHandleStride::GetModulo (C++ function), 417  
 vtkm::cont::ArrayHandleStride::GetOffset (C++ function), 417  
 vtkm::cont::ArrayHandleStride::GetStride (C++ function), 417  
 vtkm::cont::BoundsCompute (C++ function), 73, 74  
 vtkm::cont::BoundsGlobalCompute (C++ function), 74  
 vtkm::cont::CellSet (C++ class), 57  
 vtkm::cont::CellSet::DeepCopy (C++ function), 58  
 vtkm::cont::CellSet::GetCellPointIds (C++ function), 58  
 vtkm::cont::CellSet::GetCellShape (C++ function), 58  
 vtkm::cont::CellSet::GetNumberOfCells (C++ function), 58  
 vtkm::cont::CellSet::GetNumberOfPoints (C++ function), 58  
 vtkm::cont::CellSet::GetNumberOfPointsInCell (C++ function), 58  
 vtkm::cont::CellSet::NewInstance (C++ function), 58  
 vtkm::cont::CellSet::PrintSummary (C++ function), 58  
 vtkm::cont::CellSet::ReleaseResourcesExecution (C++ function), 58  
 vtkm::cont::CellSetExplicit (C++ class), 61  
 vtkm::cont::CellSetExplicit::AddCell (C++ function), 62



<code>vtkm::cont::CellSetExplicit::CompleteAddingCells</code> (C++ function), 62	<code>vtkm::cont::CellSetPermutation::GetFullCellSet</code> (C++ function), 66
<code>vtkm::cont::CellSetExplicit::DeepCopy</code> (C++ function), 61	<code>vtkm::cont::CellSetPermutation::GetNumberOfCells</code> (C++ function), 66
<code>vtkm::cont::CellSetExplicit::Fill</code> (C++ function), 62	<code>vtkm::cont::CellSetPermutation::GetNumberOfPoints</code> (C++ function), 66
<code>vtkm::cont::CellSetExplicit::GetCellPointIds</code> (C++ function), 61	<code>vtkm::cont::CellSetPermutation::GetNumberOfPointsInCell</code> (C++ function), 66
<code>vtkm::cont::CellSetExplicit::GetCellShape</code> (C++ function), 61	<code>vtkm::cont::CellSetPermutation::GetValidCellIds</code> (C++ function), 66
<code>vtkm::cont::CellSetExplicit::GetNumberOfCells</code> (C++ function), 61	<code>vtkm::cont::CellSetPermutation::NewInstance</code> (C++ function), 66
<code>vtkm::cont::CellSetExplicit::GetNumberOfPoints</code> (C++ function), 61	<code>vtkm::cont::CellSetPermutation::PrintSummary</code> (C++ function), 66
<code>vtkm::cont::CellSetExplicit::GetNumberOfPointsInCell</code> (C++ function), 61	<code>vtkm::cont::CellSetPermutation::ReleaseResourcesExecution</code> (C++ function), 66
<code>vtkm::cont::CellSetExplicit::NewInstance</code> (C++ function), 61	<code>vtkm::cont::CellSetSingleType</code> (C++ class), 64
<code>vtkm::cont::CellSetExplicit::PrepareToAddCells</code> (C++ function), 61	<code>vtkm::cont::CellSetSingleType::AddCell</code> (C++ function), 65
<code>vtkm::cont::CellSetExplicit::PrintSummary</code> (C++ function), 61	<code>vtkm::cont::CellSetSingleType::CompleteAddingCells</code> (C++ function), 65
<code>vtkm::cont::CellSetExplicit::ReleaseResourcesExecution</code> (C++ function), 61	<code>vtkm::cont::CellSetSingleType::DeepCopy</code> (C++ function), 65
<code>vtkm::cont::CellSetExtrude</code> (C++ class), 67	<code>vtkm::cont::CellSetSingleType::Fill</code> (C++ function), 65
<code>vtkm::cont::CellSetExtrude::DeepCopy</code> (C++ function), 68	<code>vtkm::cont::CellSetSingleType::GetCellShape</code> (C++ function), 65
<code>vtkm::cont::CellSetExtrude::GetCellPointIds</code> (C++ function), 68	<code>vtkm::cont::CellSetSingleType::NewInstance</code> (C++ function), 65
<code>vtkm::cont::CellSetExtrude::GetCellShape</code> (C++ function), 67	<code>vtkm::cont::CellSetSingleType::PrepareToAddCells</code> (C++ function), 65
<code>vtkm::cont::CellSetExtrude::GetNumberOfCells</code> (C++ function), 67	<code>vtkm::cont::CellSetSingleType::PrintSummary</code> (C++ function), 65
<code>vtkm::cont::CellSetExtrude::GetNumberOfPoints</code> (C++ function), 67	<code>vtkm::cont::CellSetStructured</code> (C++ class), 59
<code>vtkm::cont::CellSetExtrude::GetNumberOfPointsInCell</code> (C++ function), 67	<code>vtkm::cont::CellSetStructured::DeepCopy</code> (C++ function), 59
<code>vtkm::cont::CellSetExtrude::NewInstance</code> (C++ function), 68	<code>vtkm::cont::CellSetStructured::GetCellPointIds</code> (C++ function), 59
<code>vtkm::cont::CellSetExtrude::PrintSummary</code> (C++ function), 68	<code>vtkm::cont::CellSetStructured::GetCellShape</code> (C++ function), 59
<code>vtkm::cont::CellSetExtrude::ReleaseResourcesExecution</code> (C++ function), 68	<code>vtkm::cont::CellSetStructured::GetNumberOfCells</code> (C++ function), 59
<code>vtkm::cont::CellSetPermutation</code> (C++ class), 65	<code>vtkm::cont::CellSetStructured::GetNumberOfPoints</code> (C++ function), 59
<code>vtkm::cont::CellSetPermutation::CellSetPermutation</code> (C++ function), 66	<code>vtkm::cont::CellSetStructured::GetNumberOfPointsInCell</code> (C++ function), 59
<code>vtkm::cont::CellSetPermutation::DeepCopy</code> (C++ function), 66	<code>vtkm::cont::CellSetStructured::GetPointDimensions</code> (C++ function), 59
<code>vtkm::cont::CellSetPermutation::Fill</code> (C++ function), 66	<code>vtkm::cont::CellSetStructured::NewInstance</code> (C++ function), 59
<code>vtkm::cont::CellSetPermutation::GetCellPointIds</code> (C++ function), 66	<code>vtkm::cont::CellSetStructured::PrintSummary</code> (C++ function), 59
<code>vtkm::cont::CellSetPermutation::GetCellShape</code> (C++ function), 66	<code>vtkm::cont::CellSetStructured::ReleaseResourcesExecution</code> (C++ function), 59



vtkm::cont::CellSetStructured::SetPointDimensions (C++ function), 196  
 vtkm::cont::ColorTable (C++ class), 193  
 vtkm::cont::ColorTable::AddPoint (C++ function), 196  
 vtkm::cont::ColorTable::AddPointAlpha (C++ function), 197  
 vtkm::cont::ColorTable::AddPointHSV (C++ function), 196  
 vtkm::cont::ColorTable::AddSegment (C++ function), 196  
 vtkm::cont::ColorTable::AddSegmentAlpha (C++ function), 197  
 vtkm::cont::ColorTable::AddSegmentHSV (C++ function), 196  
 vtkm::cont::ColorTable::Clear (C++ function), 196  
 vtkm::cont::ColorTable::ClearAlpha (C++ function), 196  
 vtkm::cont::ColorTable::ClearColors (C++ function), 196  
 vtkm::cont::ColorTable::ColorTable (C++ function), 194, 195  
 vtkm::cont::ColorTable::FillColorTableFromDataPointer (C++ function), 198  
 vtkm::cont::ColorTable::FillOpacityTableFromDataPointer (C++ function), 198, 199  
 vtkm::cont::ColorTable::GetModifiedCount (C++ function), 200  
 vtkm::cont::ColorTable::GetNumberOfPoints (C++ function), 197  
 vtkm::cont::ColorTable::GetNumberOfPointsAlpha (C++ function), 198  
 vtkm::cont::ColorTable::GetPoint (C++ function), 197  
 vtkm::cont::ColorTable::GetPointAlpha (C++ function), 198  
 vtkm::cont::ColorTable::GetPresets (C++ function), 200  
 vtkm::cont::ColorTable::GetRange (C++ function), 196  
 vtkm::cont::ColorTable::LoadPreset (C++ function), 195  
 vtkm::cont::ColorTable::MakeDeepCopy (C++ function), 195  
 vtkm::cont::ColorTable::PrepareForExecution (C++ function), 200  
 vtkm::cont::ColorTable::RemovePoint (C++ function), 197  
 vtkm::cont::ColorTable::RemovePointAlpha (C++ function), 198  
 vtkm::cont::ColorTable::RescaleToRange (C++ function), 196  
 vtkm::cont::ColorTable::ReverseAlpha (C++ function), 196  
 vtkm::cont::ColorTable::ReverseColors (C++ function), 196  
 vtkm::cont::ColorTable::Sample (C++ function), 199, 200  
 vtkm::cont::ColorTable::SetAboveRangeColor (C++ function), 195  
 vtkm::cont::ColorTable::SetBelowRangeColor (C++ function), 195  
 vtkm::cont::ColorTable::SetClampingOn (C++ function), 195  
 vtkm::cont::ColorTable::UpdatePoint (C++ function), 197  
 vtkm::cont::ColorTable::UpdatePointAlpha (C++ function), 198  
 vtkm::cont::ConvertNumComponentsToOffsets (C++ function), 62  
 vtkm::cont::CoordinateSystem (C++ class), 71  
 vtkm::cont::CoordinateSystem::GetBounds (C++ function), 71  
 vtkm::cont::DataSet (C++ class), 39  
 vtkm::cont::DataSet::AddCellField (C++ function), 55, 56  
 vtkm::cont::DataSet::AddPointField (C++ function), 55  
 vtkm::cont::DataSetBuilderExplicit (C++ class), 47  
 vtkm::cont::DataSetBuilderExplicit::Create (C++ function), 48–50  
 vtkm::cont::DataSetBuilderExplicitIterative (C++ class), 52  
 vtkm::cont::DataSetBuilderExplicitIterative::AddCell (C++ function), 53, 54  
 vtkm::cont::DataSetBuilderExplicitIterative::AddCellPoint (C++ function), 54  
 vtkm::cont::DataSetBuilderExplicitIterative::AddPoint (C++ function), 52, 53  
 vtkm::cont::DataSetBuilderExplicitIterative::Begin (C++ function), 52  
 vtkm::cont::DataSetBuilderExplicitIterative::Create (C++ function), 54  
 vtkm::cont::DataSetBuilderRectilinear (C++ class), 42  
 vtkm::cont::DataSetBuilderRectilinear::Create (C++ function), 42–44  
 vtkm::cont::DataSetBuilderUniform (C++ class), 40  
 vtkm::cont::DataSetBuilderUniform::Create (C++ function), 40, 41  
 vtkm::cont::DeviceAdapterId (C++ struct), 208  
 vtkm::cont::DeviceAdapterId::GetName (C++ function), 208  
 vtkm::cont::DeviceAdapterId::GetValue (C++ function), 208

```

vtkm::cont::DeviceAdapterId::IsValueValid (C++ function), 208
vtkm::cont::DeviceAdapterTagAny (C++ struct), 209
vtkm::cont::DeviceAdapterTagCuda (C++ struct), 207
vtkm::cont::DeviceAdapterTagKokkos (C++ struct), 208
vtkm::cont::DeviceAdapterTagOpenMP (C++ struct), 207
vtkm::cont::DeviceAdapterTagSerial (C++ struct), 207
vtkm::cont::DeviceAdapterTagTBB (C++ struct), 207
vtkm::cont::DeviceAdapterTagUndefined (C++ struct), 209
vtkm::cont::Error (C++ class), 203
vtkm::cont::Error::GetIsDeviceIndependent (C++ function), 204
vtkm::cont::Error::GetMessage (C++ function), 204
vtkm::cont::Error::GetStackTrace (C++ function), 204
vtkm::cont::Error::what (C++ function), 204
vtkm::cont::ErrorBadAllocation (C++ class), 204
vtkm::cont::ErrorBadDevice (C++ class), 204
vtkm::cont::ErrorBadType (C++ class), 204
vtkm::cont::ErrorBadValue (C++ class), 204
vtkm::cont::ErrorExecution (C++ class), 204
vtkm::cont::ErrorFilterExecution (C++ class), 204
vtkm::cont::ErrorInternal (C++ class), 204
vtkm::cont::ErrorUserAbort (C++ class), 204
vtkm::cont::Field (C++ class), 69
vtkm::cont::Field::Association (C++ enum), 69
vtkm::cont::Field::Association::Any (C++ enumerator), 69
vtkm::cont::Field::Association::Cells (C++ enumerator), 70
vtkm::cont::Field::Association::Global (C++ enumerator), 70
vtkm::cont::Field::Association::Partitions (C++ enumerator), 70
vtkm::cont::Field::Association::Points (C++ enumerator), 70
vtkm::cont::Field::Association::WholeDataSet (C++ enumerator), 69
vtkm::cont::Field::GetAssociation (C++ function), 69
vtkm::cont::Field::GetData (C++ function), 69
vtkm::cont::Field::GetName (C++ function), 69
vtkm::cont::Field::GetRange (C++ function), 70
vtkm::cont::Field::IsCellField (C++ function), 70
vtkm::cont::Field::IsGlobalField (C++ function), 70
vtkm::cont::Field::IsPartitionsField (C++ function), 70
vtkm::cont::Field::IsPointField (C++ function), 70
vtkm::cont::Field::IsWholeDataSetField (C++ function), 70
vtkm::cont::FieldRangeCompute (C++ function), 74
vtkm::cont::FieldRangeGlobalCompute (C++ function), 74, 75
vtkm::cont::GetHumanReadableSize (C++ function), 315
vtkm::cont::GetLogLevelName (C++ function), 311
vtkm::cont::GetLogThreadName (C++ function), 309
vtkm::cont::GetRuntimeDeviceTracker (C++ function), 210
vtkm::cont::GetSizeString (C++ function), 315
vtkm::cont::GetStackTrace (C++ function), 315
vtkm::cont::GetStderrLogLevel (C++ function), 312
vtkm::cont::Initialize (C++ function), 35
vtkm::cont::InitializeOptions (C++ enum), 36
vtkm::cont::InitializeOptions::AddHelp (C++ enumerator), 36
vtkm::cont::InitializeOptions::DefaultAnyDevice (C++ enumerator), 36
vtkm::cont::InitializeOptions::ErrorOnBadArgument (C++ enumerator), 36
vtkm::cont::InitializeOptions::ErrorOnBadOption (C++ enumerator), 36
vtkm::cont::InitializeOptions::None (C++ enumerator), 36
vtkm::cont::InitializeOptions::RequireDevice (C++ enumerator), 36
vtkm::cont::InitializeOptions::Strict (C++ enumerator), 37
vtkm::cont::InitializeResult (C++ struct), 35
vtkm::cont::InitializeResult::Device (C++ member), 36
vtkm::cont::InitializeResult::Usage (C++ member), 36
vtkm::cont::Invoker (C++ struct), 250
vtkm::cont::Invoker::GetDevice (C++ function), 250
vtkm::cont::Invoker::Invoker (C++ function), 250
vtkm::cont::Invoker::operator() (C++ function), 250
vtkm::cont::LogLevel (C++ enum), 310
vtkm::cont::LogLevel::Cast (C++ enumerator), 311
vtkm::cont::LogLevel::DevicesEnabled (C++ enumerator), 310
vtkm::cont::LogLevel::Error (C++ enumerator),

```

310

`vtkm::cont::LogLevel::Fatal` (C++ *enumerator*), 310

`vtkm::cont::LogLevel::Info` (C++ *enumerator*), 310

`vtkm::cont::LogLevel::KernelLaunches` (C++ *enumerator*), 311

`vtkm::cont::LogLevel::MemCont` (C++ *enumerator*), 310

`vtkm::cont::LogLevel::MemExec` (C++ *enumerator*), 310

`vtkm::cont::LogLevel::MemTransfer` (C++ *enumerator*), 310

`vtkm::cont::LogLevel::Off` (C++ *enumerator*), 310

`vtkm::cont::LogLevel::Perf` (C++ *enumerator*), 310

`vtkm::cont::LogLevel::UserFirst` (C++ *enumerator*), 310

`vtkm::cont::LogLevel::UserLast` (C++ *enumerator*), 310

`vtkm::cont::LogLevel::UserVerboseFirst` (C++ *enumerator*), 311

`vtkm::cont::LogLevel::UserVerboseLast` (C++ *enumerator*), 311

`vtkm::cont::LogLevel::Warn` (C++ *enumerator*), 310

`vtkm::cont::make_ArrayHandle` (C++ *function*), 240, 241, 243

`vtkm::cont::make_ArrayHandleMove` (C++ *function*), 243

`vtkm::cont::make_ArrayHandleRuntimeVec` (C++ *function*), 420, 421

`vtkm::cont::make_ArrayHandleRuntimeVecMove` (C++ *function*), 420, 421

`vtkm::cont::make_ArrayHandleSOA` (C++ *function*), 412–415

`vtkm::cont::make_ArrayHandleSOAMove` (C++ *function*), 414

`vtkm::cont::PartitionedDataSet` (C++ *class*), 71

`vtkm::cont::PartitionedDataSet::AddField` (C++ *function*), 72

`vtkm::cont::PartitionedDataSet::AddGlobalField` (C++ *function*), 72

`vtkm::cont::PartitionedDataSet::AddPartitionsField` (C++ *function*), 72

`vtkm::cont::PartitionedDataSet::AppendPartitions` (C++ *function*), 72

`vtkm::cont::PartitionedDataSet::AppendPartitionsField` (C++ *function*), 72

`vtkm::cont::PartitionedDataSet::CopyPartitions` (C++ *function*), 73

`vtkm::cont::PartitionedDataSet::GetField` (C++ *function*), 73

`vtkm::cont::PartitionedDataSet::GetFieldFromPartition` (C++ *function*), 72

`vtkm::cont::PartitionedDataSet::GetGlobalField` (C++ *function*), 73

`vtkm::cont::PartitionedDataSet::GetGlobalNumberOfPartitions` (C++ *function*), 72

`vtkm::cont::PartitionedDataSet::GetNumberOfFields` (C++ *function*), 72

`vtkm::cont::PartitionedDataSet::GetNumberOfPartitions` (C++ *function*), 72

`vtkm::cont::PartitionedDataSet::GetPartition` (C++ *function*), 72

`vtkm::cont::PartitionedDataSet::GetPartitions` (C++ *function*), 72

`vtkm::cont::PartitionedDataSet::GetPartitionsField` (C++ *function*), 73

`vtkm::cont::PartitionedDataSet::HasField` (C++ *function*), 73

`vtkm::cont::PartitionedDataSet::HasGlobalField` (C++ *function*), 73

`vtkm::cont::PartitionedDataSet::HasPartitionsField` (C++ *function*), 73

`vtkm::cont::PartitionedDataSet::InsertPartition` (C++ *function*), 72

`vtkm::cont::PartitionedDataSet::PartitionedDataSet` (C++ *function*), 72

`vtkm::cont::PartitionedDataSet::ReplacePartition` (C++ *function*), 72

`vtkm::cont::RuntimeDeviceTracker` (C++ *class*), 210

`vtkm::cont::RuntimeDeviceTracker::CanRunOn` (C++ *function*), 210

`vtkm::cont::RuntimeDeviceTracker::ClearAbortChecker` (C++ *function*), 211

`vtkm::cont::RuntimeDeviceTracker::CopyStateFrom` (C++ *function*), 211

`vtkm::cont::RuntimeDeviceTracker::DisableDevice` (C++ *function*), 210

`vtkm::cont::RuntimeDeviceTracker::ForceDevice` (C++ *function*), 211

`vtkm::cont::RuntimeDeviceTracker::GetThreadFriendlyMemAlloc` (C++ *function*), 211

`vtkm::cont::RuntimeDeviceTracker::PrintSummary` (C++ *function*), 211

`vtkm::cont::RuntimeDeviceTracker::ReportAllocationFailure` (C++ *function*), 210

`vtkm::cont::RuntimeDeviceTracker::ReportBadDeviceFailure` (C++ *function*), 210

`vtkm::cont::RuntimeDeviceTracker::Reset` (C++ *function*), 210

`vtkm::cont::RuntimeDeviceTracker::ResetDevice` (C++ *function*), 210

`vtkm::cont::RuntimeDeviceTracker::SetAbortChecker` (C++ *function*), 211

`vtkm::cont::RuntimeDeviceTracker::SetThreadFriendlyMemAlloc` (C++ *function*), 211

```

    (C++ function), 211
vtkm::cont::RuntimeDeviceTrackerMode    (C++
    enum), 213
vtkm::cont::RuntimeDeviceTrackerMode::Disable
    (C++ enumerator), 213
vtkm::cont::RuntimeDeviceTrackerMode::Enable
    (C++ enumerator), 213
vtkm::cont::RuntimeDeviceTrackerMode::Force
    (C++ enumerator), 213
vtkm::cont::ScopedRuntimeDeviceTracker (C++
    class), 211
vtkm::cont::ScopedRuntimeDeviceTracker::~ScopedRuntimeDeviceTracker
    (C++ function), 212
vtkm::cont::ScopedRuntimeDeviceTracker::ScopedRuntimeDeviceTracker
    (C++ function), 212
vtkm::cont::SetLogLevelName (C++ function), 311
vtkm::cont::SetLogThreadName (C++ function), 309
vtkm::cont::SetStderrLogLevel (C++ function),
    311, 312
vtkm::cont::Timer (C++ class), 216
vtkm::cont::Timer::GetDevice (C++ function), 217
vtkm::cont::Timer::GetElapsedTime (C++ func-
    tion), 217
vtkm::cont::Timer::Ready (C++ function), 217
vtkm::cont::Timer::Reset (C++ function), 216
vtkm::cont::Timer::Start (C++ function), 216
vtkm::cont::Timer::Started (C++ function), 216
vtkm::cont::Timer::Stop (C++ function), 216
vtkm::cont::Timer::Stopped (C++ function), 217
vtkm::cont::Timer::Synchronize (C++ function),
    217
vtkm::cont::TypeToString (C++ function), 315
vtkm::CopyFlag (C++ enum), 241
vtkm::CopyFlag::Off (C++ enumerator), 241
vtkm::CopyFlag::On (C++ enumerator), 241
vtkm::CopySign (C++ function), 376
vtkm::Cos (C++ function), 379
vtkm::CosH (C++ function), 379, 380
vtkm::Cross (C++ function), 382
vtkm::Cylinder (C++ class), 223
vtkm::Cylinder::Cylinder (C++ function), 224
vtkm::Cylinder::Gradient (C++ function), 224
vtkm::Cylinder::SetAxis (C++ function), 224
vtkm::Cylinder::SetCenter (C++ function), 224
vtkm::Cylinder::SetRadius (C++ function), 224
vtkm::Cylinder::Value (C++ function), 224
vtkm::Epsilon (C++ function), 376
vtkm::Epsilon32 (C++ function), 376
vtkm::Epsilon64 (C++ function), 376
vtkm::ErrorCode (C++ enum), 306
vtkm::ErrorCode::CellNotFound (C++ enumera-
    tor), 307
vtkm::ErrorCode::DegenerateCellDetected
    (C++ enumerator), 307
vtkm::ErrorCode::InvalidCellMetric (C++ enu-
    merator), 306
vtkm::ErrorCode::InvalidEdgeId (C++ enumera-
    tor), 306
vtkm::ErrorCode::InvalidFaceId (C++ enumera-
    tor), 307
vtkm::ErrorCode::InvalidNumberOfPoints (C++
    enumerator), 306
vtkm::ErrorCode::InvalidPointId (C++ enumera-
    tor), 306
vtkm::ErrorCode::InvalidShapeId (C++ enumera-
    tor), 306
vtkm::ErrorCode::MalformedCellDetected (C++
    enumerator), 307
vtkm::ErrorCode::MatrixFactorizationFailed
    (C++ enumerator), 307
vtkm::ErrorCode::OperationOnEmptyCell (C++
    enumerator), 307
vtkm::ErrorCode::SolutionDidNotConverge
    (C++ enumerator), 307
vtkm::ErrorCode::Success (C++ enumerator), 306
vtkm::ErrorCode::UnknownError (C++ enumera-
    tor), 307
vtkm::ErrorCode::WrongShapeIdForTagType
    (C++ enumerator), 306
vtkm::ErrorString (C++ function), 307
vtkm::exec::BoundaryState (C++ struct), 340
vtkm::exec::BoundaryState::ClampNeighborIndex
    (C++ function), 342
vtkm::exec::BoundaryState::GetCenterIndex
    (C++ function), 341
vtkm::exec::BoundaryState::IJK (C++ member),
    343
vtkm::exec::BoundaryState::IsNeighborInBoundary
    (C++ function), 341
vtkm::exec::BoundaryState::IsNeighborInXBoundary
    (C++ function), 341
vtkm::exec::BoundaryState::IsNeighborInYBoundary
    (C++ function), 341
vtkm::exec::BoundaryState::IsNeighborInZBoundary
    (C++ function), 341
vtkm::exec::BoundaryState::IsRadiusInBoundary
    (C++ function), 341
vtkm::exec::BoundaryState::IsRadiusInXBoundary
    (C++ function), 340
vtkm::exec::BoundaryState::IsRadiusInYBoundary
    (C++ function), 340
vtkm::exec::BoundaryState::IsRadiusInZBoundary
    (C++ function), 341
vtkm::exec::BoundaryState::MaxNeighborIndices
    (C++ function), 342
vtkm::exec::BoundaryState::MinNeighborIndices
    (C++ function), 341
vtkm::exec::BoundaryState::NeighborIndexToFlatIndex

```

(C++ function), 343  
 vtkm::exec::BoundaryState::NeighborIndexToFlatIndexClass (C++ function), 94  
 (C++ function), 342, 343  
 vtkm::exec::BoundaryState::NeighborIndexToFullIndex (C++ function), 94  
 (C++ function), 342  
 vtkm::exec::BoundaryState::NeighborIndexToFullIndexClass (C++ function), 94  
 (C++ function), 342  
 vtkm::exec::BoundaryState::PointDimensions (C++ member), 343  
 vtkm::exec::CellEdgeCanonicalId (C++ function), 400  
 vtkm::exec::CellEdgeLocalIndex (C++ function), 400  
 vtkm::exec::CellEdgeNumberOfEdges (C++ function), 400  
 vtkm::exec::CellFaceCanonicalId (C++ function), 403  
 vtkm::exec::CellFaceLocalIndex (C++ function), 403  
 vtkm::exec::CellFaceNumberOfFaces (C++ function), 402  
 vtkm::exec::CellFaceNumberOfPoints (C++ function), 402  
 vtkm::exec::CellFaceShape (C++ function), 403  
 vtkm::exec::CellInterpolate (C++ function), 398  
 vtkm::exec::FieldNeighborhood (C++ struct), 338  
 vtkm::exec::FieldNeighborhood::Boundary (C++ member), 340  
 vtkm::exec::FieldNeighborhood::Get (C++ function), 339  
 vtkm::exec::FieldNeighborhood::GetUnchecked (C++ function), 339  
 vtkm::exec::FieldNeighborhood::Portal (C++ member), 340  
 vtkm::exec::ParametricCoordinatesCenter (C++ function), 396  
 vtkm::exec::ParametricCoordinatesPoint (C++ function), 396  
 vtkm::exec::ParametricCoordinatesToWorldCoordinates (C++ function), 396  
 vtkm::exec::WorldCoordinatesToParametricCoordinates (C++ function), 397  
 vtkm::Exp (C++ function), 369  
 vtkm::Exp10 (C++ function), 369, 370  
 vtkm::Exp2 (C++ function), 370  
 vtkm::ExpM1 (C++ function), 370  
 vtkm::filter  
   variable, 13, 14  
 vtkm::filter::clean\_grid::CleanGrid (C++ class), 93  
 vtkm::filter::clean\_grid::CleanGrid::GetCompactPointFields (C++ function), 94  
 vtkm::filter::clean\_grid::CleanGrid::GetFastMerge (C++ function), 95  
 vtkm::filter::clean\_grid::CleanGrid::GetMergePoints (C++ function), 94  
 vtkm::filter::clean\_grid::CleanGrid::GetRemoveDegenerateCells (C++ function), 94  
 vtkm::filter::clean\_grid::CleanGrid::GetTolerance (C++ function), 94  
 vtkm::filter::clean\_grid::CleanGrid::GetToleranceIsAbsolute (C++ function), 94  
 vtkm::filter::clean\_grid::CleanGrid::SetCompactPointFields (C++ function), 94  
 vtkm::filter::clean\_grid::CleanGrid::SetFastMerge (C++ function), 95  
 vtkm::filter::clean\_grid::CleanGrid::SetMergePoints (C++ function), 94  
 vtkm::filter::clean\_grid::CleanGrid::SetRemoveDegenerateCells (C++ function), 94  
 vtkm::filter::clean\_grid::CleanGrid::SetTolerance (C++ function), 94  
 vtkm::filter::clean\_grid::CleanGrid::SetToleranceIsAbsolute (C++ function), 94  
 vtkm::filter::connected\_components::CellSetConnectivity (C++ class), 95  
 vtkm::filter::connected\_components::ImageConnectivity (C++ class), 95  
 vtkm::filter::contour::AbstractContour::GetComputeFastNormals (C++ function), 97  
 vtkm::filter::contour::AbstractContour::GetGenerateNormals (C++ function), 97  
 vtkm::filter::contour::AbstractContour::GetIsoValue (C++ function), 96  
 vtkm::filter::contour::AbstractContour::GetMergeDuplicateFields (C++ function), 97  
 vtkm::filter::contour::AbstractContour::GetNormalArrayName (C++ function), 97  
 vtkm::filter::contour::AbstractContour::SetComputeFastNormals (C++ function), 97  
 vtkm::filter::contour::AbstractContour::SetGenerateNormals (C++ function), 96  
 vtkm::filter::contour::AbstractContour::SetIsoValue (C++ function), 96  
 vtkm::filter::contour::AbstractContour::SetIsoValues (C++ function), 96  
 vtkm::filter::contour::AbstractContour::SetMergeDuplicateFields (C++ function), 97  
 vtkm::filter::contour::AbstractContour::SetNormalArrayName (C++ function), 97  
 vtkm::filter::contour::ClipWithField (C++ class), 99  
 vtkm::filter::contour::ClipWithField::GetClipValue (C++ function), 99  
 vtkm::filter::contour::ClipWithField::GetInvertClip (C++ function), 99  
 vtkm::filter::contour::ClipWithField::SetClipValue (C++ function), 99



vtkm::filter::contour::ClipWithField::SetInvertClip: filter::density\_estimate::ParticleDensityCloudInCell  
 (C++ function), 99 (C++ class), 104  
 vtkm::filter::contour::ClipWithImplicitFunction: vtkm::filter::density\_estimate::ParticleDensityNearestGrid  
 (C++ class), 100 (C++ class), 103  
 vtkm::filter::contour::ClipWithImplicitFunction: vtkm::filter::density\_estimate::Statistics  
 (C++ function), 100 (C++ class), 105  
 vtkm::filter::contour::ClipWithImplicitFunction: vtkm::filter::entity\_extraction::ExternalFaces  
 (C++ function), 100 (C++ class), 106  
 vtkm::filter::contour::ClipWithImplicitFunction: vtkm::filter::entity\_extraction::ExternalFaces::CanThread  
 (C++ function), 100 (C++ function), 106  
 vtkm::filter::contour::Contour (C++ class), 96 vtkm::filter::entity\_extraction::ExternalFaces::GetCompact  
 vtkm::filter::contour::Slice (C++ class), 98 (C++ function), 106  
 vtkm::filter::contour::Slice::GetImplicitFunction: vtkm::filter::entity\_extraction::ExternalFaces::GetPassPol  
 (C++ function), 98 (C++ function), 106  
 vtkm::filter::contour::Slice::SetImplicitFunction: vtkm::filter::entity\_extraction::ExternalFaces::SetCompact  
 (C++ function), 98 (C++ function), 106  
 vtkm::filter::density\_estimate::Histogram vtkm::filter::entity\_extraction::ExternalFaces::SetPassPol  
 (C++ class), 101 (C++ function), 106  
 vtkm::filter::density\_estimate::Histogram::GetBinDef: vtkm::filter::entity\_extraction::ExtractGeometry  
 (C++ function), 102 (C++ class), 107  
 vtkm::filter::density\_estimate::Histogram::GetComputeRange: vtkm::filter::entity\_extraction::ExtractGeometry::ExtractE  
 (C++ function), 102 (C++ function), 108  
 vtkm::filter::density\_estimate::Histogram::GetNumberOfBins: vtkm::filter::entity\_extraction::ExtractGeometry::ExtractE  
 (C++ function), 102 (C++ function), 108  
 vtkm::filter::density\_estimate::Histogram::GetRange: vtkm::filter::entity\_extraction::ExtractGeometry::ExtractL  
 (C++ function), 102 (C++ function), 107  
 vtkm::filter::density\_estimate::Histogram::SetNumberOfBins: vtkm::filter::entity\_extraction::ExtractGeometry::ExtractL  
 (C++ function), 102 (C++ function), 107  
 vtkm::filter::density\_estimate::Histogram::SetRange: vtkm::filter::entity\_extraction::ExtractGeometry::ExtractC  
 (C++ function), 102 (C++ function), 108  
 vtkm::filter::density\_estimate::ParticleDensityBase: vtkm::filter::entity\_extraction::ExtractGeometry::ExtractC  
 (C++ class), 102 (C++ function), 108  
 vtkm::filter::density\_estimate::ParticleDensityBase::fGetComputeNumberDensity: vtkm::filter::entity\_extraction::ExtractGeometry::GetExtra  
 (C++ function), 102 (C++ function), 107  
 vtkm::filter::density\_estimate::ParticleDensityBase::fGetDimension: vtkm::filter::entity\_extraction::ExtractGeometry::GetExtra  
 (C++ function), 103 (C++ function), 107  
 vtkm::filter::density\_estimate::ParticleDensityBase::fGetDivideByVolume: vtkm::filter::entity\_extraction::ExtractGeometry::GetExtra  
 (C++ function), 103 (C++ function), 108  
 vtkm::filter::density\_estimate::ParticleDensityBase::fGetOrigin: vtkm::filter::entity\_extraction::ExtractGeometry::SetExtra  
 (C++ function), 103 (C++ function), 108  
 vtkm::filter::density\_estimate::ParticleDensityBase::fGetSpacing: vtkm::filter::entity\_extraction::ExtractGeometry::SetExtra  
 (C++ function), 103 (C++ function), 107  
 vtkm::filter::density\_estimate::ParticleDensityBase::fSetBounds: vtkm::filter::entity\_extraction::ExtractGeometry::SetExtra  
 (C++ function), 103 (C++ function), 108  
 vtkm::filter::density\_estimate::ParticleDensityBase::fSetComputeNumberDensity: vtkm::filter::entity\_extraction::ExtractGeometry::SetImpli  
 (C++ function), 102 (C++ function), 107  
 vtkm::filter::density\_estimate::ParticleDensityBase::fSetDimension: vtkm::filter::entity\_extraction::ExtractPoints  
 (C++ function), 103 (C++ class), 108  
 vtkm::filter::density\_estimate::ParticleDensityBase::fSetDivideByVolume: vtkm::filter::entity\_extraction::ExtractPoints::ExtractIns  
 (C++ function), 102 (C++ function), 109  
 vtkm::filter::density\_estimate::ParticleDensityBase::fSetOrigin: vtkm::filter::entity\_extraction::ExtractPoints::ExtractIns  
 (C++ function), 103 (C++ function), 109  
 vtkm::filter::density\_estimate::ParticleDensityBase::fSetSpacing: vtkm::filter::entity\_extraction::ExtractPoints::GetCompact  
 (C++ function), 103 (C++ function), 109

vtkm::filter::entity\_extraction::ExtractPoints::GetFilterInvert  
 (C++ function), 109

vtkm::filter::entity\_extraction::ExtractPoints::GetFilterInvert::Threshold::SetInvert  
 (C++ function), 114

vtkm::filter::entity\_extraction::ExtractPoints::SetCompactPointivity  
 (C++ function), 109

vtkm::filter::entity\_extraction::ExtractPoints::SetCompactPointivity::Threshold::SetLowerThreshold  
 (C++ function), 113

vtkm::filter::entity\_extraction::ExtractPoints::SetFilterInvert  
 (C++ function), 109

vtkm::filter::entity\_extraction::ExtractPoints::SetFilterInvert::Threshold::SetThresholdAbove  
 (C++ function), 113

vtkm::filter::entity\_extraction::ExtractPoints::SetImplicitFunction  
 (C++ function), 109

vtkm::filter::entity\_extraction::ExtractPoints::SetImplicitFunction::Threshold::SetThresholdBelow  
 (C++ function), 113

vtkm::filter::entity\_extraction::ExtractStructure::vtkm::filter::entity\_extraction::Threshold::SetThresholdBelow  
 (C++ class), 110

vtkm::filter::entity\_extraction::ExtractStructure::vtkm::filter::entity\_extraction::Threshold::SetUpperThreshold  
 (C++ function), 111

vtkm::filter::entity\_extraction::ExtractStructure::vtkm::filter::entity\_extraction::Threshold::SetUpperThreshold  
 (C++ function), 113

vtkm::filter::entity\_extraction::ExtractStructure::vtkm::filter::field\_conversion::CellAverage  
 (C++ function), 110

vtkm::filter::entity\_extraction::ExtractStructure::vtkm::filter::field\_conversion::CellAverage  
 (C++ class), 114

vtkm::filter::entity\_extraction::ExtractStructure::vtkm::filter::field\_conversion::PointAverage  
 (C++ function), 111

vtkm::filter::entity\_extraction::ExtractStructure::vtkm::filter::field\_conversion::PointAverage  
 (C++ class), 115

vtkm::filter::entity\_extraction::ExtractStructure::vtkm::filter::field\_transform::CompositeVectors  
 (C++ function), 110

vtkm::filter::entity\_extraction::ExtractStructure::vtkm::filter::field\_transform::CompositeVectors  
 (C++ class), 115

vtkm::filter::entity\_extraction::GhostCellRemoval::vtkm::filter::field\_transform::CompositeVectors::GetNumberOfFields  
 (C++ class), 111

vtkm::filter::entity\_extraction::GhostCellRemoval::vtkm::filter::field\_transform::CompositeVectors::SetFieldNames  
 (C++ function), 112

vtkm::filter::entity\_extraction::GhostCellRemoval::vtkm::filter::field\_transform::CompositeVectors::SetFieldNames  
 (C++ function), 115

vtkm::filter::entity\_extraction::GhostCellRemoval::vtkm::filter::field\_transform::CylindricalCoordinateTransform  
 (C++ function), 111

vtkm::filter::entity\_extraction::GhostCellRemoval::vtkm::filter::field\_transform::CylindricalCoordinateTransform  
 (C++ class), 116

vtkm::filter::entity\_extraction::GhostCellRemoval::vtkm::filter::field\_transform::CylindricalCoordinateTransform  
 (C++ function), 112

vtkm::filter::entity\_extraction::GhostCellRemoval::vtkm::filter::field\_transform::CylindricalCoordinateTransform  
 (C++ function), 116

vtkm::filter::entity\_extraction::GhostCellRemoval::vtkm::filter::field\_transform::CylindricalCoordinateTransform  
 (C++ function), 112

vtkm::filter::entity\_extraction::GhostCellRemoval::vtkm::filter::field\_transform::CylindricalCoordinateTransform  
 (C++ function), 116

vtkm::filter::entity\_extraction::GhostCellRemoval::vtkm::filter::field\_transform::FieldToColors  
 (C++ function), 111

vtkm::filter::entity\_extraction::GhostCellRemoval::vtkm::filter::field\_transform::FieldToColors  
 (C++ class), 116

vtkm::filter::entity\_extraction::GhostCellRemoval::vtkm::filter::field\_transform::FieldToColors::GetColorTable  
 (C++ function), 111

vtkm::filter::entity\_extraction::GhostCellRemoval::vtkm::filter::field\_transform::FieldToColors::GetColorTable  
 (C++ function), 117

vtkm::filter::entity\_extraction::GhostCellRemoval::vtkm::filter::field\_transform::FieldToColors::GetMappingColors  
 (C++ function), 112

vtkm::filter::entity\_extraction::GhostCellRemoval::vtkm::filter::field\_transform::FieldToColors::GetMappingColors  
 (C++ function), 118

vtkm::filter::entity\_extraction::GhostCellRemoval::vtkm::filter::field\_transform::FieldToColors::GetMappingColors  
 (C++ function), 112

vtkm::filter::entity\_extraction::GhostCellRemoval::vtkm::filter::field\_transform::FieldToColors::GetMappingColors  
 (C++ function), 117

vtkm::filter::entity\_extraction::Threshold::vtkm::filter::field\_transform::FieldToColors::GetNumberOfFields  
 (C++ class), 112

vtkm::filter::entity\_extraction::Threshold::vtkm::filter::field\_transform::FieldToColors::GetNumberOfFields  
 (C++ function), 118

vtkm::filter::entity\_extraction::Threshold::GetAllInRange  
 (C++ function), 114

vtkm::filter::entity\_extraction::Threshold::GetAllInRange::vtkm::filter::field\_transform::FieldToColors::GetOutputMode  
 (C++ function), 114

vtkm::filter::entity\_extraction::Threshold::GetAllInRange::vtkm::filter::field\_transform::FieldToColors::GetOutputMode  
 (C++ enum), 116

vtkm::filter::entity\_extraction::Threshold::GetLowerThreshold  
 (C++ function), 113

vtkm::filter::entity\_extraction::Threshold::GetLowerThreshold::vtkm::filter::field\_transform::FieldToColors::InputMode::C  
 (C++ function), 113

vtkm::filter::entity\_extraction::Threshold::GetLowerThreshold::vtkm::filter::field\_transform::FieldToColors::InputMode::C  
 (C++ enumerator), 117

vtkm::filter::entity\_extraction::Threshold::GetUpperThreshold  
 (C++ function), 113

vtkm::filter::entity\_extraction::Threshold::GetUpperThreshold::vtkm::filter::field\_transform::FieldToColors::InputMode::M  
 (C++ function), 113

vtkm::filter::entity\_extraction::Threshold::GetUpperThreshold::vtkm::filter::field\_transform::FieldToColors::InputMode::M  
 (C++ enumerator), 116

vtkm::filter::entity\_extraction::Threshold::SetAllInRange  
 (C++ function), 113

vtkm::filter::entity\_extraction::Threshold::SetAllInRange::vtkm::filter::field\_transform::FieldToColors::InputMode::S  
 (C++ function), 113

vtkm::filter::entity\_extraction::Threshold::SetAllInRange::vtkm::filter::field\_transform::FieldToColors::InputMode::S  
 (C++ enumerator), 116

vtkm::filter::entity\_extraction::Threshold::SetComponentToField  
 (C++ function), 113

vtkm::filter::entity\_extraction::Threshold::SetComponentToField::vtkm::filter::field\_transform::FieldToColors::IsMappingCom  
 (C++ function), 113

vtkm::filter::entity\_extraction::Threshold::SetComponentToField::vtkm::filter::field\_transform::FieldToColors::IsMappingCom  
 (C++ function), 118

vtkm::filter::entity\_extraction::Threshold::SetComponentToField::vtkm::filter::field\_transform::FieldToColors::IsMappingMag  
 (C++ function), 113

vtkm::filter::entity\_extraction::Threshold::SetComponentToField::vtkm::filter::field\_transform::FieldToColors::IsMappingMag  
 (C++ function), 118

vtkm::filter::entity\_extraction::Threshold::SetComponentToField::vtkm::filter::field\_transform::FieldToColors::IsMappingScal  
 (C++ function), 113

vtkm::filter::entity\_extraction::Threshold::SetComponentToField::vtkm::filter::field\_transform::FieldToColors::IsMappingScal  
 (C++ function), 118

vtkm::filter::field\_transform::FieldToColors::IsOutputRGB  
 (C++ function), 118

vtkm::filter::field\_transform::FieldToColors::IsOutputRGBA  
 (C++ function), 118

vtkm::filter::field\_transform::FieldToColors::OutputFilter  
 (C++ enum), 117

vtkm::filter::field\_transform::FieldToColors::OutputFilterRGB  
 (C++ enumerator), 117

vtkm::filter::field\_transform::FieldToColors::OutputFilterRGBA  
 (C++ enumerator), 117

vtkm::filter::field\_transform::FieldToColors::SetColorTable  
 (C++ function), 117

vtkm::filter::field\_transform::FieldToColors::SetMappingComponent  
 (C++ function), 118

vtkm::filter::field\_transform::FieldToColors::SetMappingMode  
 (C++ function), 117

vtkm::filter::field\_transform::FieldToColors::SetMappingToComponent  
 (C++ function), 117

vtkm::filter::field\_transform::FieldToColors::SetMappingToMagnitude  
 (C++ function), 117

vtkm::filter::field\_transform::FieldToColors::SetMappingToScalar  
 (C++ function), 117

vtkm::filter::field\_transform::FieldToColors::SetNumberOfSamplingPoints  
 (C++ function), 118

vtkm::filter::field\_transform::FieldToColors::SetOutputMode  
 (C++ function), 118

vtkm::filter::field\_transform::FieldToColors::SetOutputRGB  
 (C++ function), 118

vtkm::filter::field\_transform::FieldToColors::SetOutputRGBA  
 (C++ function), 118

vtkm::filter::field\_transform::GenerateIds  
 (C++ class), 119

vtkm::filter::field\_transform::GenerateIds::GetCellFileName  
 (C++ function), 119

vtkm::filter::field\_transform::GenerateIds::GetCellFileIds  
 (C++ function), 119

vtkm::filter::field\_transform::GenerateIds::GetCellPointIds  
 (C++ function), 119

vtkm::filter::field\_transform::GenerateIds::GetPointFileName  
 (C++ function), 119

vtkm::filter::field\_transform::GenerateIds::GetPointFileIds  
 (C++ function), 119

vtkm::filter::field\_transform::GenerateIds::SetCellFileName  
 (C++ function), 119

vtkm::filter::field\_transform::GenerateIds::SetCellFileIds  
 (C++ function), 119

vtkm::filter::field\_transform::GenerateIds::SetCellPointIds  
 (C++ function), 119

vtkm::filter::field\_transform::GenerateIds::SetPointFileName  
 (C++ function), 119

vtkm::filter::field\_transform::GenerateIds::SetPointFileIds  
 (C++ function), 120

vtkm::filter::field\_transform::LogValues  
 (C++ class), 120

vtkm::filter::field\_transform::LogValues::GetBaseValue  
 (C++ function), 121

vtkm::filter::field\_transform::LogValues::GetMinValue  
 (C++ function), 121

vtkm::filter::field\_transform::LogValues::LogBase  
 (C++ enum), 120

vtkm::filter::field\_transform::LogValues::LogBase::E  
 (C++ enumerator), 120

vtkm::filter::field\_transform::LogValues::LogBase::TEN  
 (C++ enumerator), 120

vtkm::filter::field\_transform::LogValues::LogBase::TWO  
 (C++ enumerator), 120

vtkm::filter::field\_transform::LogValues::SetBaseValue  
 (C++ function), 121

vtkm::filter::field\_transform::LogValues::SetBaseValueTo10  
 (C++ function), 121

vtkm::filter::field\_transform::LogValues::SetBaseValueTo2  
 (C++ function), 121

vtkm::filter::field\_transform::LogValues::SetBaseValueToE  
 (C++ function), 121

vtkm::filter::field\_transform::LogValues::SetMinValue  
 (C++ function), 121

vtkm::filter::field\_transform::LogValues::SetNumberOfSamplingPoints  
 (C++ class), 121

vtkm::filter::field\_transform::LogValues::SetHighPoint  
 (C++ function), 122

vtkm::filter::field\_transform::LogValues::SetLowPoint  
 (C++ function), 122

vtkm::filter::field\_transform::LogValues::SetRange  
 (C++ function), 122

vtkm::filter::field\_transform::PointTransform  
 (C++ class), 122

vtkm::filter::field\_transform::PointTransform::SetChangeColor  
 (C++ function), 124

vtkm::filter::field\_transform::PointTransform::SetRotation  
 (C++ function), 123

vtkm::filter::field\_transform::PointTransform::SetRotation  
 (C++ function), 123

vtkm::filter::field\_transform::PointTransform::SetRotation  
 (C++ function), 123

vtkm::filter::field\_transform::PointTransform::SetScale  
 (C++ function), 123

vtkm::filter::field\_transform::PointTransform::SetTransform  
 (C++ function), 123

vtkm::filter::field\_transform::PointTransform::SetTranslation  
 (C++ function), 123

vtkm::filter::field\_transform::SphericalCoordinateTransform  
 (C++ class), 124

vtkm::filter::field\_transform::SphericalCoordinateTransform::UseFilter  
 (C++ function), 124

vtkm::filter::field\_transform::SphericalCoordinateTransform  
 (C++ function), 124



vtkm::filter::field\_transform::Warp (C++ class), 124  
 vtkm::filter::field\_transform::Warp::GetChangeCoordinateSystem (C++ function), 126  
 vtkm::filter::field\_transform::Warp::GetConstantDirection (C++ function), 125  
 vtkm::filter::field\_transform::Warp::GetDirectionFieldName (C++ function), 125  
 vtkm::filter::field\_transform::Warp::GetScaleFactor (C++ function), 126  
 vtkm::filter::field\_transform::Warp::GetScaleFieldName (C++ function), 125  
 vtkm::filter::field\_transform::Warp::GetUseConstantDirection (C++ function), 125  
 vtkm::filter::field\_transform::Warp::GetUseScaleField (C++ function), 126  
 vtkm::filter::field\_transform::Warp::SetChangeCoordinateSystem (C++ function), 126  
 vtkm::filter::field\_transform::Warp::SetConstantDirection (C++ function), 125  
 vtkm::filter::field\_transform::Warp::SetDirectionFieldName (C++ function), 125  
 vtkm::filter::field\_transform::Warp::SetScaleFactor (C++ function), 126  
 vtkm::filter::field\_transform::Warp::SetScaleField (C++ function), 125  
 vtkm::filter::field\_transform::Warp::SetUseConstantDirection (C++ function), 125  
 vtkm::filter::field\_transform::Warp::SetUseScaleField (C++ function), 126  
 vtkm::filter::FieldSelection (C++ class), 88  
 vtkm::filter::FieldSelection::AddField (C++ function), 88  
 vtkm::filter::FieldSelection::ClearFields (C++ function), 90  
 vtkm::filter::FieldSelection::FieldSelection (C++ function), 89  
 vtkm::filter::FieldSelection::GetFieldMode (C++ function), 89  
 vtkm::filter::FieldSelection::GetMode (C++ function), 90  
 vtkm::filter::FieldSelection::HasField (C++ function), 90  
 vtkm::filter::FieldSelection::IsFieldSelected (C++ function), 90  
 vtkm::filter::FieldSelection::SetMode (C++ function), 90  
 vtkm::filter::Filter::CastAndCallScalarField (C++ function), 255, 256  
 vtkm::filter::Filter::CastAndCallVariableVecField (C++ function), 358  
 vtkm::filter::Filter::CastAndCallVecField (C++ function), 357, 358  
 vtkm::filter::Filter::CreateResult (C++ function), 257  
 vtkm::filter::Filter::CreateResultCoordinateSystem (C++ function), 259, 260  
 vtkm::filter::Filter::CreateResultField (C++ function), 258  
 vtkm::filter::Filter::CreateResultFieldCell (C++ function), 259  
 vtkm::filter::Filter::CreateResultFieldPoint (C++ function), 258  
 vtkm::filter::Filter::DoExecute (C++ function), 253  
 vtkm::filter::Filter::DoExecutePartitions (C++ function), 254  
 vtkm::filter::Filter::Execute (C++ function), 83  
 vtkm::filter::Filter::GetActiveCoordinateSystemIndex (C++ function), 85  
 vtkm::filter::Filter::GetActiveFieldAssociation (C++ function), 85  
 vtkm::filter::Filter::GetActiveFieldName (C++ function), 85  
 vtkm::filter::Filter::GetFieldFromDataSet (C++ function), 255  
 vtkm::filter::Filter::GetFieldsToPass (C++ function), 87  
 vtkm::filter::Filter::GetNumberOfActiveFields (C++ function), 86  
 vtkm::filter::Filter::GetOutputFieldName (C++ function), 92  
 vtkm::filter::Filter::GetPassCoordinateSystems (C++ function), 91  
 vtkm::filter::Filter::GetUseCoordinateSystemAsField (C++ function), 85  
 vtkm::filter::Filter::SetActiveCoordinateSystem (C++ function), 85  
 vtkm::filter::Filter::SetActiveField (C++ function), 85  
 vtkm::filter::Filter::SetFieldsToPass (C++ function), 86, 87, 90  
 vtkm::filter::Filter::SetOutputFieldName (C++ function), 92  
 vtkm::filter::Filter::SetPassCoordinateSystems (C++ function), 91  
 vtkm::filter::Filter::SetUseCoordinateSystemAsField (C++ function), 85  
 vtkm::filter::flow::FilterParticleAdvection (C++ class), 126  
 vtkm::filter::flow::FilterParticleAdvection::CanThread (C++ function), 127  
 vtkm::filter::flow::FilterParticleAdvection::SetNumberOfSteps (C++ function), 127  
 vtkm::filter::flow::FilterParticleAdvection::SetSeeds (C++ function), 127  
 vtkm::filter::flow::FilterParticleAdvection::SetStepSize (C++ function), 127

<code>vtkm::filter::flow::LagrangianStructures</code> (C++ class), 130	<code>vtkm::filter::geometry_refinement::Shrink::GetShrinkFactor</code> (C++ function), 134
<code>vtkm::filter::flow::LagrangianStructures::CanThread</code> (C++ function), 131	<code>vtkm::filter::geometry_refinement::Shrink::SetShrinkFactor</code> (C++ function), 134
<code>vtkm::filter::flow::LagrangianStructures::GetAdvectionTime</code> (C++ function), 131	<code>vtkm::filter::geometry_refinement::SplitSharpEdges</code> (C++ class), 134
<code>vtkm::filter::flow::LagrangianStructures::GetAuxiliaryGridDimensions</code> (C++ function), 132	<code>vtkm::filter::geometry_refinement::SplitSharpEdges::GetFeature</code> (C++ function), 134
<code>vtkm::filter::flow::LagrangianStructures::GetFlowMapOutput</code> (C++ function), 132	<code>vtkm::filter::geometry_refinement::SplitSharpEdges::SetFeature</code> (C++ function), 134
<code>vtkm::filter::flow::LagrangianStructures::GetNumberOfSteps</code> (C++ function), 131	<code>vtkm::filter::geometry_refinement::Tetrahedralize</code> (C++ class), 135
<code>vtkm::filter::flow::LagrangianStructures::GetOutputFieldName</code> (C++ function), 132	<code>vtkm::filter::geometry_refinement::Triangulate</code> (C++ class), 135
<code>vtkm::filter::flow::LagrangianStructures::GetStepSize</code> (C++ function), 131	<code>vtkm::filter::geometry_refinement::Tube</code> (C++ class), 135
<code>vtkm::filter::flow::LagrangianStructures::GetUseAuxiliaryGrid</code> (C++ function), 131	<code>vtkm::filter::geometry_refinement::Tube::SetCapping</code> (C++ function), 136
<code>vtkm::filter::flow::LagrangianStructures::GetUseFlowMapOutput</code> (C++ function), 132	<code>vtkm::filter::geometry_refinement::Tube::SetNumberOfSides</code> (C++ function), 136
<code>vtkm::filter::flow::LagrangianStructures::SetAdvectionTime</code> (C++ function), 131	<code>vtkm::filter::geometry_refinement::Tube::SetRadius</code> (C++ function), 136
<code>vtkm::filter::flow::LagrangianStructures::SetAuxiliaryGridDimensions</code> (C++ function), 131	<code>vtkm::filter::geometry_refinement::VertexClustering</code> (C++ class), 136
<code>vtkm::filter::flow::LagrangianStructures::SetFlowMapOutput</code> (C++ function), 132	<code>vtkm::filter::geometry_refinement::VertexClustering::GetNum</code> (C++ function), 137
<code>vtkm::filter::flow::LagrangianStructures::SetNumberOfSteps</code> (C++ function), 131	<code>vtkm::filter::geometry_refinement::VertexClustering::SetNum</code> (C++ function), 137
<code>vtkm::filter::flow::LagrangianStructures::SetOutputFieldName</code> (C++ function), 132	<code>vtkm::filter::MapFieldMergeAverage</code> (C++ func-
<code>vtkm::filter::flow::LagrangianStructures::SetStepSize</code> (C++ function), 131	<code>tion</code> ), 363, 364
<code>vtkm::filter::flow::LagrangianStructures::SetUseAuxiliaryGrid</code> (C++ function), 131	<code>vtkm::filter::Mesh_info::CellMeasures</code> (C++
<code>vtkm::filter::flow::LagrangianStructures::SetUseFlowMapOutput</code> (C++ function), 132	<code>class</code> ), 137
<code>vtkm::filter::flow::Pathline</code> (C++ class), 128	<code>vtkm::filter::Mesh_info::CellMeasures::GetCellMeasureName</code> (C++ function), 138
<code>vtkm::filter::flow::Streamline</code> (C++ class), 127	<code>vtkm::filter::Mesh_info::CellMeasures::GetMeasure</code> (C++ function), 137
<code>vtkm::filter::flow::StreamSurface</code> (C++ class), 129	<code>vtkm::filter::Mesh_info::CellMeasures::SetCellMeasureName</code> (C++ function), 138
<code>vtkm::filter::flow::StreamSurface::SetNumberOfSteps</code> (C++ function), 129	<code>vtkm::filter::Mesh_info::CellMeasures::SetMeasure</code> (C++ function), 137
<code>vtkm::filter::flow::StreamSurface::SetSeeds</code> (C++ function), 129, 130	<code>vtkm::filter::Mesh_info::CellMeasures::SetMeasureToAll</code> (C++ function), 138
<code>vtkm::filter::flow::StreamSurface::SetStepSize</code> (C++ function), 129	<code>vtkm::filter::Mesh_info::CellMeasures::SetMeasureToArcLength</code> (C++ function), 138
<code>vtkm::filter::geometry_refinement::ConvertToPointCloud</code> (C++ class), 133	<code>vtkm::filter::Mesh_info::CellMeasures::SetMeasureToArea</code> (C++ function), 138
<code>vtkm::filter::geometry_refinement::ConvertToPointCloud</code> (C++ function), 133	<code>vtkm::filter::Mesh_info::CellMeasures::SetMeasureToVolume</code> (C++ function), 138
<code>vtkm::filter::geometry_refinement::ConvertToPointCloud</code> (C++ function), 133	<code>vtkm::filter::Mesh_info::CellMetric</code> (C++
<code>vtkm::filter::geometry_refinement::Shrink</code> (C++ class), 133	<code>enum</code> ), 143
	<code>vtkm::filter::Mesh_info::CellMetric::Area</code> (C++ enumerator), 143

vtkm::filter::mesh_info::CellMetric::AspectGamma (C++ enumerator), 144	vtkm::filter::mesh_info::IntegrationType::ArcLength (C++ enumerator), 138
vtkm::filter::mesh_info::CellMetric::AspectRatio (C++ enumerator), 144	vtkm::filter::mesh_info::IntegrationType::Area (C++ enumerator), 138
vtkm::filter::mesh_info::CellMetric::Condition (C++ enumerator), 144	vtkm::filter::mesh_info::IntegrationType::None (C++ enumerator), 138
vtkm::filter::mesh_info::CellMetric::DiagonalRatio (C++ enumerator), 144	vtkm::filter::mesh_info::IntegrationType::Volume (C++ enumerator), 138
vtkm::filter::mesh_info::CellMetric::Dimension (C++ enumerator), 144	vtkm::filter::mesh_info::MeshQuality (C++ class), 143
vtkm::filter::mesh_info::CellMetric::Jacobian (C++ enumerator), 144	vtkm::filter::mesh_info::MeshQuality::GetMetric (C++ function), 143
vtkm::filter::mesh_info::CellMetric::MaxAngle (C++ enumerator), 144	vtkm::filter::mesh_info::MeshQuality::GetMetricName (C++ function), 143
vtkm::filter::mesh_info::CellMetric::MaxDiagonalRatio (C++ enumerator), 144	vtkm::filter::mesh_info::MeshQuality::SetMetric (C++ function), 143
vtkm::filter::mesh_info::CellMetric::MinAngle (C++ enumerator), 145	vtkm::filter::mesh_info::MeshQualityArea (C++ class), 139
vtkm::filter::mesh_info::CellMetric::MinDiagonalRatio (C++ enumerator), 145	vtkm::filter::mesh_info::MeshQualityArea::ComputeAverageArea (C++ function), 139
vtkm::filter::mesh_info::CellMetric::None (C++ enumerator), 147	vtkm::filter::mesh_info::MeshQualityArea::ComputeTotalArea (C++ function), 139
vtkm::filter::mesh_info::CellMetric::Oddy (C++ enumerator), 145	vtkm::filter::mesh_info::MeshQualityAspectGamma (C++ class), 139
vtkm::filter::mesh_info::CellMetric::RelativeSizeSquared (C++ enumerator), 145	vtkm::filter::mesh_info::MeshQualityAspectRatio (C++ class), 140
vtkm::filter::mesh_info::CellMetric::ScaledJacobian (C++ enumerator), 145	vtkm::filter::mesh_info::MeshQualityCondition (C++ class), 140
vtkm::filter::mesh_info::CellMetric::Shape (C++ enumerator), 146	vtkm::filter::mesh_info::MeshQualityDiagonalRatio (C++ class), 140
vtkm::filter::mesh_info::CellMetric::ShapeAndSize (C++ enumerator), 146	vtkm::filter::mesh_info::MeshQualityDimension (C++ class), 140
vtkm::filter::mesh_info::CellMetric::Shear (C++ enumerator), 146	vtkm::filter::mesh_info::MeshQualityJacobian (C++ class), 140
vtkm::filter::mesh_info::CellMetric::Skew (C++ enumerator), 146	vtkm::filter::mesh_info::MeshQualityMaxAngle (C++ class), 140
vtkm::filter::mesh_info::CellMetric::Stretch (C++ enumerator), 146	vtkm::filter::mesh_info::MeshQualityMaxDiagonal (C++ class), 140
vtkm::filter::mesh_info::CellMetric::Taper (C++ enumerator), 146	vtkm::filter::mesh_info::MeshQualityMinAngle (C++ class), 140
vtkm::filter::mesh_info::CellMetric::Volume (C++ enumerator), 147	vtkm::filter::mesh_info::MeshQualityMinDiagonal (C++ class), 141
vtkm::filter::mesh_info::CellMetric::Warpage (C++ enumerator), 147	vtkm::filter::mesh_info::MeshQualityOddy (C++ class), 141
vtkm::filter::mesh_info::GhostCellClassify (C++ class), 139	vtkm::filter::mesh_info::MeshQualityRelativeSizeSquared (C++ class), 141
vtkm::filter::mesh_info::GhostCellClassify::GetGhostCellName (C++ function), 139	vtkm::filter::mesh_info::MeshQualityScaledJacobian (C++ class), 141
vtkm::filter::mesh_info::GhostCellClassify::SetGhostCellName (C++ function), 139	vtkm::filter::mesh_info::MeshQualityShape (C++ class), 141
vtkm::filter::mesh_info::IntegrationType (C++ enum), 138	vtkm::filter::mesh_info::MeshQualityShapeAndSize (C++ class), 142
vtkm::filter::mesh_info::IntegrationType::AllMeasures (C++ enumerator), 138	vtkm::filter::mesh_info::MeshQualityShear (C++ class), 142

<code>vtkm::filter::mesh_info::MeshQualitySkew</code> (C++ class), 142	<code>vtkm::filter::vector_analysis::CrossProduct::GetSecondaryField</code> (C++ function), 152
<code>vtkm::filter::mesh_info::MeshQualityStretch</code> (C++ class), 142	<code>vtkm::filter::vector_analysis::CrossProduct::GetSecondaryField</code> (C++ function), 152
<code>vtkm::filter::mesh_info::MeshQualityTaper</code> (C++ class), 142	<code>vtkm::filter::vector_analysis::CrossProduct::GetUseCoordinates</code> (C++ function), 152
<code>vtkm::filter::mesh_info::MeshQualityVolume</code> (C++ class), 142	<code>vtkm::filter::vector_analysis::CrossProduct::GetUseCoordinates</code> (C++ function), 153
<code>vtkm::filter::mesh_info::MeshQualityVolume::ComputeAverageVolume</code> (C++ function), 143	<code>vtkm::filter::vector_analysis::CrossProduct::SetPrimaryCoordinates</code> (C++ function), 152
<code>vtkm::filter::mesh_info::MeshQualityVolume::ComputeFieldVolume</code> (C++ function), 143	<code>vtkm::filter::vector_analysis::CrossProduct::SetPrimaryField</code> (C++ function), 151
<code>vtkm::filter::mesh_info::MeshQualityWarpage</code> (C++ class), 143	<code>vtkm::filter::vector_analysis::CrossProduct::SetSecondaryCoordinates</code> (C++ function), 153
<code>vtkm::filter::multi_block::AmrArrays</code> (C++ class), 147	<code>vtkm::filter::vector_analysis::CrossProduct::SetSecondaryField</code> (C++ function), 152
<code>vtkm::filter::multi_block::MergeDataSets</code> (C++ class), 148	<code>vtkm::filter::vector_analysis::CrossProduct::SetUseCoordinates</code> (C++ function), 152
<code>vtkm::filter::multi_block::MergeDataSets::GetInvalidValues</code> (C++ function), 149	<code>vtkm::filter::vector_analysis::CrossProduct::SetUseCoordinates</code> (C++ function), 153
<code>vtkm::filter::multi_block::MergeDataSets::SetInvalidValues</code> (C++ function), 149	<code>vtkm::filter::vector_analysis::DotProduct</code> (C++ class), 153
<code>vtkm::filter::resampling::HistSampling</code> (C++ class), 149	<code>vtkm::filter::vector_analysis::DotProduct::GetPrimaryCoordinates</code> (C++ function), 154
<code>vtkm::filter::resampling::HistSampling::GetNumberOfBins</code> (C++ function), 150	<code>vtkm::filter::vector_analysis::DotProduct::GetPrimaryField</code> (C++ function), 154
<code>vtkm::filter::resampling::HistSampling::GetSampleFraction</code> (C++ function), 150	<code>vtkm::filter::vector_analysis::DotProduct::GetPrimaryField</code> (C++ function), 153
<code>vtkm::filter::resampling::HistSampling::GetSeed</code> (C++ function), 150	<code>vtkm::filter::vector_analysis::DotProduct::GetSecondaryCoordinates</code> (C++ function), 155
<code>vtkm::filter::resampling::HistSampling::SetNumberOfBins</code> (C++ function), 150	<code>vtkm::filter::vector_analysis::DotProduct::GetSecondaryField</code> (C++ function), 155
<code>vtkm::filter::resampling::HistSampling::SetSampleFraction</code> (C++ function), 150	<code>vtkm::filter::vector_analysis::DotProduct::GetSecondaryField</code> (C++ function), 154
<code>vtkm::filter::resampling::HistSampling::SetSeed</code> (C++ function), 150	<code>vtkm::filter::vector_analysis::DotProduct::GetUseCoordinates</code> (C++ function), 154
<code>vtkm::filter::resampling::Probe</code> (C++ class), 150	<code>vtkm::filter::vector_analysis::DotProduct::GetUseCoordinates</code> (C++ function), 155
<code>vtkm::filter::resampling::Probe::GetGeometry</code> (C++ function), 151	<code>vtkm::filter::vector_analysis::DotProduct::SetPrimaryCoordinates</code> (C++ function), 154
<code>vtkm::filter::resampling::Probe::GetInvalidValues</code> (C++ function), 151	<code>vtkm::filter::vector_analysis::DotProduct::SetPrimaryField</code> (C++ function), 153
<code>vtkm::filter::resampling::Probe::SetGeometry</code> (C++ function), 151	<code>vtkm::filter::vector_analysis::DotProduct::SetSecondaryCoordinates</code> (C++ function), 155
<code>vtkm::filter::resampling::Probe::SetInvalidValues</code> (C++ function), 151	<code>vtkm::filter::vector_analysis::DotProduct::SetSecondaryField</code> (C++ function), 154
<code>vtkm::filter::vector_analysis::CrossProduct</code> (C++ class), 151	<code>vtkm::filter::vector_analysis::DotProduct::SetUseCoordinates</code> (C++ function), 154
<code>vtkm::filter::vector_analysis::CrossProduct::GetPrimaryFieldAssociation</code> (C++ function), 152	<code>vtkm::filter::vector_analysis::DotProduct::SetUseCoordinates</code> (C++ function), 155
<code>vtkm::filter::vector_analysis::CrossProduct::GetPrimaryFieldName</code> (C++ function), 151	<code>vtkm::filter::vector_analysis::Gradient</code> (C++ class), 155
<code>vtkm::filter::vector_analysis::CrossProduct::GetSecondaryCoordinates</code> (C++ function), 153	<code>vtkm::filter::vector_analysis::Gradient::GetComputedDivergence</code> (C++ function), 156

vtkm::filter::vector\_analysis::Gradient::GetComputeGradient:vector\_analysis::SurfaceNormals::SetCellNormals  
 (C++ function), 157 (C++ function), 158  
 vtkm::filter::vector\_analysis::Gradient::GetComputeFilterGradient:vector\_analysis::SurfaceNormals::SetConsistency  
 (C++ function), 156 (C++ function), 159  
 vtkm::filter::vector\_analysis::Gradient::GetComputeFilterCriterion:vector\_analysis::SurfaceNormals::SetFlipNormals  
 (C++ function), 156 (C++ function), 159  
 vtkm::filter::vector\_analysis::Gradient::GetComputeFilterCityvector\_analysis::SurfaceNormals::SetGenerateNormals  
 (C++ function), 156 (C++ function), 158  
 vtkm::filter::vector\_analysis::Gradient::GetDivergenceName:vector\_analysis::SurfaceNormals::SetGenerateNormals  
 (C++ function), 156 (C++ function), 158  
 vtkm::filter::vector\_analysis::Gradient::GetQCriteriumName:vector\_analysis::SurfaceNormals::SetNormalization  
 (C++ function), 157 (C++ function), 158  
 vtkm::filter::vector\_analysis::Gradient::GetVolumetricName:vector\_analysis::SurfaceNormals::SetPointNormals  
 (C++ function), 156 (C++ function), 158  
 vtkm::filter::vector\_analysis::Gradient::SetColumnMajorOrdering:vector\_analysis::SurfaceNormals::SurfaceNormals  
 (C++ function), 157 (C++ function), 158  
 vtkm::filter::vector\_analysis::Gradient::SetComputeDivergence:vector\_analysis::VectorMagnitude  
 (C++ function), 156 (C++ class), 159  
 vtkm::filter::vector\_analysis::Gradient::SetComputeGradient:zfp::ZFPCompressor1D (C++  
 (C++ function), 157 class), 160  
 vtkm::filter::vector\_analysis::Gradient::SetComputeFilterGradient:zfp::ZFPCompressor1D::GetRate  
 (C++ function), 156 (C++ function), 160  
 vtkm::filter::vector\_analysis::Gradient::SetComputeFilterCriterion:zfp::ZFPCompressor1D::SetRate  
 (C++ function), 156 (C++ function), 160  
 vtkm::filter::vector\_analysis::Gradient::SetComputeFilterCityzfp::ZFPCompressor2D (C++  
 (C++ function), 156 class), 160  
 vtkm::filter::vector\_analysis::Gradient::SetDivergenceName:zfp::ZFPCompressor2D::GetRate  
 (C++ function), 156 (C++ function), 160  
 vtkm::filter::vector\_analysis::Gradient::SetQCriteriumName:zfp::ZFPCompressor2D::SetRate  
 (C++ function), 157 (C++ function), 160  
 vtkm::filter::vector\_analysis::Gradient::SetRowMajorOrdering:zfp::ZFPCompressor3D (C++  
 (C++ function), 157 class), 160  
 vtkm::filter::vector\_analysis::Gradient::SetVolumetricName:zfp::ZFPCompressor3D::GetRate  
 (C++ function), 156 (C++ function), 161  
 vtkm::filter::vector\_analysis::SurfaceNormals vtkm::filter::zfp::ZFPCompressor3D::SetRate  
 (C++ class), 157 (C++ function), 161  
 vtkm::filter::vector\_analysis::SurfaceNormals::vGetAutoOrientNormalZFPDecompressor1D (C++  
 (C++ function), 159 class), 161  
 vtkm::filter::vector\_analysis::SurfaceNormals::vGetCellNormalsNameZFPDecompressor1D::GetRate  
 (C++ function), 158 (C++ function), 161  
 vtkm::filter::vector\_analysis::SurfaceNormals::vGetConsistencyzfp::ZFPCompressor1D::SetRate  
 (C++ function), 159 (C++ function), 161  
 vtkm::filter::vector\_analysis::SurfaceNormals::vGetFlipNormalszfp::ZFPCompressor2D (C++  
 (C++ function), 159 class), 161  
 vtkm::filter::vector\_analysis::SurfaceNormals::vGetGenerateCellNormalszfp::ZFPCompressor2D::GetRate  
 (C++ function), 158 (C++ function), 161  
 vtkm::filter::vector\_analysis::SurfaceNormals::vGetGeneratePointNormalszfp::ZFPCompressor2D::SetRate  
 (C++ function), 158 (C++ function), 161  
 vtkm::filter::vector\_analysis::SurfaceNormals::vGetNormalizeCellNormalszfp::ZFPCompressor3D (C++  
 (C++ function), 158 class), 161  
 vtkm::filter::vector\_analysis::SurfaceNormals::vGetPointNormalsNameZFPDecompressor3D::GetRate  
 (C++ function), 159 (C++ function), 162  
 vtkm::filter::vector\_analysis::SurfaceNormals::vGetAutoOrientNormalZFPDecompressor3D::SetRate  
 (C++ function), 159 (C++ function), 162



```

vtkm::filter_contour
    variable, 14
vtkm::filter_field_transform
    variable, 14
vtkm::Float32 (C++ type), 25
vtkm::Float64 (C++ type), 25
vtkm::FloatDefault (C++ type), 25
vtkm::FloatDistance (C++ function), 381
vtkm::Floor (C++ function), 377
vtkm::FMod (C++ function), 376
vtkm::ForEach (C++ function), 302
vtkm::Frustum (C++ class), 226
vtkm::Get (C++ function), 301
vtkm::get (C++ function), 301
vtkm::Id (C++ type), 26
vtkm::Id2 (C++ type), 28
vtkm::Id3 (C++ type), 28
vtkm::Id4 (C++ type), 29
vtkm::IdComponent (C++ type), 26
vtkm::IdComponent2 (C++ type), 29
vtkm::IdComponent3 (C++ type), 29
vtkm::IdComponent4 (C++ type), 29
vtkm::ImplicitFunctionGeneral (C++ class), 227
vtkm::Infinity (C++ function), 372
vtkm::Infinity32 (C++ function), 372
vtkm::Infinity64 (C++ function), 372
vtkm::Int16 (C++ type), 26
vtkm::Int32 (C++ type), 27
vtkm::Int64 (C++ type), 27
vtkm::Int8 (C++ type), 26
vtkm::io
    variable, 14
vtkm::io::ErrorIO (C++ class), 204
vtkm::io::FileType (C++ enum), 80
vtkm::io::FileType::ASCII (C++ enumerator), 80
vtkm::io::FileType::BINARY (C++ enumerator), 80
vtkm::io::ImageReaderPNG (C++ class), 78
vtkm::io::ImageReaderPNM (C++ class), 78
vtkm::io::ImageWriterBase::PixelDepth (C++
    enum), 80
vtkm::io::ImageWriterBase::PixelDepth::PIXEL_16 (C++
    enumerator), 80
vtkm::io::ImageWriterBase::PixelDepth::PIXEL_8 (C++
    enumerator), 80
vtkm::io::ImageWriterPNG (C++ class), 80
vtkm::io::ImageWriterPNM (C++ class), 81
vtkm::io::ImageWriterPNM::Write (C++ function),
    81
vtkm::io::VTKDataSetReader (C++ class), 77
vtkm::io::VTKDataSetReader::VTKDataSetReader
    (C++ function), 78
vtkm::io::VTKDataSetWriter (C++ class), 79
vtkm::io::VTKDataSetWriter::GetFileType
    (C++ function), 79
vtkm::io::VTKDataSetWriter::SetFileType
    (C++ function), 79
vtkm::io::VTKDataSetWriter::SetFileTypeToAscii
    (C++ function), 79
vtkm::io::VTKDataSetWriter::SetFileTypeToBinary
    (C++ function), 80
vtkm::io::VTKDataSetWriter::VTKDataSetWriter
    (C++ function), 79
vtkm::io::VTKDataSetWriter::WriteDataSet
    (C++ function), 79
vtkm::IsFinite (C++ function), 372
vtkm::IsInf (C++ function), 372
vtkm::IsNan (C++ function), 372
vtkm::IsNegative (C++ function), 372
vtkm::Lerp (C++ function), 382
vtkm::List (C++ struct), 288
vtkm::ListAppend (C++ type), 293
vtkm::ListApply (C++ type), 294
vtkm::ListAt (C++ type), 292
vtkm::ListCross (C++ type), 296
vtkm::ListEmpty (C++ type), 288
vtkm::ListForEach (C++ function), 296
vtkm::ListHas (C++ type), 292
vtkm::ListIndexOf (C++ type), 292
vtkm::ListIntersect (C++ type), 294
vtkm::ListRemoveIf (C++ type), 295
vtkm::ListSize (C++ type), 291
vtkm::ListTransform (C++ type), 295
vtkm::ListUniversal (C++ type), 288
vtkm::Log (C++ function), 370
vtkm::Log10 (C++ function), 370, 371
vtkm::Log1P (C++ function), 371
vtkm::Log2 (C++ function), 371
vtkm::Magnitude (C++ function), 382
vtkm::MagnitudeSquared (C++ function), 382
vtkm::make_Pair (C++ function), 299
vtkm::make_tuple (C++ function), 300
vtkm::make_Vec (C++ function), 266
vtkm::make_VecC (C++ function), 268
vtkm::MakeTuple (C++ function), 300
vtkm::Matrix (C++ class), 384
vtkm::Matrix::Matrix (C++ function), 384
vtkm::Matrix::operator() (C++ function), 384
vtkm::Matrix::operator[] (C++ function), 384
vtkm::MatrixDeterminant (C++ function), 385
vtkm::MatrixGetColumn (C++ function), 385
vtkm::MatrixGetRow (C++ function), 385
vtkm::MatrixIdentity (C++ function), 385
vtkm::MatrixInverse (C++ function), 385
vtkm::MatrixMultiply (C++ function), 385, 386
vtkm::MatrixSetColumn (C++ function), 386
vtkm::MatrixSetRow (C++ function), 386
vtkm::MatrixTranspose (C++ function), 386
vtkm::Max (C++ function), 381

```

vtkm::Min (C++ function), 382  
 vtkm::ModF (C++ function), 375  
 vtkm::Nan (C++ function), 372  
 vtkm::Nan32 (C++ function), 372  
 vtkm::Nan64 (C++ function), 372  
 vtkm::NegativeInfinity (C++ function), 373  
 vtkm::NegativeInfinity32 (C++ function), 373  
 vtkm::NegativeInfinity64 (C++ function), 373  
 vtkm::NewtonsMethod (C++ function), 386  
 vtkm::NewtonsMethodResult (C++ struct), 387  
 vtkm::NewtonsMethodResult::Converged (C++ member), 387  
 vtkm::NewtonsMethodResult::Solution (C++ member), 387  
 vtkm::NewtonsMethodResult::Valid (C++ member), 387  
 vtkm::Normal (C++ function), 382  
 vtkm::Normalize (C++ function), 382  
 vtkm::Orthonormalize (C++ function), 382  
 vtkm::Pair (C++ struct), 298  
 vtkm::Pair::first (C++ member), 299  
 vtkm::Pair::first\_type (C++ type), 298  
 vtkm::Pair::FirstType (C++ type), 298  
 vtkm::Pair::operator!= (C++ function), 299  
 vtkm::Pair::operator= (C++ function), 299  
 vtkm::Pair::operator== (C++ function), 299  
 vtkm::Pair::operator> (C++ function), 299  
 vtkm::Pair::operator>= (C++ function), 299  
 vtkm::Pair::operator< (C++ function), 299  
 vtkm::Pair::operator<= (C++ function), 299  
 vtkm::Pair::Pair (C++ function), 298, 299  
 vtkm::Pair::second (C++ member), 299  
 vtkm::Pair::second\_type (C++ type), 298  
 vtkm::Pair::SecondType (C++ type), 298  
 vtkm::Pi (C++ function), 380  
 vtkm::Pi\_180 (C++ function), 380  
 vtkm::Pi\_2 (C++ function), 380  
 vtkm::Pi\_3 (C++ function), 380  
 vtkm::Pi\_4 (C++ function), 380  
 vtkm::Plane (C++ class), 219  
 vtkm::Plane::ClosestPoint (C++ function), 220  
 vtkm::Plane::DistanceTo (C++ function), 220  
 vtkm::Plane::GetNormal (C++ function), 221  
 vtkm::Plane::GetOrigin (C++ function), 221  
 vtkm::Plane::Gradient (C++ function), 221  
 vtkm::Plane::Intersect (C++ function), 220, 221  
 vtkm::Plane::IsValid (C++ function), 220  
 vtkm::Plane::Plane (C++ function), 220, 221  
 vtkm::Plane::SetNormal (C++ function), 221  
 vtkm::Plane::SetOrigin (C++ function), 221  
 vtkm::Plane::Value (C++ function), 221  
 vtkm::Pow (C++ function), 371  
 vtkm::Project (C++ function), 383  
 vtkm::ProjectedDistance (C++ function), 383  
 vtkm::QuadraticRoots (C++ function), 373  
 vtkm::Range (C++ struct), 271  
 vtkm::Range::Center (C++ function), 271  
 vtkm::Range::Contains (C++ function), 271  
 vtkm::Range::Include (C++ function), 271  
 vtkm::Range::Intersection (C++ function), 272  
 vtkm::Range::IsEmpty (C++ function), 271  
 vtkm::Range::Length (C++ function), 271  
 vtkm::Range::Max (C++ member), 272  
 vtkm::Range::Min (C++ member), 272  
 vtkm::Range::operator+ (C++ function), 272  
 vtkm::Range::Range (C++ function), 271  
 vtkm::Range::Union (C++ function), 272  
 vtkm::RangeId (C++ struct), 276  
 vtkm::RangeId2 (C++ struct), 277  
 vtkm::RangeId2::Center (C++ function), 277  
 vtkm::RangeId2::Contains (C++ function), 277  
 vtkm::RangeId2::Include (C++ function), 278  
 vtkm::RangeId2::IsEmpty (C++ function), 277  
 vtkm::RangeId2::operator+ (C++ function), 278  
 vtkm::RangeId2::RangeId2 (C++ function), 277  
 vtkm::RangeId2::Union (C++ function), 278  
 vtkm::RangeId2::X (C++ member), 278  
 vtkm::RangeId2::Y (C++ member), 278  
 vtkm::RangeId3 (C++ struct), 278  
 vtkm::RangeId3::Center (C++ function), 279  
 vtkm::RangeId3::Contains (C++ function), 279  
 vtkm::RangeId3::Include (C++ function), 279  
 vtkm::RangeId3::IsEmpty (C++ function), 279  
 vtkm::RangeId3::operator+ (C++ function), 279  
 vtkm::RangeId3::RangeId3 (C++ function), 279  
 vtkm::RangeId3::Union (C++ function), 279  
 vtkm::RangeId3::X (C++ member), 280  
 vtkm::RangeId3::Y (C++ member), 280  
 vtkm::RangeId3::Z (C++ member), 280  
 vtkm::RangeId::Center (C++ function), 276  
 vtkm::RangeId::Contains (C++ function), 276  
 vtkm::RangeId::Include (C++ function), 276  
 vtkm::RangeId::IsEmpty (C++ function), 276  
 vtkm::RangeId::Length (C++ function), 276  
 vtkm::RangeId::Max (C++ member), 277  
 vtkm::RangeId::Min (C++ member), 277  
 vtkm::RangeId::operator+ (C++ function), 277  
 vtkm::RangeId::RangeId (C++ function), 276  
 vtkm::RangeId::Union (C++ function), 276  
 vtkm::RCbrt (C++ function), 373, 374  
 vtkm::Remainder (C++ function), 375  
 vtkm::RemainderQuotient (C++ function), 375  
 vtkm::rendering  
     variable, 14  
 vtkm::rendering::Actor (C++ class), 163  
 vtkm::rendering::Actor::Actor (C++ function), 164

`vtkm::rendering::Actor::SetScalarRange` (C++ function), 164

`vtkm::rendering::Camera` (C++ class), 176

`vtkm::rendering::Camera::Azimuth` (C++ function), 183

`vtkm::rendering::Camera::Dolly` (C++ function), 184

`vtkm::rendering::Camera::Elevation` (C++ function), 183

`vtkm::rendering::Camera::GetClippingRange` (C++ function), 178

`vtkm::rendering::Camera::GetFieldOfView` (C++ function), 181

`vtkm::rendering::Camera::GetLookAt` (C++ function), 180

`vtkm::rendering::Camera::GetMode` (C++ function), 178

`vtkm::rendering::Camera::GetPan` (C++ function), 182

`vtkm::rendering::Camera::GetPosition` (C++ function), 180

`vtkm::rendering::Camera::GetViewport` (C++ function), 179

`vtkm::rendering::Camera::GetViewRange2D` (C++ function), 184

`vtkm::rendering::Camera::GetViewUp` (C++ function), 180

`vtkm::rendering::Camera::GetXScale` (C++ function), 180

`vtkm::rendering::Camera::GetZoom` (C++ function), 182

`vtkm::rendering::Camera::Pan` (C++ function), 181

`vtkm::rendering::Camera::ResetToBounds` (C++ function), 182, 183

`vtkm::rendering::Camera::Roll` (C++ function), 183

`vtkm::rendering::Camera::SetClippingRange` (C++ function), 178

`vtkm::rendering::Camera::SetFieldOfView` (C++ function), 181

`vtkm::rendering::Camera::SetLookAt` (C++ function), 180

`vtkm::rendering::Camera::SetMode` (C++ function), 178

`vtkm::rendering::Camera::SetModeTo2D` (C++ function), 178

`vtkm::rendering::Camera::SetModeTo3D` (C++ function), 178

`vtkm::rendering::Camera::SetPosition` (C++ function), 180

`vtkm::rendering::Camera::SetViewport` (C++ function), 179

`vtkm::rendering::Camera::SetViewRange2D` (C++ function), 184, 185

`vtkm::rendering::Camera::SetViewUp` (C++ function), 180

`vtkm::rendering::Camera::SetXScale` (C++ function), 181

`vtkm::rendering::Camera::TrackballRotate` (C++ function), 182

`vtkm::rendering::Camera::Zoom` (C++ function), 182

`vtkm::rendering::Canvas` (C++ class), 165

`vtkm::rendering::Canvas::BlendBackground` (C++ function), 165

`vtkm::rendering::Canvas::Canvas` (C++ function), 165

`vtkm::rendering::Canvas::Clear` (C++ function), 165

`vtkm::rendering::Canvas::CreateWorldAnnotator` (C++ function), 167

`vtkm::rendering::Canvas::GetBackgroundColor` (C++ function), 166

`vtkm::rendering::Canvas::GetColorBuffer` (C++ function), 166

`vtkm::rendering::Canvas::GetDataSet` (C++ function), 166

`vtkm::rendering::Canvas::GetDepthBuffer` (C++ function), 166

`vtkm::rendering::Canvas::GetForegroundColor` (C++ function), 166

`vtkm::rendering::Canvas::GetHeight` (C++ function), 166

`vtkm::rendering::Canvas::GetWidth` (C++ function), 166

`vtkm::rendering::Canvas::NewCopy` (C++ function), 165

`vtkm::rendering::Canvas::ResizeBuffers` (C++ function), 166

`vtkm::rendering::Canvas::SaveAs` (C++ function), 166

`vtkm::rendering::Canvas::SetBackgroundColor` (C++ function), 166

`vtkm::rendering::Canvas::SetForegroundColor` (C++ function), 166

`vtkm::rendering::CanvasRayTracer` (C++ class), 165

`vtkm::rendering::CanvasRayTracer::CanvasRayTracer` (C++ function), 165

`vtkm::rendering::CanvasRayTracer::NewCopy` (C++ function), 165

`vtkm::rendering::Color` (C++ class), 174

`vtkm::rendering::Color::Color` (C++ function), 174

`vtkm::rendering::Color::SetComponentFromByte` (C++ function), 175

`vtkm::rendering::Mapper` (C++ class), 167

`vtkm::rendering::MapperCylinder` (C++ class),



167  
 vtkm::rendering::MapperCylinder::SetRadius (C++ function), 167  
 vtkm::rendering::MapperCylinder::SetRadiusDelta (C++ function), 167  
 vtkm::rendering::MapperCylinder::UseVariableRadius (C++ function), 167  
 vtkm::rendering::MapperGlyphBase (C++ class), 167  
 vtkm::rendering::MapperGlyphBase::GetAssociation (C++ function), 168  
 vtkm::rendering::MapperGlyphBase::GetBaseSize (C++ function), 168  
 vtkm::rendering::MapperGlyphBase::GetScaleByValue (C++ function), 168  
 vtkm::rendering::MapperGlyphBase::GetScaleDelta (C++ function), 169  
 vtkm::rendering::MapperGlyphBase::GetUseCells (C++ function), 168  
 vtkm::rendering::MapperGlyphBase::GetUsePoints (C++ function), 168  
 vtkm::rendering::MapperGlyphBase::SetAssociation (C++ function), 168  
 vtkm::rendering::MapperGlyphBase::SetBaseSize (C++ function), 168  
 vtkm::rendering::MapperGlyphBase::SetScaleByValue (C++ function), 169  
 vtkm::rendering::MapperGlyphBase::SetScaleDelta (C++ function), 169  
 vtkm::rendering::MapperGlyphBase::SetUseCells (C++ function), 168  
 vtkm::rendering::MapperGlyphBase::SetUsePoints (C++ function), 168  
 vtkm::rendering::MapperGlyphScalar (C++ class), 169  
 vtkm::rendering::MapperGlyphScalar::GetGlyphType (C++ function), 169  
 vtkm::rendering::MapperGlyphScalar::SetGlyphType (C++ function), 169  
 vtkm::rendering::MapperGlyphVector (C++ class), 169  
 vtkm::rendering::MapperGlyphVector::GetGlyphType (C++ function), 169  
 vtkm::rendering::MapperGlyphVector::SetGlyphType (C++ function), 169  
 vtkm::rendering::MapperPoint (C++ class), 169  
 vtkm::rendering::MapperPoint::GetAssociation (C++ function), 170  
 vtkm::rendering::MapperPoint::GetUseCells (C++ function), 170  
 vtkm::rendering::MapperPoint::GetUsePoints (C++ function), 170  
 vtkm::rendering::MapperPoint::SetAssociation (C++ function), 170  
 vtkm::rendering::MapperPoint::SetRadius (C++ function), 170  
 vtkm::rendering::MapperPoint::SetRadiusDelta (C++ function), 170  
 vtkm::rendering::MapperPoint::SetUseCells (C++ function), 170  
 vtkm::rendering::MapperPoint::SetUsePoints (C++ function), 170  
 vtkm::rendering::MapperPoint::UseVariableRadius (C++ function), 170  
 vtkm::rendering::MapperQuad (C++ class), 171  
 vtkm::rendering::MapperRayTracer (C++ class), 171  
 vtkm::rendering::MapperVolume (C++ class), 171  
 vtkm::rendering::MapperVolume::SetSampleDistance (C++ function), 171  
 vtkm::rendering::MapperWireframer (C++ class), 171  
 vtkm::rendering::MapperWireframer::GetShowInternalZones (C++ function), 171  
 vtkm::rendering::MapperWireframer::SetShowInternalZones (C++ function), 171  
 vtkm::rendering::Scene (C++ class), 164  
 vtkm::rendering::Scene::AddActor (C++ function), 164  
 vtkm::rendering::Scene::GetActor (C++ function), 164  
 vtkm::rendering::Scene::GetNumberOfActors (C++ function), 164  
 vtkm::rendering::Scene::GetSpatialBounds (C++ function), 164  
 vtkm::rendering::View (C++ class), 172  
 vtkm::rendering::View1D (C++ class), 173  
 vtkm::rendering::View1D::Paint (C++ function), 173  
 vtkm::rendering::View1D::SetLogX (C++ function), 173  
 vtkm::rendering::View1D::SetLogY (C++ function), 173  
 vtkm::rendering::View2D (C++ class), 173  
 vtkm::rendering::View2D::Paint (C++ function), 174  
 vtkm::rendering::View3D (C++ class), 174  
 vtkm::rendering::View3D::Paint (C++ function), 174  
 vtkm::rendering::View::GetBackgroundColor (C++ function), 173  
 vtkm::rendering::View::GetCamera (C++ function), 172, 173  
 vtkm::rendering::View::GetCanvas (C++ function), 172  
 vtkm::rendering::View::GetMapper (C++ function), 172  
 vtkm::rendering::View::GetScene (C++ function),

172  
`vtkm::rendering::View::Paint` (C++ function), 173  
`vtkm::rendering::View::SaveAs` (C++ function), 173  
`vtkm::rendering::View::SetBackgroundColor` (C++ function), 173  
`vtkm::rendering::View::SetCamera` (C++ function), 173  
`vtkm::rendering::View::SetForegroundColor` (C++ function), 173  
`vtkm::rendering::View::SetScene` (C++ function), 172  
`vtkm::RMagnitude` (C++ function), 383  
`vtkm::Round` (C++ function), 376, 377  
`vtkm::RSqrt` (C++ function), 374  
`vtkm::SignBit` (C++ function), 377  
`vtkm::Sin` (C++ function), 380  
`vtkm::SinH` (C++ function), 380  
`vtkm::SolveLinearSystem` (C++ function), 386  
`vtkm::source` variable, 14  
`vtkm::Sphere` (C++ class), 222  
`vtkm::Sphere::Classify` (C++ function), 223  
`vtkm::Sphere::Contains` (C++ function), 222  
`vtkm::Sphere::GetCenter` (C++ function), 223  
`vtkm::Sphere::GetRadius` (C++ function), 223  
`vtkm::Sphere::Gradient` (C++ function), 223  
`vtkm::Sphere::IsValid` (C++ function), 222  
`vtkm::Sphere::SetCenter` (C++ function), 223  
`vtkm::Sphere::SetRadius` (C++ function), 223  
`vtkm::Sphere::Sphere` (C++ function), 222, 223  
`vtkm::Sphere::Value` (C++ function), 223  
`vtkm::Sqrt` (C++ function), 374, 375  
`vtkm::Tan` (C++ function), 381  
`vtkm::TanH` (C++ function), 381  
`vtkm::Transform` (C++ function), 303  
`vtkm::TriangleNormal` (C++ function), 383  
`vtkm::Tuple` (C++ class), 300  
`vtkm::TupleElement` (C++ type), 301  
`vtkm::TupleSize` (C++ type), 300  
`vtkm::TwoPi` (C++ function), 381  
`vtkm::TypeListAll` (C++ type), 290  
`vtkm::TypeListBaseC` (C++ type), 290  
`vtkm::TypeListCommon` (C++ type), 290  
`vtkm::TypeListField` (C++ type), 289  
`vtkm::TypeListFieldScalar` (C++ type), 289  
`vtkm::TypeListFieldVec2` (C++ type), 289  
`vtkm::TypeListFieldVec3` (C++ type), 289  
`vtkm::TypeListFieldVec4` (C++ type), 289  
`vtkm::TypeListFloatVec` (C++ type), 289  
`vtkm::TypeListId` (C++ type), 289  
`vtkm::TypeListId2` (C++ type), 289  
`vtkm::TypeListId3` (C++ type), 289  
`vtkm::TypeListId4` (C++ type), 289  
`vtkm::TypeListIdComponent` (C++ type), 289  
`vtkm::TypeListIndex` (C++ type), 289  
`vtkm::TypeListScalarAll` (C++ type), 290  
`vtkm::TypeListVecAll` (C++ type), 290  
`vtkm::TypeListVecCommon` (C++ type), 290  
`vtkm::TypeTraits` (C++ class), 280  
`vtkm::TypeTraits::DimensionalityTag` (C++ type), 280  
`vtkm::TypeTraits::NumericTag` (C++ type), 280  
`vtkm::TypeTraits::ZeroInitialization` (C++ function), 281  
`vtkm::TypeTraitsIntegerTag` (C++ struct), 281  
`vtkm::TypeTraitsRealTag` (C++ struct), 281  
`vtkm::TypeTraitsScalarTag` (C++ struct), 281  
`vtkm::TypeTraitsUnknownTag` (C++ struct), 281  
`vtkm::TypeTraitsVectorTag` (C++ struct), 281  
`vtkm::UInt16` (C++ type), 26  
`vtkm::UInt32` (C++ type), 27  
`vtkm::UInt64` (C++ type), 27  
`vtkm::UInt8` (C++ type), 26  
`vtkm::Vec` (C++ class), 265  
`vtkm::Vec2f` (C++ type), 27  
`vtkm::Vec2f_32` (C++ type), 28  
`vtkm::Vec2f_64` (C++ type), 28  
`vtkm::Vec2i` (C++ type), 29  
`vtkm::Vec2i_16` (C++ type), 30  
`vtkm::Vec2i_32` (C++ type), 30  
`vtkm::Vec2i_64` (C++ type), 30  
`vtkm::Vec2i_8` (C++ type), 29  
`vtkm::Vec2ui` (C++ type), 29  
`vtkm::Vec2ui_16` (C++ type), 30  
`vtkm::Vec2ui_32` (C++ type), 30  
`vtkm::Vec2ui_64` (C++ type), 30  
`vtkm::Vec2ui_8` (C++ type), 30  
`vtkm::Vec3f` (C++ type), 27  
`vtkm::Vec3f_32` (C++ type), 28  
`vtkm::Vec3f_64` (C++ type), 28  
`vtkm::Vec3i` (C++ type), 29  
`vtkm::Vec3i_16` (C++ type), 30  
`vtkm::Vec3i_32` (C++ type), 31  
`vtkm::Vec3i_64` (C++ type), 31  
`vtkm::Vec3i_8` (C++ type), 30  
`vtkm::Vec3ui` (C++ type), 29  
`vtkm::Vec3ui_16` (C++ type), 31  
`vtkm::Vec3ui_32` (C++ type), 31  
`vtkm::Vec3ui_64` (C++ type), 31  
`vtkm::Vec3ui_8` (C++ type), 30  
`vtkm::Vec4f` (C++ type), 27  
`vtkm::Vec4f_32` (C++ type), 28  
`vtkm::Vec4f_64` (C++ type), 28  
`vtkm::Vec4i` (C++ type), 29  
`vtkm::Vec4i_16` (C++ type), 31  
`vtkm::Vec4i_32` (C++ type), 31  
`vtkm::Vec4i_64` (C++ type), 32

vtkm::Vec4i\_8 (C++ type), 31  
 vtkm::Vec4ui (C++ type), 29  
 vtkm::Vec4ui\_16 (C++ type), 31  
 vtkm::Vec4ui\_32 (C++ type), 31  
 vtkm::Vec4ui\_64 (C++ type), 32  
 vtkm::Vec4ui\_8 (C++ type), 31  
 vtkm::VecC (C++ class), 268  
 vtkm::VecCConst (C++ class), 268  
 vtkm::VecFromPortal (C++ class), 270  
 vtkm::VecFromPortalPermute (C++ class), 270  
 vtkm::VecTraits (C++ struct), 283  
 vtkm::VecTraits::BaseComponentType (C++ type), 283  
 vtkm::VecTraits::ComponentType (C++ type), 283  
 vtkm::VecTraits::CopyInto (C++ function), 285  
 vtkm::VecTraits::GetComponent (C++ function), 284  
 vtkm::VecTraits::GetNumberOfComponents (C++ function), 284  
 vtkm::VecTraits::HasMultipleComponents (C++ type), 284  
 vtkm::VecTraits::IsSizeStatic (C++ type), 284  
 vtkm::VecTraits::NUM\_COMPONENTS (C++ member), 285  
 vtkm::VecTraits::ReplaceBaseComponentType (C++ type), 284  
 vtkm::VecTraits::ReplaceComponentType (C++ type), 284  
 vtkm::VecTraits::SetComponent (C++ function), 284  
 vtkm::VecTraitsTagMultipleComponents (C++ struct), 285  
 vtkm::VecTraitsTagSingleComponent (C++ struct), 285  
 vtkm::VecTraitsTagSizeStatic (C++ struct), 285  
 vtkm::VecTraitsTagSizeVariable (C++ struct), 285  
 vtkm::VecVariable (C++ class), 269  
 vtkm::worklet::Keys (C++ class), 349  
 vtkm::worklet::Keys::BuildArrays (C++ function), 350  
 vtkm::worklet::Keys::BuildArraysInPlace (C++ function), 350  
 vtkm::worklet::Keys::GetInputRange (C++ function), 350  
 vtkm::worklet::Keys::GetNumberOfValues (C++ function), 351  
 vtkm::worklet::Keys::GetOffsets (C++ function), 350  
 vtkm::worklet::Keys::GetSortedValuesMap (C++ function), 350  
 vtkm::worklet::Keys::GetUniqueKeys (C++ function), 350  
 vtkm::worklet::Keys::Keys (C++ function), 350  
 vtkm::worklet::WorkletCellNeighborhood (C++ class), 335  
 vtkm::worklet::WorkletCellNeighborhood::\_1 (C++ struct), 337  
 vtkm::worklet::WorkletCellNeighborhood::AtomicArrayInOut (C++ struct), 336  
 vtkm::worklet::WorkletCellNeighborhood::Boundary (C++ struct), 337  
 vtkm::worklet::WorkletCellNeighborhood::CellSetIn (C++ struct), 335  
 vtkm::worklet::WorkletCellNeighborhood::Device (C++ struct), 338  
 vtkm::worklet::WorkletCellNeighborhood::ExecObject (C++ struct), 337  
 vtkm::worklet::WorkletCellNeighborhood::FieldIn (C++ struct), 335  
 vtkm::worklet::WorkletCellNeighborhood::FieldInNeighborhood (C++ struct), 335  
 vtkm::worklet::WorkletCellNeighborhood::FieldInOut (C++ struct), 336  
 vtkm::worklet::WorkletCellNeighborhood::FieldOut (C++ struct), 336  
 vtkm::worklet::WorkletCellNeighborhood::InputIndex (C++ struct), 337  
 vtkm::worklet::WorkletCellNeighborhood::OutputIndex (C++ struct), 338  
 vtkm::worklet::WorkletCellNeighborhood::ThreadIndices (C++ struct), 338  
 vtkm::worklet::WorkletCellNeighborhood::VisitIndex (C++ struct), 337  
 vtkm::worklet::WorkletCellNeighborhood::WholeArrayIn (C++ struct), 336  
 vtkm::worklet::WorkletCellNeighborhood::WholeArrayInOut (C++ struct), 336  
 vtkm::worklet::WorkletCellNeighborhood::WholeArrayOut (C++ struct), 336  
 vtkm::worklet::WorkletCellNeighborhood::WholeCellSetIn (C++ struct), 337  
 vtkm::worklet::WorkletCellNeighborhood::WorkIndex (C++ struct), 337  
 vtkm::worklet::WorkletMapField (C++ class), 318  
 vtkm::worklet::WorkletMapField::\_1 (C++ struct), 319  
 vtkm::worklet::WorkletMapField::AtomicArrayInOut (C++ struct), 319  
 vtkm::worklet::WorkletMapField::Device (C++ struct), 320  
 vtkm::worklet::WorkletMapField::ExecObject (C++ struct), 319  
 vtkm::worklet::WorkletMapField::FieldIn (C++ struct), 318  
 vtkm::worklet::WorkletMapField::FieldInOut (C++ struct), 318  
 vtkm::worklet::WorkletMapField::FieldOut

(C++ struct), 318	(C++ struct), 333
vtkm::worklet::WorkletMapField::InputIndex (C++ struct), 320	vtkm::worklet::WorkletPointNeighborhood::WholeCellSetIn (C++ struct), 333
vtkm::worklet::WorkletMapField::OutputIndex (C++ struct), 320	vtkm::worklet::WorkletPointNeighborhood::WorkIndex (C++ struct), 334
vtkm::worklet::WorkletMapField::ThreadIndices (C++ struct), 320	vtkm::worklet::WorkletReduceByKey (C++ class), 346
vtkm::worklet::WorkletMapField::VisitIndex (C++ struct), 320	vtkm::worklet::WorkletReduceByKey::_1 (C++ struct), 348
vtkm::worklet::WorkletMapField::WholeArrayIn (C++ struct), 319	vtkm::worklet::WorkletReduceByKey::AtomicArrayInOut (C++ struct), 347
vtkm::worklet::WorkletMapField::WholeArrayInOut (C++ struct), 319	vtkm::worklet::WorkletReduceByKey::Device (C++ struct), 349
vtkm::worklet::WorkletMapField::WholeArrayOut (C++ struct), 319	vtkm::worklet::WorkletReduceByKey::ExecObject (C++ struct), 348
vtkm::worklet::WorkletMapField::WholeCellSetIn (C++ struct), 319	vtkm::worklet::WorkletReduceByKey::InputIndex (C++ struct), 349
vtkm::worklet::WorkletMapField::WorkIndex (C++ struct), 320	vtkm::worklet::WorkletReduceByKey::KeysIn (C++ struct), 346
vtkm::worklet::WorkletPointNeighborhood (C++ class), 332	vtkm::worklet::WorkletReduceByKey::OutputIndex (C++ struct), 349
vtkm::worklet::WorkletPointNeighborhood::_1 (C++ struct), 334	vtkm::worklet::WorkletReduceByKey::ReducedValuesIn (C++ struct), 347
vtkm::worklet::WorkletPointNeighborhood::AtomicArrayIn (C++ struct), 333	vtkm::worklet::WorkletReduceByKey::ReducedValuesInOut (C++ struct), 347
vtkm::worklet::WorkletPointNeighborhood::Boundary (C++ struct), 334	vtkm::worklet::WorkletReduceByKey::ReducedValuesOut (C++ struct), 346
vtkm::worklet::WorkletPointNeighborhood::CellSetIn (C++ struct), 332	vtkm::worklet::WorkletReduceByKey::ThreadIndices (C++ struct), 349
vtkm::worklet::WorkletPointNeighborhood::Device (C++ struct), 335	vtkm::worklet::WorkletReduceByKey::ValueCount (C++ struct), 348
vtkm::worklet::WorkletPointNeighborhood::ExecObject (C++ struct), 333	vtkm::worklet::WorkletReduceByKey::ValuesIn (C++ struct), 346
vtkm::worklet::WorkletPointNeighborhood::FieldIn (C++ struct), 332	vtkm::worklet::WorkletReduceByKey::ValuesInOut (C++ struct), 346
vtkm::worklet::WorkletPointNeighborhood::FieldInNeighborhood (C++ struct), 332	vtkm::worklet::WorkletReduceByKey::ValuesOut (C++ struct), 346
vtkm::worklet::WorkletPointNeighborhood::FieldInOut (C++ struct), 332	vtkm::worklet::WorkletReduceByKey::VisitIndex (C++ struct), 348
vtkm::worklet::WorkletPointNeighborhood::FieldOut (C++ struct), 332	vtkm::worklet::WorkletReduceByKey::WholeArrayIn (C++ struct), 347
vtkm::worklet::WorkletPointNeighborhood::InputIndex (C++ struct), 334	vtkm::worklet::WorkletReduceByKey::WholeArrayInOut (C++ struct), 347
vtkm::worklet::WorkletPointNeighborhood::OutputIndex (C++ struct), 334	vtkm::worklet::WorkletReduceByKey::WholeArrayOut (C++ struct), 347
vtkm::worklet::WorkletPointNeighborhood::ThreadIndices (C++ struct), 335	vtkm::worklet::WorkletReduceByKey::WholeCellSetIn (C++ struct), 348
vtkm::worklet::WorkletPointNeighborhood::VisitIndex (C++ struct), 334	vtkm::worklet::WorkletReduceByKey::WorkIndex (C++ struct), 348
vtkm::worklet::WorkletPointNeighborhood::WholeArrayIn (C++ struct), 333	vtkm::worklet::WorkletVisitCellsWithPoints (C++ class), 322
vtkm::worklet::WorkletPointNeighborhood::WholeArrayInOut (C++ struct), 333	vtkm::worklet::WorkletVisitCellsWithPoints::_1 (C++ struct), 324
vtkm::worklet::WorkletPointNeighborhood::WholeArrayOut (C++ struct), 333	vtkm::worklet::WorkletVisitCellsWithPoints::AtomicArrayIn (C++ struct), 324

(C++ struct), 324  
 vtkm::worklet::WorkletVisitCellsWithPoints::CellSet vtkm::worklet::WorkletVisitPointsWithCells::CellIndices  
 (C++ struct), 322 (C++ struct), 329  
 vtkm::worklet::WorkletVisitCellsWithPoints::CellShape vtkm::worklet::WorkletVisitPointsWithCells::CellSetIn  
 (C++ struct), 324 (C++ struct), 327  
 vtkm::worklet::WorkletVisitCellsWithPoints::Device vtkm::worklet::WorkletVisitPointsWithCells::Device  
 (C++ struct), 326 (C++ struct), 330  
 vtkm::worklet::WorkletVisitCellsWithPoints::ExecObject vtkm::worklet::WorkletVisitPointsWithCells::ExecObject  
 (C++ struct), 324 (C++ struct), 329  
 vtkm::worklet::WorkletVisitCellsWithPoints::FieldInCell vtkm::worklet::WorkletVisitPointsWithCells::FieldInCell  
 (C++ struct), 322 (C++ struct), 327  
 vtkm::worklet::WorkletVisitCellsWithPoints::FieldInIncident vtkm::worklet::WorkletVisitPointsWithCells::FieldInIncident  
 (C++ struct), 323 (C++ struct), 327  
 vtkm::worklet::WorkletVisitCellsWithPoints::FieldInOut vtkm::worklet::WorkletVisitPointsWithCells::FieldInOut  
 (C++ struct), 323 (C++ struct), 328  
 vtkm::worklet::WorkletVisitCellsWithPoints::FieldInOutCell vtkm::worklet::WorkletVisitPointsWithCells::FieldInOutPoint  
 (C++ struct), 323 (C++ struct), 328  
 vtkm::worklet::WorkletVisitCellsWithPoints::FieldInPoint vtkm::worklet::WorkletVisitPointsWithCells::FieldInPoint  
 (C++ struct), 322 (C++ struct), 327  
 vtkm::worklet::WorkletVisitCellsWithPoints::FieldInVisit vtkm::worklet::WorkletVisitPointsWithCells::FieldInVisit  
 (C++ struct), 323 (C++ struct), 327  
 vtkm::worklet::WorkletVisitCellsWithPoints::FieldOut vtkm::worklet::WorkletVisitPointsWithCells::FieldOut  
 (C++ struct), 323 (C++ struct), 328  
 vtkm::worklet::WorkletVisitCellsWithPoints::FieldOutCell vtkm::worklet::WorkletVisitPointsWithCells::FieldOutPoint  
 (C++ struct), 323 (C++ struct), 327  
 vtkm::worklet::WorkletVisitCellsWithPoints::InputIndex vtkm::worklet::WorkletVisitPointsWithCells::InputIndex  
 (C++ struct), 325 (C++ struct), 330  
 vtkm::worklet::WorkletVisitCellsWithPoints::OutputIndex vtkm::worklet::WorkletVisitPointsWithCells::OutputIndex  
 (C++ struct), 325 (C++ struct), 330  
 vtkm::worklet::WorkletVisitCellsWithPoints::PointCount vtkm::worklet::WorkletVisitPointsWithCells::ThreadIndices  
 (C++ struct), 325 (C++ struct), 330  
 vtkm::worklet::WorkletVisitCellsWithPoints::PointIndices vtkm::worklet::WorkletVisitPointsWithCells::VisitIndex  
 (C++ struct), 325 (C++ struct), 329  
 vtkm::worklet::WorkletVisitCellsWithPoints::ThreadIndices vtkm::worklet::WorkletVisitPointsWithCells::WholeArrayIn  
 (C++ struct), 326 (C++ struct), 328  
 vtkm::worklet::WorkletVisitCellsWithPoints::VisitIndex vtkm::worklet::WorkletVisitPointsWithCells::WholeArrayInOut  
 (C++ struct), 325 (C++ struct), 328  
 vtkm::worklet::WorkletVisitCellsWithPoints::WholeArrayIn vtkm::worklet::WorkletVisitPointsWithCells::WholeArrayOut  
 (C++ struct), 323 (C++ struct), 328  
 vtkm::worklet::WorkletVisitCellsWithPoints::WholeArrayInDet vtkm::worklet::WorkletVisitPointsWithCells::WholeCellSetIn  
 (C++ struct), 324 (C++ struct), 329  
 vtkm::worklet::WorkletVisitCellsWithPoints::WholeArrayOut vtkm::worklet::WorkletVisitPointsWithCells::WorkIndex  
 (C++ struct), 324 (C++ struct), 329  
 vtkm::worklet::WorkletVisitCellsWithPoints::WholeCellSetIn C macro), 205  
 (C++ struct), 324 VTKm\_ENABLE\_BENCHMARKS  
 vtkm::worklet::WorkletVisitCellsWithPoints::WorkIndex variable, 11  
 (C++ struct), 325 VTKm\_ENABLE\_CUDA  
 vtkm::worklet::WorkletVisitPointsWithCells variable, 11, 14  
 (C++ class), 327 VTKm\_ENABLE\_EXAMPLES  
 vtkm::worklet::WorkletVisitPointsWithCells::\_1 variable, 11  
 (C++ struct), 329 VTKm\_ENABLE\_KOKKOS  
 vtkm::worklet::WorkletVisitPointsWithCells::AtomicArrayOut C macro), 205  
 (C++ struct), 328 VTKm\_ENABLE\_Kokkos  
 vtkm::worklet::WorkletVisitPointsWithCells::CellCount variable, 15



VTKm\_ENABLE\_MPI  
     variable, [11](#), [15](#)  
 VTKm\_ENABLE\_OPENMP  
     variable, [11](#), [15](#)  
 VTKm\_ENABLE\_RENDERING  
     variable, [11](#), [14](#), [15](#)  
 VTKm\_ENABLE\_TBB  
     variable, [11](#), [15](#)  
 VTKm\_ENABLE\_TESTING  
     variable, [11](#)  
 VTKm\_ENABLE\_TUTORIALS  
     variable, [11](#)  
 VTKm\_FOUND  
     variable, [14](#)  
 VTKM\_IS\_ARRAY\_HANDLE (*C macro*), [245](#)  
 VTKM\_IS\_CELL\_SHAPE\_TAG (*C macro*), [392](#)  
 VTKM\_IS\_LIST (*C macro*), [291](#)  
 VTKM\_LOG\_F (*C macro*), [312](#)  
 VTKM\_LOG\_IF\_F (*C macro*), [313](#)  
 VTKM\_LOG\_IF\_S (*C macro*), [313](#)  
 VTKM\_LOG\_S (*C macro*), [312](#)  
 VTKM\_LOG\_SCOPE (*C macro*), [314](#)  
 VTKM\_LOG\_SCOPE\_FUNCTION (*C macro*), [314](#)  
 VTKm\_USE\_64BIT\_IDS  
     variable, [11](#), [26](#)  
 VTKm\_USE\_DOUBLE\_PRECISION  
     variable, [11](#), [26](#), [27](#)  
 VTKm\_VERSION  
     variable, [14](#), [33](#)  
 VTKM\_VERSION (*C macro*), [33](#)  
 VTKm\_VERSION\_FULL  
     variable, [14](#), [33](#)  
 VTKM\_VERSION\_FULL (*C macro*), [33](#)  
 VTKm\_VERSION\_MAJOR  
     variable, [14](#), [33](#)  
 VTKM\_VERSION\_MAJOR (*C macro*), [33](#)  
 VTKm\_VERSION\_MINOR  
     variable, [14](#), [33](#)  
 VTKM\_VERSION\_MINOR (*C macro*), [33](#)  
 VTKm\_VERSION\_PATCH  
     variable, [14](#), [33](#)  
 VTKM\_VERSION\_PATCH (*C macro*), [33](#)  
 vtkmGenericCellShapeMacro (*C macro*), [393](#)

## W

warp  
     filter, [124](#)  
 wireframe  
     rendering, [176](#)  
 worklet, [231](#), [317](#)  
     cell neighborhood, [335](#)  
     control signature, [248](#)  
     creating, [247](#), [317](#)  
     error handling, [367](#)

    execution signature, [248](#)  
     field map, [317](#), [318](#)  
     input domain, [249](#)  
     invoke, [249](#)  
     neighborhood, [331](#)  
     point neighborhood, [317](#), [332](#)  
     reduce by key, [317](#), [345](#)  
     topology map, [317](#)  
     visit cells, [317](#), [322](#)  
     visit points, [317](#), [327](#)  
 world coordinates  
     cell, [396](#)  
 write file, [79](#)

## Z

zfp  
     filter, [160](#)  
 zoom  
     camera rendering, [186](#)  
     mouse, [193](#)  
     rendering, [193](#)