VTK software Process
*(short form)*
2017-05-19 (version 2)

Index

1. Introduction
2. Releases
3. Coding Conventions
   a. General
   b. C++ Conventions
   c. Other Languages
4. Contributing Code
5. Regression Testing
6. Interpreting the Dashboard Results
7. Writing New Tests
8. Bug and Feature Tracking
9. Documentation

Introduction

VTK’s software process enables decentralized development of the library and ensures constant software quality. The purpose of this document is to record and explain the current VTK software development process. By doing so, we aim to soften the learning curve for developers new and old.

The process has evolved throughout the lifetime of VTK. By setting down the process in writing we seek not to hamper that evolution but rather to provide a defined basis for proposing further changes in order to continue its improvement. We seek a process that does not place needless or difficult restrictions on developers, encourages contributions from new developers, and does not require significant effort to centrally manage.

Releases
We aim for a six-month release cycle. However we allow that this schedule is a soft one. The project is funded and developed by many different groups, all of which work towards their own particular sets of features. In the past, Kitware has released official versions only when a customer with a particular need for a release asked for one.

VTK releases are named with a Major.Minor.Patch scheme. Prior to VTK 6, odd minor numbers indicated only any place in the span between official even minor numbered stable releases. After VTK 6 that scheme was dropped.

The shape of the overall release history is that of a skinny tree. Development proceeds along the trunk or “master” branch (taking the form of topic branches that start from and are merged into master), and every so often a release is tagged and branched from it. In general no work goes into the “release” branch, other than the handful of important patches that make up the occasional patch release.

On the master branch, bug fixes and new features are continuously developed. At release time, the focus temporarily shifts to producing a library that is as stable and robust as possible. The process for cutting releases is as follows:

1. Inform developers that a release is coming
   A few weeks before the intended release branch, announce on the mailing list that a new release is nearing. This alerts developers to hold off making drastic changes that might delay the release and gives them a chance to push important and nearly completed features in time for the release.

2. Polish the dashboards and bug tracker
   Persistent compilation and regression test problems are fixed. Serious outstanding bugs are fixed.

3. Forward release branch
   When the dashboards are clean and the outstanding features are finished, we pick a point on the development branch to be the start of the next release branch. Next we move the release branch forward from its current position to the new one.

4. Gather descriptions of changes
   a. Compile a list of developers and their changes and send emails to each developer asking them to summarize their work.
   b. Run the API differencer script in Utilities/Maintainance/semanticDiffVersion.py

5. Do the release candidate cycle
   a. Tag the release branch and make and publish release candidate tar
balls and change summaries.

b. Announce the release candidate and request feedback from the community and especially third party packagers.

Bug reports should be entered into the bug tracker with milestone set to the upcoming release number.

If no important bugs are reported fourteen days after the candidate is published, the source is re-tagged and packaged as the official release.

c. If the community does report bugs, the manager classifies them in the bug tracker and sees that they are fixed.

Only serious bugs and regressions need to be fixed before the release. New features and minor problems should go into the master branch as usual.

Patches for the release branch must start from the release branch, be submitted through gitlab, and merged into master. Once fully tested there the branch can be merged into the release branch.

When the selected issues are fixed in the release branch, the tip of the release branch is tagged and released as the next candidate and the cycle continues.

6. Package the official release

The official VTK package consists of tar balls and zip files of: the source, the doxygen documentation, and regression test data, all at the tag point. Volunteer third party packagers create binary packages from the official release for various platforms, so their input is especially valuable during the release cycle.

The release manager also compiles release notes that go into the official release announcement. These are compiled from the information gathered in step 4 above and from the bug tracker’s report of closed items. The manager also ensures that API changes in the new version of VTK are summarized on the wiki. Ex. (http://www.vtk.org/Wiki/VTK/Release5100_New_Classes and http://www.vtk.org/Wiki/VTK/API_Changes_6_0_0_to_6_1_0). The summary should include names of new and removed classes, descriptions of other changes to the API, and a listing of all newly deprecated code. The summary is useful when upgrading external applications to use the new version.

General Coding Conventions
**General**

VTK is a large body of code with many users and developers. Coding in a consistent style eases shared development. VTK’s style guidelines also ensure wide portability. All code that is contributed to VTK must conform to the following style guidelines. Exceptions are permissible, following discussion in code review, as long as the result passes the nightly regression tests. External code contributed into the ThirdParty directory is exempt from most of the following rules except for the rules that say “All code”.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>All code that is compiled into VTK by default must be compatible with VTK’s BSD-style license.</td>
</tr>
<tr>
<td>2.</td>
<td>Copyright notices should appear at the top of header and implementation files.</td>
</tr>
<tr>
<td>3.</td>
<td>All C++ code must be valid C++11 code.</td>
</tr>
<tr>
<td>4.</td>
<td>The Tcl, Java and Python wrappers must work on new code, or it should be excluded from wrapping.</td>
</tr>
</tbody>
</table>
| 5.   | Multiple inheritance is not allowed in VTK classes.  
Rationale: One important reason is that Java does not support it. |
| 6.   | Only one public class per header file. Internal helper classes may be forward declared in header files, but can then only be defined in implementation files, ie using the PIMPL idiom.  
Rationale: helpful when searching the code and limits header inclusion bloat that slows compilation time. |
| 7.   | Class names and file names must match, class names must be unique.  
Rationale: helpful when searching the code, includes are flattened at install. |
| 8.   | The indentation style can be characterized as the modified Allman style (https://en.wikipedia.org/wiki/Indent_style#Allman_style) style. Indentations are two spaces, and the curly brace (scope delimiter) is placed on the following line and indented to the same level as the control statement.  
Rationale: Readability and historical |
<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.</td>
<td>Conditional clauses (including loop conditionals such as <code>for</code> and <code>while</code>) must be in braces below the conditional. <strong>Rationale</strong>: helpful when running code through a debugger</td>
</tr>
<tr>
<td></td>
<td>Instead of <code>if (test) clause or if (test) { clause }</code>, use <code>if (test) { clause }</code></td>
</tr>
<tr>
<td>10.</td>
<td>Two space indentation. Tabs are not allowed. Trailing whitespace is not allowed. <strong>Rationale</strong>: Removing tabs ensures that blocks are indented consistently in all editors</td>
</tr>
<tr>
<td>11.</td>
<td>Only alphanumeric characters in names. Use capitalization to demarcate words within a name (i.e., camel case). Preprocessor variables are the exception, and should be in all caps with a single underscore to demarcate words. <strong>Rationale</strong>: Readability</td>
</tr>
<tr>
<td>12.</td>
<td>Every class, macro, etc starts with either vtk or VTK. Classes should all start with lowercase vtk and macros or constants can start with either. <strong>Rationale</strong>: avoids name clashes with other libraries</td>
</tr>
<tr>
<td>13.</td>
<td>After the “vtk” prefix, capitalize the first letter of class names, methods and static and instance variables. Local variables are allowed to vary, but ideally should start in lower case and then proceed in camel case. <strong>Rationale</strong>: Readability</td>
</tr>
<tr>
<td>14.</td>
<td>Try to always spell out a name and not use abbreviations except in cases where the shortened form is obvious and widely understood. <strong>Rationale</strong>: Readability, self-documentation</td>
</tr>
<tr>
<td>15.</td>
<td>Classes that derive from vtkObject should have protected constructors and destructors, and privately declared but unimplemented copy constructor and assignment operator.</td>
</tr>
</tbody>
</table>
a. Classes that don't derive from vtkObject should obey the rule of three. If the class implements the destructor, copy constructor or copy assignment operator they should implement all of them.

Rationale: VTK's reference counting implementation depends on carefully controlling each object's lifetime.

16. Following the copyright notice, the name and purpose of each class should be documented at the top of the header with standard doxygen markup:

```/**
 * @class vtkclassname
 * @brief one line description
 *
 * Longer description of class here.
 */
```

Rationale: Doxygen generated documentation uses this to describe each class.

17. Public methods must be documented with doxygen markup.

```/**
 * Explanation of what the method/ivar is for
 */
```

Descriptions should do more than simply restate the method or ivar’s name.

The documentation for each public ivar should document the default value.

The documentation style for SetGet macros should be a single comment for the pair and a brief description of the variable that is being set/get. Use doxygen group marking to make the comment apply to both macro expanded functions.

```//@{
/**
 */```
* Set / get the sharpness of decay of the splats.
* This is the exponent constant in the Gaussian
* equation. Normally this is a negative value.
*/
*/
vtkSetMacro(ExponentFactor, double);
vtkGetMacro(ExponentFactor, double);
//@}

The documentation style for vector macros is to name each of the resulting variables. For example
/**
 * Set/Get the color which is used to draw shapes in
 * the image. The parameters are SetDrawColor(red, green,
 * blue, alpha)
 */
vtkSetVector4Macro(DrawColor, double);
vtkGetVector4Macro(DrawColor, double);

The description for SetClamp macros must describe the valid range of values.
/**
 * Should the data with value 0 be ignored? Valid
 * range (0, 1).
 */
vtkSetClampMacro(IgnoreZero, int, 0, 1);
vtkGetMacro(IgnoreZero, int);

Rationale: Doxygen generated documentation
(http://www.vtk.org/doc/nightly/html/) is generated from these comments and should be consistently readable.

18. Public instance variables are allowed only in exceptional situations.
   Protected (or if necessary Private) variables should be used instead with
   public access given via Set/Get macro methods.
<table>
<thead>
<tr>
<th></th>
<th>Rationale: Consistent API, and SetMacro takes part in reference counting.</th>
</tr>
</thead>
<tbody>
<tr>
<td>19.</td>
<td>Accessors to vtkObject instance variables should be declared in the header file, and defined in the implementation file with the vtkCxxSetObjectMacro.</td>
</tr>
<tr>
<td></td>
<td>Rationale: Reduces header file bloat and assists in reference counting.</td>
</tr>
<tr>
<td>20.</td>
<td>Use &quot;this-&gt;&quot; inside of methods to access class methods and instance variables.</td>
</tr>
<tr>
<td></td>
<td>Rationale: Readability as it helps to distinguish local variables from instance variables.</td>
</tr>
<tr>
<td>21.</td>
<td>Header files should normally have just two includes, one for the superclass' header file and one for the class' module export header declaration. It is required that all but the superclass header have a comment explaining why the extra includes are necessary. Care should be taken to minimize the number of includes in public headers, with predeclaration/PIMPL preferred.</td>
</tr>
<tr>
<td></td>
<td>Rationale: limits header inclusion bloat that slows compilation time.</td>
</tr>
<tr>
<td>22.</td>
<td>Include statements in implementation files should generally be in alphabetical order, grouped by type. For example, VTK includes first, system includes, STL includes, and Qt includes.</td>
</tr>
<tr>
<td></td>
<td>Rationale: avoid redundant includes, and keep a logical order.</td>
</tr>
<tr>
<td>23.</td>
<td>All subclasses of vtkObject should include a PrintSelf() method that prints all publicly accessible ivars.</td>
</tr>
<tr>
<td></td>
<td>Rationale: useful in debugging and in wrapped languages that lack sufficient introspection.</td>
</tr>
<tr>
<td>24.</td>
<td>All subclasses of vtkObject should include a type macro in their class declaration.</td>
</tr>
<tr>
<td></td>
<td>Rationale: VTK's implementation of runtime type information depends on it</td>
</tr>
<tr>
<td>25.</td>
<td>Do not use 'id' as a variable name in public headers, also avoid ‘min’, ‘max’, and other symbols that conflict with the Windows API.</td>
</tr>
<tr>
<td></td>
<td>Rationale: ‘id’ is a reserved word in Objective-C++, and against variable name rules. ‘min’, ‘max’, and less common identifiers listed in Testing/Core/WindowsMangleList.py are declared in the Windows API.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>26.</td>
<td>Prefer the use of <code>vtkNew</code> when the variable would be classically treated as a stack variable.</td>
</tr>
<tr>
<td>27.</td>
<td>Eighty character line width is preferred.</td>
</tr>
<tr>
<td>Rationale:</td>
<td>Readability</td>
</tr>
<tr>
<td>28.</td>
<td>Method definitions in implementation files should be preceded by <code>//</code> followed by 78 “-” characters.</td>
</tr>
<tr>
<td>Rationale:</td>
<td>Readability</td>
</tr>
<tr>
<td>29.</td>
<td>New code must include regression tests that will run on the dashboards. The name of the file to test <code>vtkClassName</code> should be <code>TestClassName.cxx</code>. Each test should call several functions, each as short as possible, to exercise a specific functionality of the class. The &quot;main()&quot; function of the test file must be called <code>TestClassName(int, char* [])</code></td>
</tr>
<tr>
<td>Rationale:</td>
<td>Code that is not tested can not be said to be working.</td>
</tr>
<tr>
<td>30.</td>
<td>All code must compile and run without warning or error messages on the nightly dashboards, which include Windows, Mac, Linux and Unix machines. Exceptions can be made, for example to exclude warnings from ThirdParty libraries, by adding exceptions to CMake/CTestCustom.cmake.in</td>
</tr>
<tr>
<td>31.</td>
<td>Namespaces should not be brought into global scope in any public headers, i.e. the &quot;using&quot; keyword should not appear in any public headers except within class scope. It can be used in implementations, but it is preferred to bring symbols into the global scope rather than an entire namespace.</td>
</tr>
<tr>
<td>Rationale:</td>
<td>Using VTK API should not have side-effects where parts of the std namespace (or the entire thing) are suddenly moved to global scope.</td>
</tr>
<tr>
<td>34.</td>
<td>While much of the legacy VTK API uses integers for boolean values, new interfaces should prefer the <code>bool</code> type.</td>
</tr>
<tr>
<td>Rationale:</td>
<td>Readability.</td>
</tr>
<tr>
<td>35.</td>
<td>Template classes are permitted, but must be excluded from wrapped languages.</td>
</tr>
<tr>
<td>Rationale:</td>
<td>The concept of templates doesn’t exist in all wrapped languages.</td>
</tr>
</tbody>
</table>
Specific C++ Language Guidelines

C++ Standard Library

<table>
<thead>
<tr>
<th>Do not use vtkStdString in new API; prefer std::string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rationale: vtkStdString was introduced as a workaround for compilers that couldn't handle the long symbol name for the expanded std::string type. It is no longer needed on modern platforms.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>STL usage is disallowed in the Common modules’ public API, and should only be used in implementation files there. The other modules may use STL, but should do so only when necessary if there is not an appropriate VTK class. Care should be taken when using the STL in public API, especially in the context of what can be wrapped.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exception: std::string should be used as the container for all 8-bit character data, and is permitted throughout VTK.</td>
</tr>
<tr>
<td>Rationale: limits header inclusion bloat, wrappers are not capable of handling many non-vtkObject derived classes.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>References to STL derived classes in header files should be private. If the class is not intended to be subclassed it is safe to put the references in the protected section.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rationale: avoids DLL boundary issues.</td>
</tr>
</tbody>
</table>

C++ Language Features Required when using VTK

<table>
<thead>
<tr>
<th>nullptr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use nullptr instead of 0 and NULL when dealing with pointer types</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>override</th>
</tr>
</thead>
<tbody>
<tr>
<td>VTK_OVERRIDE will be replaced with the override keyword</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>final</th>
</tr>
</thead>
<tbody>
<tr>
<td>VTK_FINAL will be replaced with the final keyword</td>
</tr>
</tbody>
</table>
**delete**

The use of delete is preferred over making default members private and unimplemented.

---

**C++11 Features allowed throughout VTK**

**default**

The use of default is encouraged in preference to empty destructor implementations.

**static_assert**

Must use the `static_assert ( bool constexpr , message )` signature. The signature without the message in C++17.

**non static data member initializers**

**strongly typed enums**

VTK prefers the usage of strongly typed enums over classic weakly typed enums.

Weakly typed enums conversion to integers is undesirable, and the ability for strongly typed enums to specify explicit storage size make it the preferred form of enums.

strongly typed: `enum class Color { red, blue }`;

weakly typed: `enum Color { red, blue }`;

While VTK is aware that conversion of all enums over to strongly typed enums will uncover a collection of subtle faults and incorrect assumptions. Converting existing classes to use strongly typed enums will need to be investigated and discussed with the mailing list, as this will break API/ABI, potentially cause issues with VTK bindings, and possibly require changes to users VTK code.
**C++11 Features acceptable in VTK implementation files, private headers, and template implementations**

**auto**

Use `auto` to avoid type names that are noisy, obvious, or unimportant - cases where the type doesn’t aid in clarity for the reader.

`auto` is permitted when it increases readability, particularly as described below. *Never initialize an auto-typed variable with a braced initializer list.*

Specific cases where `auto` is allowed or encouraged:

- **(Encouraged)** For iterators and other long/convoluted type names, particularly when the type is clear from context (calls to `find`, `begin`, or `end` for instance).
- **(Allowed)** When the type is clear from local context (in the same expression or within a few lines). Initialization of a pointer or smart pointer with calls to `new` commonly falls into this category, as does use of `auto` in a range-based loop over a container whose type is spelled out nearby.
- **(Allowed)** When the type doesn’t matter because it isn’t being used for anything other than equality comparison.
- **(Encouraged)** When iterating over a map with a range-based loop (because it is often assumed that the correct type is `std::pair<KeyType, ValueType>` whereas it is actually `std::pair<const KeyType, ValueType>`). This is particularly well paired with local key and value aliases for `.first` and `.second` (often `const-ref`).
  ```cpp
  for (const auto& item : some_map) {
    const KeyType& key = item.first;
    const ValType& value = item.second;
    // The rest of the loop can now just refer to key and value,
    // a reader can see the types in question, and we’ve avoided
    // the too-common case of extra copies in this iteration.
  }
  ```
- **(Discouraged)** When iterating in integer space. for (auto i=0; i < grid->GetNumberOfPoints(); ++i). Because vtk data structures usually contain more than 2 billion elements, iterating using 32bit integer is discouraged (and often doesn’t match the type used)
### braced initializer list

Braced initializer list are allowed as they prevent implicit narrowing conversions, and "most vexing parse" errors. They can be used when constructing POD’s and other containers.

**Braced initializer lists are not allowed to be used as the right hand side for auto:**

```cpp
auto a = { 10, 20 }; // not allowed as a is std::initializer_list<int>
```

### lambda expressions

Usage of lambda expressions are allowed with the following guidelines.

- Use default capture by value ([=]) only as a means of binding a few variables for a short lambda, where the set of captured variables is obvious at a glance. Prefer not to write long or complex lambdas with default capture by value.
- Except for the above, all capture arguments must be explicitly captured. Using the default capture by reference ( [&] ) is not allowed. This is to done so that it is easier to evaluate lifespan and reference ownership.
- Keep unnamed lambdas short. If a lambda body is more than maybe five lines long, prefer using a named function instead of a lambda.
- Specify the return type of the lambda explicitly if that will make it more obvious to readers.

### shared_ptr

- **Do not combine shared_ptr and vtk derived objects.** Vtk internal reference counting makes the shared_ptr reference counting ( and destructor tracking ) pointless.

### unique_ptr

- **Do not combine unique_ptr and vtk derived objects.** We prefer using vtkNew as VTK objects use internal reference counting and custom deletion logic, the ownership semantics of unique_ptr are invalid.
  - Make_unique is not part of c++11

### template alias

The use of alias templates is prefered over using ‘typedefs’. They provide the
same language pattern of normal declarations, and reduce the need for helper template structs. For example (Scott Meyers, Effective Modern C++)

```cpp
template<typename T> using MyAllocList = std::list<T, MyAlloc<T>>;
```

universal references (&&) / std::move / std::forward

**extern templates**

Note: This should be investigated as an update to the current infrastructure used to export explicit template instantiations used within VTK

**unordered maps**

**std::array**

The use of std::array is preferred over using raw fixed sized arrays. They offer compile time bounds checking without any runtime cost.

**range based for loop**

---

### C++11 Features allowed under certain conditions

**concurrency**

Concurrency inside of vtk should be handled by using or extending the already existing collection of support classes like vtkAtomic and vtkSMPTThreadLocal.

Instead of directly using new c++11 constructs such as std::compare_exchange_weak instead extend the functionality of vtk core concurrency classes.

Note: Thread local storage has not been supported on OSX previously to XCode 8. VTK offers the following classes that should be used instead:

- [vtkSMPTThreadLocalObject](#)
- [vtkSMPTThreadLocal](#)
These functions should not be called directly, instead the wrapped versions provided by vtk should be used instead.

- vtk::isnan -> std::isnan
- vtk::isfinite -> std::isfinite
- vtk::isinfnan -> std::isinf

The reason for these wrappings is to work around compiler performance issues. For example, some clang version would convert integral types to double and do the operation on the double value, instead of simply returning false/true.

Future/Async based programming inside of vtk should be handled on a case by case basis. In general the use cases for this kind of execution model is best applied at the vtkExecutive / vtkPipeline level, or at the File IO level.

In these cases the recommendation is to extending or adding support classes so that these design patterns can be utilized in the future.

Variadic Templates are not allowed in VTK unless they are the only solution to the given problem.

---

**C++11 Features that are not allowed**

std::regex
<table>
<thead>
<tr>
<th>Feature</th>
<th>Compatibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>constexpr</td>
<td>Not supported by VS2013</td>
</tr>
<tr>
<td>unicode string literals</td>
<td>Not supported by VS2013</td>
</tr>
<tr>
<td>universal character names in literals</td>
<td>Not supported by VS2013</td>
</tr>
<tr>
<td>user-defined literals</td>
<td>Not supported by VS2013</td>
</tr>
<tr>
<td>Extended sizeof</td>
<td>Not supported by VS2013</td>
</tr>
<tr>
<td>Unrestricted Unions</td>
<td>Not supported by VS2013</td>
</tr>
<tr>
<td>Noexcept</td>
<td>Not supported by VS2013</td>
</tr>
</tbody>
</table>

**Table 1: coding style with rationale**

Parts of this coding style are enforced by git commit hooks that are put in place when the developer runs the SetupForDevelopment script, other parts are enforced by smoke tests that run as part of VTK’s regression test suite. Most of these guidelines are not automatically enforced. Tables 2 and 3 list the style checks that are in place.
**HeaderTesting** - Testing/Core/HeaderTesting.py

A static test of header files for a variety of style issues including rules 4, 5, 6, 14, 15, 16, 20, 23). This test is enabled whenever TESTING is on and python is available. ThirdParty modules avoid the test by marking themselves with the NO_HeaderTest property. Individual files can be exempted by ending the file with: // VTK-HeaderTest-Exclude: filename.h

**TestSetObjectMacro** - Testing/Core/FindString.tcl

A static test of header files that looks for uses of the discouraged SetObject macro in header files (rule 18). Test is enabled whenever TESTING is on and Tcl is available.

**PrintSelf** - Testing/Core/TestPrintSelf

A dynamic test that checks that every class reports on the status of all of its instance variables when the PrintSelf method is called. Test is enabled whenever TESTING is on and Tcl is available.

**TestSetGet** – Removed in vtk6, but will return.

A dynamic test that verifies that all methods have proper accessors and are not stateful. Thus classes will work the same regardless of the order that they are called in and not only in some order that the developer had in mind.

**TestEmptyInput** – Removed in vtk6, but will return.

A dynamic test that verifies that algorithms do not crash when they are given an empty input.

| Table 2: VTK’s Smoke Tests |

**Well formed commit message**

Every commit message should consist of a one line summary optionally followed by a blank line and further details. This is most easily approximated to the subject of an email, and the body in the form of paragraphs.

**Valid committer username and email address**
Every developer must have a valid name and email configured in git.

<table>
<thead>
<tr>
<th>ASCII filename check</th>
</tr>
</thead>
<tbody>
<tr>
<td>All file names must contain only ASCII characters.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>No tabs</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>No trailing whitespace</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>No empty line at end of file</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Proper file access mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Files must be committed with sensible access modes.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>One megabyte maximum file size</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>No submodules</th>
</tr>
</thead>
<tbody>
<tr>
<td>The VTK project does not allow submodules. For required third party dependencies, the recommended scheme is to use git’s subtree merge strategy to reproducibly import code and thereby simplify eventual integration of upstream changes.</td>
</tr>
</tbody>
</table>

**Table 3: VTK’s commit hook enforced style checks**

Additionally, new developers should be aware that the regression test machines have fairly strict compiler warnings enabled and usually have VTK_DEBUG_LEAKS configured on to catch leaks of VTK objects. Developers should be in the habit of doing the same in their own environments so as to avoid pushing code that the dashboards will immediately object to. With GCC, it is easiest to do so by turning on VTK_EXTRA_COMPILER_WARNINGS.

**Contributing Code**

VTK’s development history is stored in a git server running at Kitware, and
mirrored elsewhere. We maintain two long-lived branches, master and release. Only the release manager has permissions to push changes onto to the release branch. On the master branch, all changes go through Kitware’s Gitlab source code review system instance. Gitlab is a full git hosting instance with a web interface through which developers can discuss proposed changes. Kitware’s Gitlab instance is tied to buildbot regression testing servers that compile and run VTK’s regression tests on each change from trusted developers. The core group of trusted developers also has permission to merge new code from Gitlab onto the master branch. Developers not in the core group can push their changes to the Gitlab server to share it with the the community who can review, test and merge the change.

All code is contributed through Gitlab as doing so automates and thus greatly reduces the time and effort needed to validate each change. Merge requests can request buildbot builds and regression tests on Windows, Mac, and Linux volunteer machines. Code contributed by others is tested whenever someone in the core group adds a review comment that says “Do: test”. After the tests complete, anyone in the core group can merge the change into the master branch by positively reviewing the code and commenting “Do: merge”.

Step by step instructions on how to commit changes to VTK through Gitlab are maintained as part of the VTK source code: https://gitlab.kitware.com/vtk/vtk/blob/master/Documentation/dev/git/develop.md. Tips for using Git and Gitlab include:

1. To determine who you should ask to review your code, type “git log --follow” in your source tree. The most recent developers to work on a file you change are usually the best reviewers.

2. To share a change that you are not finished with, and do not want merged into master, start your commit message with “WIP:” for work in progress. Alternatively you can give yourself a -2 review score.

3. If you are a core developer and want to test an externally contributed change, make a comment of “Do: test” on the MR.

4. To merge a change, make a comment of “Do: merge” on the MR.

5. Self review of code is not allowed. Only rarely should you merge your own topics before another developer gives you a +2 score. Repeat offences will result in removal of your merge privileges.

6. Make sure the author knows when their work is merged into master so that they know when to pay special attention to the continuous and nightly dashboards. For work from external developers, the developer who did the merge should send an email and help look for problems. Core developers
should not merge each other’s changes.

7. Topics should not be overly long nor entirely compressed. It is best to consolidate your work into a handful of substantial changes that are functional on their own. This helps others understand and review your work and eases the task of keeping it up to date with master. At the same time it is not good practice to habitually squash topics into single commits, as this obscures too much of the thinking that went into the final result.

8. Gitlab offers multiple ways to associate Issues and Merge Requests.

After a change is merged into the master branch, the changes are further tested as described in the Regression Testing section below. It is important that every developer watches the dashboard, particularly immediately following and on the morning after their topics are merged, to ensure that the code works well on all of the platforms that VTK supports.

It is recommended that the VTK Architecture Review Board (ARB) is consulted before major changes impacting backwards compatibility are developed. The ARB is a consortium of interested parties that help to direct the evolution of VTK. Details can be found at: http://www.vtk.org/Wiki/VTK/ARB. If you are unsure if your change should be classified as substantial, it is a good idea to send an email to the VTK developers list and explain what you intend to do.

It is especially important that substantial backwards incompatible changes are discussed with and agreed upon by the community. Minor backwards incompatible changes are allowed in general with the following deprecation policy. Code that is to be removed is marked using the VTK_LEGACY macros in Common/Core/vtkSetGet.h and with the use of @deprecated Doxygen comments in the header files. Deprecated code can then be removed after a minimum of one minor release, i.e. deprecated and released in 6.1, can be removed after 6.1 for the 6.2 release. Backward incompatible changes should be kept to a minimum, we aim for source but not ABI compatibility between minor releases.

Regression Testing

Testing and dashboard submitter setup

Regression testing in VTK takes the form of a set of programs, that are included in the VTK source code and enabled in builds configured through CMake to have the BUILD_TESTING flag turned on. Test pass/fail results are returned to CTest
via a test program’s exit code. VTK contains helper classes that do specific checks, such as comparing a produced image against a known valid one, that are used in many of the regression tests. Test results may be submitted to Kitware’s CDash instance, where they will be gathered and displayed at [http://open.cdash.org/index.php?project=VTK](http://open.cdash.org/index.php?project=VTK).

All proposed changes to VTK are automatically tested on Windows, Mac and Linux machines as described in the Contributing Code section above. All changes that are merged into the master branch are subsequently tested again by more rigorously configured Windows, Mac and Linux continuous dashboard submitters. After 9PM Eastern Time, the master branch is again tested by a wider set of machines and platforms. These results appear in the next day’s page.

At each step in the code integration path the developers who contribute and merge code are responsible for checking the test results to look for problems that the new code might have introduced. Plus signs in CDash indicate newly detected problems. Developers can correlate problems with contributions by logging in to CDash. Submissions that contain a logged in developer’s change are highlighted with yellow dots.

It is highly recommended that developers test changes locally before submitting them. To run tests locally:

1. **Configure with BUILD_TESTING set ON**
   
   The exact set of tests created depends on many configuration options. Tests in non-default modules are only tested when those modules are purposefully enabled, the smoke tests described in the Coding Style section above are enabled only when the python or Tcl interpreter is installed, tests written in wrapped languages are only enabled when wrapping is turned on, etc.

2. **Build.**
   
   VTK tests are only available from the build tree.

3. **Run ctest at the command line in the build directory or make the TESTING target in Visual Studio.**
   
   As ctest runs the tests it prints a summary. You should expect 90% of the tests or better to pass if your VTK is configured correctly. Detailed results (which are also printed if you supply a \(-v\) argument to ctest) are put into the Testing/Temporary directory. The detailed results include the command line that ctest uses to spawn each test. Other particularly useful arguments are:

   \[-R\ \text{TestNameSubstringToInclude}\] to choose tests by name

   \[-E\ \text{TestNameSubstringToExclude}\] to reject tests by name

   \[-I\ \text{start,stop,step}\] to run a portion of the tests

   \[-j\ N\] to run \(N\) tests simultaneously.

Dashboard submitting machines work at a slightly higher level of abstraction that adds the additional stages of downloading, configuring and building VTK before running the tests, and submitting all results to CDash afterward. With a build tree in
place you can run “ctest –D Experimental” to run at this level and submit
the results to the experimental section of the VTK dashboard or “ctest –M
Experimental –T Build –T Submit” etc to pick and choose from among the
stages. When setting up a test submitter machine one should start with the
experimental configuration and then, once the kinks are worked out, promote
the submitter to the Nightly section.

The volunteer machines use cron or Windows task scheduler to run CMake
scripts that configure a VTK build with specific options, and then run ctest –D as
above. Within CDash, you can see each test machine’s specific configuration by
clicking on the Advanced View and then clicking on the note icon in the Build Name
column. This is a useful starting point when setting up a new submitter. It is
important that each submitter’s dashboard script include the name of the person
who configures or maintains the machine so that, when the machine has problems,
the dashboard maintainer can address it.

At Kitware, the dashboard scripts are maintained in a separate git repository
and generally include a common script. The common script facilitates the task of
setting up the submitter. See http://www.vtk.org/Wiki/VTK/Git/Dashboard
for more information.

Interpreting the Dashboard Results

The dashboard presents all submitted results, for the present day, in a
tabular format. Kitware’s CDash server keeps the last four months worth of results
which you can browse through via the web interface. On any given day, rows in the
table are results for one particular submission from one particular submitting
machine. Columns in the table show each machine’s identity and results for the
download, configure, build and test stages. Good results show as green, bad results
show as red and in between results are shown in orange. (In the next CDash release
the color scheme will be configurable in the browser.) On any result you can click
through to drill down and get more detail.

<NOTE The following are out of data. We need to take out references to removed
sections and add the new buildbot CI ones.>

The rows in the table are organized into sections. These are:

- **Nightly Expected**
  Nightly results from reliable and trusted machines. All developers pay
close attention to this section to see that their changes work on an
assortment of platforms and configurations.

- **Continuous**
Daily results that test newly merged changes into the master branch. This is an intermediate step between quick gitlab/buildbot tests and the full nightly test gamut.

- **Dependencies**
  Daily results from machines that build VTK with development versions of VTK’s dependencies: CMake, Mesa, etc. Failures in this section indicate to the developers that VTK may need to change to keep pace with them.

- **Nightly**
  Nightly results from unproven machines. If and when persistent problems with a nightly submitter are fixed, and the responsible person agrees to maintain it, the dashboard manager can promote it to the Nightly Expected section so that more developers will pay close attention to it.

- **Merge-Requests**
  This is where results from proposed Merge Requests on Gitlab go. It is generally more useful to view these results directly though the specific Gitlab’s Merge Request page.

- **Experimental**
  Submissions from new machines and tests of experimental code can go here. This is good place to test code that has not been merged into the mainstream.

- **Release Branch**
  Nightly results that test the release branch, this is mostly of interest to the release manager during the release candidate cycle.

- **Release-X Branch**
  Nightly results that test an older release branch. This is currently being used to ensure that the 5.10 branch remains functional while people adopt 6.x.

- **Coverage**
  This section contains results from machines that have been configured to do code coverage counts that count the number of lines in the VTK source that are exercised or missed by the regression test suite.

- **Dynamic Analysis**
  This section contains results from machines that have been configured to run the regression tests through a memory checker to expose memory leaks and harder to find code problems.
In the Utilities/Maintenance directory of the VTK source code there are a few helper scripts that present alternative views of the data collected by CDash, and one script that is useful when setting up a new memory checking submission. These are:

- `parse_valgrind.py` - eases the task of setting up a memory checking submitter.
- `vtk_fail_summary.py` – presents a test centric view of failures so that problematic tests can be readily identified.
- `vtk_submitter_summary.py` – determines the configuration of each submitter to help determine under-tested platform and feature sets and correlate failures with causes. This is very useful when trying to tease apart the different roles that each of the dashboard submitters fulfill.
- `vtk_site_history.py` – presents a time centric view of failures to gauge project and dashboard health with.
- `computeCodeCoverage*.sh` – presents coverage results in a more intuitive and helpful way than CDash currently does.

**Writing new tests**

All new features that go into VTK must be accompanied by tests. This ensures that the feature works on many platforms and that it will continue to work as VTK evolves.

Tests for the classes in each module of VTK are placed underneath the module’s Testing/<Language> subdirectory. Modules that the tests depend upon beyond those that the module itself depends upon are declared with the TEST DEPENDS argument in the module.cmake file. Test executables are added to VTK’s build system by naming them in the CMakeLists.txt files in each Testing/<Language> directory. In those CMakeLists, standard `add_executable()` + `add_test()` command pairs could be used, but the following macros defined in vtkTestingMacros.cmake are preferable as they consolidate multiple tests together, participate in VTK’s modular build scripts, and ensure consistency:

- `vtk_add_test_cxx(name.cxx [NO_DATA] [NO_VALID] [NO_OUTPUT])` adds a test written in C++
  NO_DATA indicates that this test doesn’t require any input data files
  NO_VALID indicates that this test doesn’t compare results against
baseline images
  NO_OUTPUT indicates that the test doesn't produce any output files

- vtk_add_test_mpi(name [TESTING_DATA])
  adds a test which should be run as an MPI job.
  TESTING_DATA indicates that this test looks for input data files and
  produces regression test images.

- vtk_add_test_python(name [NO_RT] [NO_DATA] [NO_VALID] [NO_OUTPUT])
  adds a test written in python
  NO_RT indicates that the test won't use the image comparison helpers
  from vtk.test.testing

- vtk_add_test_tcl(name [NO_RT] [NO_DATA] [NO_VALID] [NO_OUTPUT])
  NO_RT means that your test won't use the image comparison helpers
  from rtImageTest.tcl

Tests indicate success to CTest by returning EXIT_SUCCESS (0) and failure by
returning EXIT_FAILURE (1). How the test determines what result to return is up to
the developer. VTK contains a number of utilities for this task. For example,
vtkRegressionTester is a helper class that does a fuzzy comparison of images drawn
by VTK against known good baseline images and returns a metric that can be simply
compared against a numeric threshold.

(Write and link to a source article that describes them in enough detail so
that adding new tests is an entirely rote process.)

Many tests require data files to run. The image comparison tests for example
need baseline images to compare against, and many tests open up one or more files
to visualize. In the past, test data was kept in two external repositories. One each for
small and large data files. Since VTK 6.1.0, data is placed on a public web server
automatically when topics are pushed, and copied to a Midas
http://www.midasplatform.org instance at release time. The add_test macros
mentioned above use CMake to download the version of the data file that exactly
matches the version needed by the source code during the build process.

The source code and data file versions are kept in sync because the
Testing/Data directory contains, instead of the real files, similarly named files which
contain only the MD5 hash of the matching data files. During the build process,
when CMake sees that a required data file is not available, it downloads it into the
directory defined by the ExternalData_OBJECT_STORES cmake configuration
entry. The test executables read all data from there. The default setting for
ExternalData_OBJECTSTORES is the ExternalData directory underneath the VTK
To make a change to VTK that modifies or adds a new test data file, place the new version in the Testing/Data or directory (for input data files) or Module/Name/Testing/Data (for regression test images), and build (or run cmake). CMake will do the work of moving the original out of the way and replacing it with an MD5 link file. When you push the new link file to Gitlab, git pre-commit hooks push the original file up to Kitware's data service, where everyone can retrieve it.

Step by step descriptions for how to add new test data are given here: http://www.vtk.org/Wiki/VTK/Git/Develop/Data

**Bug and Feature Tracking**

Despite all of this software quality process, people *occasionally* find bugs in VTK. Bugs should be reported to the developers so that they might be addressed in future releases. Bugs should be reported first on the user’s mailing list for discussion. In a technically sophisticated area like visualization, user error is always possible. If people on the list confirm that the problem is in fact a new bug, or if no one on the list responds to the report, then please enter the bug into the bug tracker.

*NOTE: The following is out of date - we now use issues within gitlab>*

We track bugs in the mantis bug tracker instance at: http://vtk.org/Bug/my_view_page.php. As always proper netiquette is valued. First search to see if someone else has already reported the bug. Second, realize that there is no guarantee that a reported bug will be corrected promptly or at all. In the event that no one takes interest in your bug know that Kitware offers paid support options that you can find at http://www.kitware.com/products/support.html. Third, the most important thing to try to do in the report is to make it easy to reproduce the problem.

Feature requests should not be made through the bug tracker. The proper channels for feature requests include the the mailing list, Kitware’s paid support options (refer to http://www.kitware.com/products/support.html), and User Voice. User Voice is a feature voting web service that you can find at: http://vtk.uservoice.com/forums/31508-general. From time to time, we take the requested features into account when we decide which direction to steer VTK.

Feature requests made on the bug tracker are handled by directing the reporter to the above resources.

Patches are likewise discouraged in the bug tracker. The effort necessary to
validate and integrate suggested changes is rarely insignificant. Patch requests made via the issue tracker or mailing list are handled by directing the reporter to Gitlab.

When reporting a bug the important things to describe are:

- **Summary**: A concise description of the problem. Always try to include searchable terms in the summary so that other reporters may find it later.

- **Description**: It is essential to describe the problem in enough detail to reproduce it. Ideally the description should have URLs or attachments that contain concise runnable code samples and any small data files needed to reproduce it. Links to Gitlab reviews and email discussions are extremely helpful. The perfect way to demonstrate a problem is to make the demonstration into new test and submit it via Gitlab. This make it trivial to reproduce and, once merged will ensure that the problem remains fixed.

- **Found in Version** (available only once the bug is submitted)
  What VTK release was the problem detected in. Versions ending in dev and rc have special meaning. "dev" indicates that the bug was found in the master branch and not on a released version. le 6.0.dev means that the bug was found in the master branch sometime after 6.0.0 release. The fix for this might end up in 6.1.0, and possibly in 6.0.1 as well. "rc" indicates that the bug was found during the release candidate testing cycle. The release manager uses it to track problems and related Gitlab merge requests that need to be addressed before the release is finalized. The related **Target Version** and **Fixed in Version** fields exist to help plan what should be fixed in upcoming versions and to keep track of what actually was fixed in past versions.

- **OS and platform** (available once the bug is submitted)
  This is sometimes necessary to reproduce the problem.

Reporters can generally ignore the rest of the fields. The **Project** and **Type** fields are largely ignored in VTK but are used in other projects. The **Priority** and **Assign To** fields are for developers and the release manager to use while addressing reported issues.

When a new bug report is made, mantis sends it off to the developer’s mailing list. Meanwhile, the release manager reads all newly submitted bugs and asks developers who are familiar with the area of VTK that the bug relates to to do a cursory review. These developers may or may not decide to take the bug up. At VTK release time, the release manager reviews all bugs, decides if any of them are critical to fix before the release, and ensures that they are addressed. Bugs that remain in the bug tracker for more than one year will be expired. When bugs are fixed, the
fixes happen in the master branch and generally only appear in subsequent minor releases.

The **Status** field in the bug tracker may be in one of several states, depending on where a bug is along the path to being fixed. The bug states are a subset of those in ParaView’s Kanban inspired software process, in which pools of new work are taken up by various developers and pursued until the requesting customer is satisfied with their resolution. Details of the ParaView’s software process state machine are given here: [http://paraview.org/ParaView/index.php/Development_Workflow](http://paraview.org/ParaView/index.php/Development_Workflow)

VTK has a more dispersed set of contributors and less well defined developer roles and funding sources. As such we simplify to the following states and transitions.

- **backlog** - All bugs start off in this general pool of work.
- **active development** - When a developer begins working on the bug he or she should move it into the active development state to indicate that work has begun.
- **gitlab review** - When a developer pushes a fix for the problem into gitlab to be reviewed he or she should move it into the gitlab review state to indicate that a fix is ready for testing.
- **closed** - When the code has been fixed, reviewed, merged into master, and the dashboards are green, the developer moves it to the closed state and sets the Fixed in Version to the current `dev` version. Closed bugs, if found to be broken, can be reopened, in which case they go back to the backlog state.
- **expired** - When the bug has not gathered attention for a long time it moves into the expired state. Expired bugs can be reopened and placed back in the backlog.
Documentation

The VTK FAQ (http://www.vtk.org/Wiki/VTK_FAQ), examples repository (http://www.vtk.org/Wiki/VTK/Examples) and overall Wiki are community efforts (http://www.vtk.org/Wiki/VTK) and open to all. Only rarely is there a concerted effort made by the community to keep them up-to-date.

The Examples Wiki is regression nightly by at least one Linux volunteer machine and verifies that the examples work on the most recent VTK releases. The results can be found at: http://open.cdash.org/index.php?project=VTKWikiExamples. Note that fewer developers pay close attention to these results than to VTK’s own regression test results, so it is a good idea to warn the mailing list when problems are found.

Kitware updates and releases new versions of the VTK Textbook and User’s guide every few years as demand calls for them to be.