Vurtigo Design White Paper

For Vurtigo Version 3.2

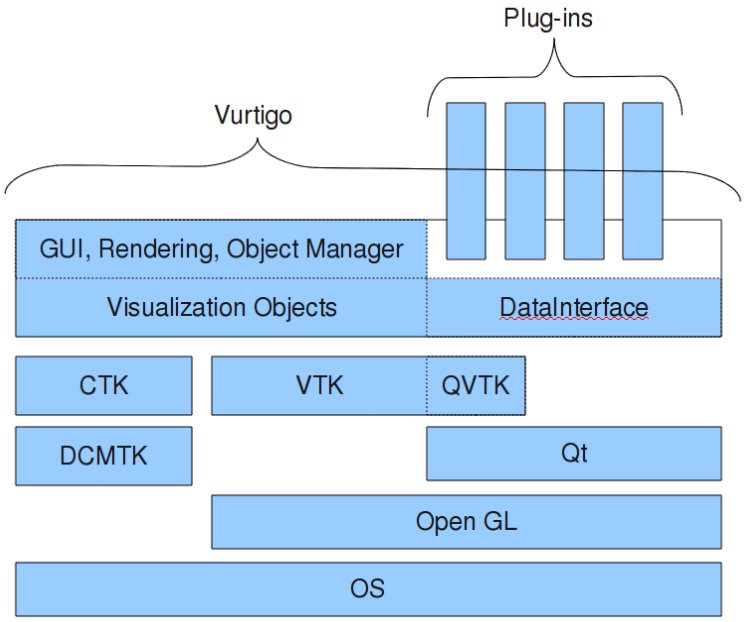
Table of Contents

Introduction

This design document describes Vurtigo, a four-dimensional (3D + time) real-time visualization software. Vurtigo is written in C++ and uses a plugin-based architecture to implement much of the functionality. The application is composed of two pieces: a base, and a series of plugins. It is a cross-platform program, though recent work has focused on supporting the Linux platform.

The base of Vurtigo maintains a list of **visualization objects** that both the user and the plugins can interact with. Visualization objects are entities that can be loaded and saved by Vurtigo that contribute in some way to the scene. Some of these visualization objects can be rendered in 3D, some in 2D, and some may not be rendered at all since they have no visual representation. For example, a 3D volume can be rendered both in 3D as a volume and in 2D as a cut plane through the volume. Another visualization object, a colour function, cannot be rendered in 3D or 2D but affects the colour schemes for other visualization objects.

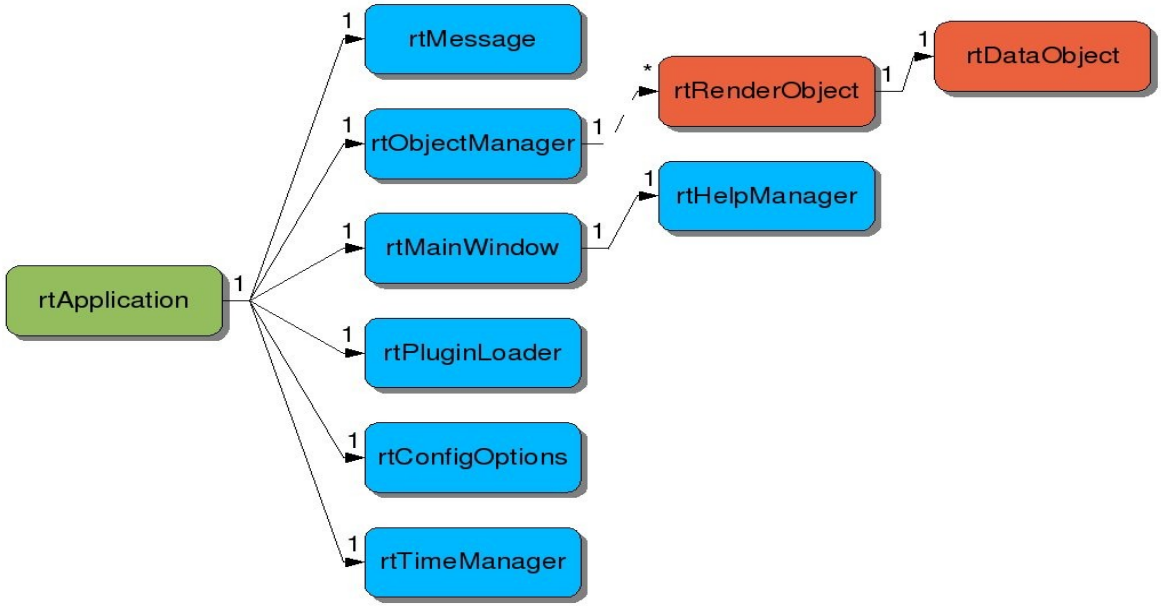
Dependency Diagram



Vurtigo depends on: CTK for DICOM handling, VTK for rendering, and Qt for the GUI. The QVTK component allows the VTK render window to be placed into the Qt GUI. Both Qt and VTK make use of the OpenGL library. All of Vurtigo's plugins have one class derived from the DataInterface class.

Vurtigo Base

The Vurtigo base is composed of several manager classes each of which is only instantiated once in the execution of the program. They are not singleton classes and so the unique instance is not strictly enforced. However, all of these manager classes are created exactly once, either directly, or indirectly from the singleton class rtApplication. The collaboration diagram for rtApplication is displayed below:



The Managers (all of which can be seen above) are as follows

rtApplication – The main handle for the entire application.

- *rtPluginLoader*: Responsible for loading all of the plugins for Vurtigo. Uses QPluginLoader.
- *rtMainWindow*: The main Vurtigo window that contains the visualization object list, the list of loaded plugins and the 2D/3D windows. The main window also controls two other important classes:
  - rtHelpManager. Provides support for loading help documents that would be shown to the user. Uses QHelpEngine.
- *rtConfigOptions*: Loads the initial configuration options from an XML file.
- *rtTimeManager*: Maintains the global application clock and is also responsible for triggering timed events.
- *rtMessage*: This class is a way for Vurtigo to output different types of messages; error, debug, warning, bench, debug.
- *rtObjectManager*: Is responsible for the creation, deletion and general maintenance of all of the visualization object (rtRenderObject, rtDataObject) lists within Vurtigo.

Visualization Objects

Within Vurtigo there are 11 different types of Visualization objects. These objects are created and deleted by the rtObjectManager class. Each visualization object has a data part which is derived from rtDataObject, and a rendering part which is derived from rtRenderObject. The purpose of the data part of the object is to store the information required to define a particular rendering. For example, to render a point in 3D space one would need the position of the point in (x, y, z), the radius of the sphere that represents the point, the colour of the sphere, the transparency of the sphere, etc. The purpose of the render object is to get new information from the data object and transform that into a VTK pipeline that will render the object. The render object is tasked with creating all of the pipelines necessary in both 2D and 3D to render the object correctly.

The different types of objects are:

Visualization Object	Render
Collection of 3D Points. Marking points in 3D space.	Yes
A 2D Slice. A representation of a 2D image.	Yes (Use a default position if no 3D position is specified)
A 3D/4D Volume.	Yes
A Catheter with one or more coils	Yes
A Colour Function. Used to add colour to visible objects.	No
A Piecewise Function. Used to specify transparency to visible objects.	No
Label text. Text to be rendered.	Yes
Transformation matrix. Specify a position by which an object may be moved.	No
Polygonal surface.	Yes

2D X-Y Plot	Yes
Region of Interest (ROI). A collection of points defining a closed contour.	Yes

Data Object

The following is a set of important properties common to all Data Objects:

- *Object ID*: Unique ID for this instance of this object.
- *Object Name*: A string that can be used as a name for this object. Not unique.
- *Object Type*: Specifies one of the above 13 types of objects.
- *Base Widget*: All object types (Except 'None') have a GUI that the user can manipulate to adjust settings for the object. This GUI is placed directly on top of the base widget.
- *Modified Time*: The time stamp of the last modification of the data object. The time stamp can be reset with Modified() and can be queried with getModified(). The render object will check this flag to determine if modifications to the rendering are required.
- *Object Lock*: Objects should be locked if they are being modified outside the main rendering thread. For example, plugins should lock a data object before modifying the contents.

Render Object

The following is a set of important properties common to all Render Objects:

- *Data Object*: A pointer to the data object that this renderer is paired with.
- *Can Render in 3D* and *Is Rendering in 3D*: Two boolean variables that determine if this object can be rendered in 3D.
- *Visibility Hash*: To determine if the object is currently visible in each of the render windows.
- *Tree Item*: A graphical list item that is placed in the object list so that the user may be able to select this object.
- *Last Update Time*: The time stamp for the last update from the Data Object. Can be compared to the current data object time stamp to determine if another update is required.
- *2D Pipeline Hash*: A table that relates string names with individual 2D pipelines for this object. The strings must be unique within the same object. The idea is for the user to select one type of 2D output for this object and then that type name will be used to retrieve the string.
- *Selected Prop*: A handle to the selected prop for the current object. A vtkProp is a basic form of actor on the 3D scene. The selected prop handle is usually NULL but when the user clicks on an object in the scene the selected prop can be set. The prop is then used in the manipulation of that object.

Vurtigo Plugins

Since the Vurtigo base is only responsible for the storage and rendering of different visualization objects, much of the functionality within Vurtigo is provided by plugins. All plugins are brought into the application via the rtPluginLoader class which is given an XML file that lists the specifications for one or more plugins. The loader will then load each plugin from the file and give each a unique ID. Loaded plugins will be listed in the plugin browser in the main window. All plugins must include a class derived from the DataInterface class which will be used as a way for Vurtigo to load and control the plugin.

A plugin can optionally define a new Vurtigo visualization object (which can be used by Vurtigo or other plugins once the plugin has been loaded), by deriving a class from both rtRenderObject and rtDataObject and registering the new object with Vurtigo using the rtBaseHandle. See the HelloWorld plugin for an example of how to implement this.

Some important elements of the DataInterface class:

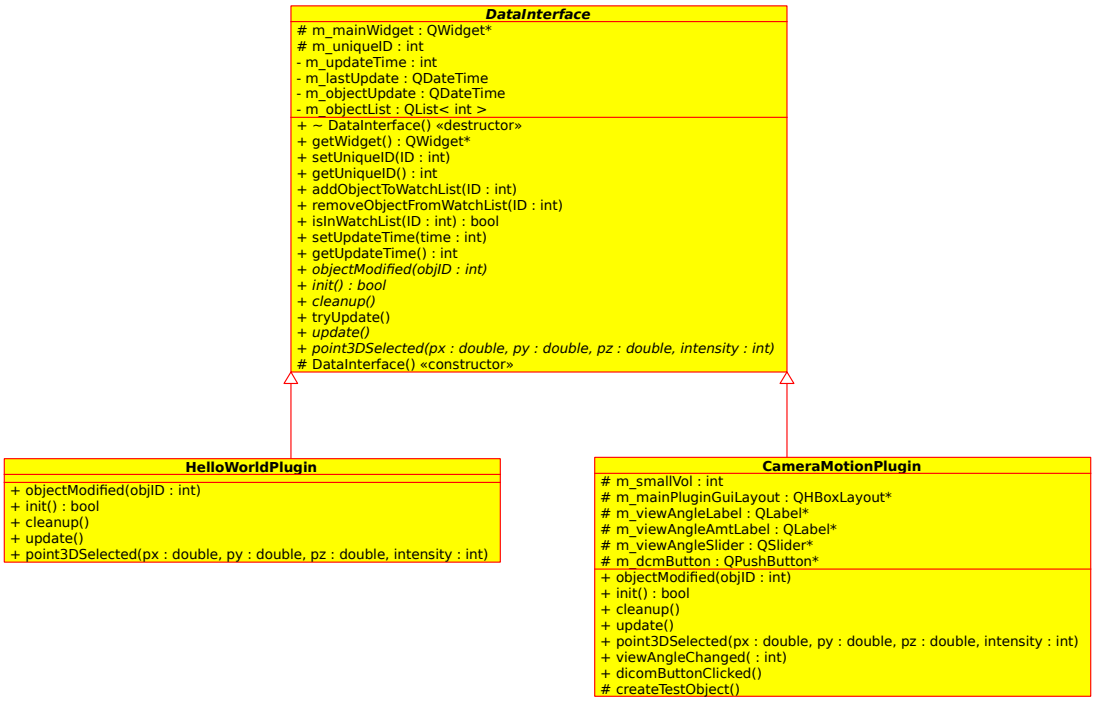
- *Unique ID:* Provides Vurtigo with access to the unique ID of this plugin. This is implemented within the DataInterface base class and the plugin developer should not overwrite it.
- *Watch List:* Plugins may place certain objects on a watch list by specifying the visualization object's unique ID. When the data for that object changes then the plugin is notified.
- *Main Widget:* Each plugin can have a set of user interface controls. These interface controls are placed on the main widget. When a plugin is loaded into Vurtigo the main widget is automatically loaded into the Vurtigo interface such that when the user clicks on the plugin the custom interface will appear.

There are two ways for a plugin to request services from the Vurtigo Base.

1. The rtApplication object: This is a singleton class that can be accessed from anywhere in the code and it gives direct access to all of the managers in Vurtigo. Even though it is not strictly enforced in code, it is not intended for plugins to use this class to obtain access. Plugins that use this method of interaction must be careful not to disrupt the internal workings of the Vurtigo base. For example, thread safety is not guaranteed for plugins as Vurtigo does not expect certain manager functions to be called from different threads.
2. The rtBaseHandle object: This is a singleton class that was made specifically for plugins. Plugins that only use the base handle are guaranteed a certain level of thread safety. (Unless specified in the function docs.) Creating and deleting objects from a plugin through this handle is considered thread safe.

Either one or both of these ways for interacting with Vurtigo can be used.

Parts of the Vurtigo source include the “Hello World” and “Camera Motion” plugins that are examples and starting points for plugin implementation. The class diagram below shows the sample plugins and their inheritance from the DataInterface.



Camera Motion Plugin (An Example)

Vurtigo plugins need two basic components. The first is an XML file that Vurtigo uses to get information about the plugin before loading it. The second component is the C++ class that provides the link between Vurtigo and the plugin. That linking class should implement the DataInterface (provided with Vurtigo) and should also be a subclass of QObject in order to be able to use signals and slots.

This section will use a simple plugin called the Camera Motion Plugin. This plugin is not particularly useful as it only allows the user to change the view angle for the camera and launch the dicom database tool. It is only meant as an example of how a plugin can be written. The plugin is composed of three files: CameraMotionConfig.xml, CameraMotionPlugin.h, CameraMotionPlugin.cpp.

The XML files for Vurtigo are all similar and the following template can be used:

CameraMotionConfig.xml

```
<plugin>
 <name title="Camera Motion" version="1.0"/>
 <lib libname="CameraMotion" libpath="/path/to/plugin/lib/"/>
</plugin>
```

Multiple plugins can also be grouped into a single XML file. Vurtigo can then load all of those plugins at the same time.

MultiplePluginConfig.xml

```
<PluginList>
 <plugin>
  <name title="Camera Motion" version="1.0"/>
  <lib libname="CameraMotion" libpath="/path/to/plugin/lib/"/>
 <\plugin>
 <plugin>
  ...
```

```
 <\plugin>

 ...

</PluginList>
```

The title and version are displayed by Vurtigo as information to the user in the plugin list. These strings are not parsed by Vurtigo. The libname is the name of the library minus all of the OS specific endings (omit endings such as .dll, .so, .dylib). If an operating system adds 'lib' to the front of the library name (as Linux does) that should also be omitted. The libpath is the path where Vurtigo will try to find the library. If the library is not found in the specified path, or the path is left empty, then the default system paths will be searched.

All plugins must also have one class derived from both QObject and DataInterface. This is the class that enables Vurtigo to load the plugin. The following is an example of a header file from such a class.

CameraMotionPlugin.h

```
#ifndef CAMERA_MOTION_PLUGIN_H
#define CAMERA_MOTION_PLUGIN_H

#include "rtPluginInterface.h"
#include "CompatibilityWithQt.h"

// Qt includes
#include <QHBoxLayout>
#include <QLabel>
#include <QSlider>
#include <QPushButton>

//! Plugin to control certain movements of the camera.
class CameraMotionPlugin : public QObject, public DataInterface {

// Need to use Q_OBJECT so that this class can use signals and slots.
Q_OBJECT
// This class implements the 'DataInterface'.
Q_INTERFACES(DataInterface)
V_PLUGIN_METADATA(IID "CameraMotionPlugin" FILE "CameraMotionPlugin.json")

public:
//! This function is called when an object in the base is modified.
/*!
The plugin can register to have certain objects as part of a waatch list and be alerted when they change.
\sa DataInterface::addObjectToWatchList()
*/
void objectModified(int objID);

//! Function called when the plugin is first created.
/*!
All of the plugin setup commands should go in here.
*/
bool init();

//! Function called just before the plugin is deleted.
/*!
All of the plugin cleanup work is done here.
*/
void cleanup();

//! Plugin can register to receive update calls every so often
/*!
Called by the base at regular time intervals.
\sa DataInterface::setUpdateTime()
```

```
*/
void update();

//! Plugins can watch for clicks in the 3D window.
/*!
Plugin must register with the rtBaseHandle first.
\sa rtBaseHandle::watchClick()
*/
void point3DSelected(double px, double py, double pz, int intensity);

public slots:

//! Slot that is called when the view angle is changed with the slider.
void viewAngleChanged(int);

//! Slot called when the dicom database button is clicked
void dicomButtonClicked();

protected:

//! Create a Simple Vurtigo test object.
void createTestObject();

//! Object ID for the small test object.
int m_smallVol;

////////////
// Gui Elements
////////////
//! Layout for the plugin GUI
QHBoxLayout *m_mainPluginGuiLayout;

//! Generic Qt Label to inform the user about the purpose of the slider
QLabel *m_viewAngleLabel;

//! Qt Labal to show the acurrent view angle value
QLabel *m_viewAngleAmtLabel;

//! Slider so the user can adjust the view angle.
QSlider *m_viewAngleSlider;

//! Button to launch the dicom widget
QPushButton *m_dcmButton;
private:

};

#endif
```

Both deriving from class DataInterface and the Q_INTERFACES(DataInterface) line are needed for Vurtigo to be able to load the plugin.

The V_PLUGIN_METADATA macro is used to export metadata for the plugin: it refers to an interface ID and a JSON file containing the metadata. The file property is optional, but if used, the JSON file may also refer to an empty set of metadata: {}

In the future, the interface ID may need to be a specific interface identifer for Vurtigo plugins, but at present can be any text identifier.

It is important to note that a number of functions need to be included in the plugin as they are pure virtual in

the interface:
objectModified(), init(), cleanup(), update(), point3DSelected().

The corresponding C++ source file for the above header is:

CameraMotionPlugin.cpp

```
#include "CameraMotionPlugin.h"
#include "rtApplication.h"
#include "rtMainWindow.h"
#include "rtCameraControl.h"
#include "rtBaseHandle.h"

#include "rt3dVolumeDataObject.h"

#include <vtkImageSinusoidSource.h>

void CameraMotionPlugin::objectModified(int) {
  // No objects are watched by this plugin.
}

bool CameraMotionPlugin::init() {

  // First create the GUI
  // Create the main widget (From DataInterface).
  // This parameter is NULL if the plugin does not use a widget.
  m_mainWidget = new QWidget();

  // Create the layout.
  m_mainPluginGuiLayout = new QHBoxLayout();

  // Create the widgets to go into the layout.
  m_viewAngleLabel = new QLabel();
  m_viewAngleAmtLabel = new QLabel();
  m_viewAngleSlider = new QSlider();
  m_dcmButton = new QPushButton("Dicom Database");

  // Add the widgets to the layout
  m_mainPluginGuiLayout->addWidget(m_viewAngleLabel);
  m_mainPluginGuiLayout->addWidget(m_viewAngleSlider);
  m_mainPluginGuiLayout->addWidget(m_viewAngleAmtLabel);
  m_mainPluginGuiLayout->addWidget(m_dcmButton);

  // Set the propeties for those widgets.
  m_viewAngleLabel->setText("View Angle: ");
  m_viewAngleAmtLabel->setText("30");
  m_viewAngleSlider->setMinimum(1);
  m_viewAngleSlider->setMaximum(179);
  m_viewAngleSlider->setValue(30);
  m_viewAngleSlider->setOrientation(Qt::Horizontal);

  // Make the horizontal at least 180 pixels.
  m_viewAngleSlider->setMinimumSize(180, 20);

  // Set the layout containing the custom widgets.
  m_mainWidget->setLayout(m_mainPluginGuiLayout);

  // Connect the signal to the slot.
  connect(m_viewAngleSlider, SIGNAL(valueChanged(int)), this, SLOT(viewAngleChanged(int)));

  connect(m_dcmButton, SIGNAL(clicked()), this, SLOT(dicomButtonClicked()));
  // Create a simple Vurtigo test object.
  createTestObject();

  // Ask the base to update this plugin every 500 msec
```

```
  setUpdateTime(500);

  // Return true to show that the plugin has been created as required.
  return true;
}

void CameraMotionPlugin::cleanup() {
  // Note that there is no cleanup of widgets here.
  // In Qt when a widget is added to a layout or a layout is added to a widget the parent takes ownership of the child
and is will delete it.
}

void CameraMotionPlugin::update(){
  // This function will be called as specified by setUpdateTime().

  // Check that the object was actually created.
  // If an object creation fails then the id is negative.
  if (m_smallVol >= 0) {

    // Get the object with the unique ID
    rt3DVolumeDataObject* ptObj =
static_cast<rt3DVolumeDataObject*>(rtBaseHandle::instance().getObjectWithID(m_smallVol));

    // Check that the object exists.
    if (ptObj) {

      // Lock the object before doing modifications
      ptObj->lock();

      // Apply a rotation to the data
      ptObj->getTransform()->RotateX(5);

      // Reset the modified time.
      ptObj->Modified();

      // Unlock the object after modifications are finished.
      ptObj->unlock();

    }
  }
}

void CameraMotionPlugin::point3DSelected(double, double, double, int) {
  // Plugin does not watch for point selection in the 3D view.
}

void CameraMotionPlugin::viewAngleChanged(int angle) {

  // Change the text in the GUI to inform the user that a change has taken place.
  m_viewAngleAmtLabel->setText(QString::number(angle));

  // Get a pointer to Vurtigo's camera control
  rtCameraControl *camera = NULL;
  QList<int> keys = rtBaseHandle::instance().getRenWinIDs();
  for (int ix1=0; ix1<keys.size(); ix1++)
  {
     if (rtApplication::instance().getMainWinHandle()->getCameraControl(keys[ix1]))
     {
        camera = rtApplication::instance().getMainWinHandle()->getCameraControl(keys[ix1]);
        break;
     }

  }

  // Give the camera control the new view angle.
  if (camera) { camera->setViewAngle(static_cast<double>(angle)); }
}
```

```
void CameraMotionPlugin::createTestObject() {
  m_smallVol = rtBaseHandle::instance().requestNewObject(rt3DVolumeDataObject::ObjectType(), "Small Test
Volume");

  // Check that the object was actually created.
  // If an object creation fails then the id is negative.
  if (m_smallVol >= 0) {

    // Get the object with the unique ID
    rt3DVolumeDataObject* ptObj =
static_cast<rt3DVolumeDataObject*>(rtBaseHandle::instance().getObjectWithID(m_smallVol));

    // Check that the object exists.
    if (ptObj) {

      // Create a source of input data.
      vtkImageSinusoidSource* sinSrc = vtkImageSinusoidSource::New();
      sinSrc->SetWholeExtent(1,32, 1, 32, 1, 32);
      sinSrc->SetDirection(1, 2, 3);
      sinSrc->SetPeriod(5);
      sinSrc->SetPhase(1);
      sinSrc->SetAmplitude(15);
      sinSrc->Update();

      // Lock the object before doing modifications
      ptObj->lock();


      // Copy the image data over.
      ptObj->copyNewImageData(sinSrc->GetOutput(),0);
      // Apply a translation to the data
      //ptObj->translateData(10, 20, 2);
      // Reset the modified time.
      ptObj->Modified();

      // Unlock the object after modifications are finished.
      ptObj->unlock();

      // Delete the source as it has already been copied.
      sinSrc->Delete();
    }
  }

}

//! An example of using the tool launcher
void CameraMotionPlugin::dicomButtonClicked()
{
    rtBaseHandle::instance().launchTool(rtBaseHandle::DICOM_TOOL);
}

// This last line will export the plugin.
V_EXPORT_PLUGIN(CameraMotion, CameraMotionPlugin)
```

The functions of interest are:

- init() where the plugin is created.
- update() where the volume is updated at regular intervals.
- viewAngleChanged() where the plugin interacts with the Vurtigo camera.
- createTestObject() where the plugin creates the small test volume.

The V_EXPORT_PLUGIN macro at the end of the file exports the class CameraMotionPlugin for the plugin

CameraMotion.

Building Plugins

Plugins can be built with CMake. This requires a "CMakeLists.txt" file for each plugin. This file is distributed with the plugin sources and will differ depending on whether the plugin is being built with the main Vurtigo source code or separately with the Vurtigo SDK.

Internal Developers

To build a plugin distributed with the internal Vurtigo source code, ensure the associated CMake variable is turned on during the configuration process. Most are turned on by default. For the CameraMotionPlugin, turn on VURTIGO_PLUGIN_CAMERA_MOTION then configure, generate, and build as usual.

SDK Developers

The Vurtigo SDK is distributed as a docker image: see the README file distributed with the sample plugins for instructions on how to build them and recommendations for working with your own plugins.

The CameraMotion plugin is provided in `/home/vdev/CameraMotion` in the SDK image and a HelloWorld plugin is available on github.

Generally, to build a plugin with the Vurtigo SDK: create a build directory, configure, generate, and build with CMake.

From a Linux terminal, for example:

```
mkdir build
cd build
ccmake ..
```

This will bring up a terminal-based user interface with keyboard controls, as documented in the menu at the bottom. Press the "c" key to configure; this will likely need to be done twice. Press the "g" key to generate the makefiles. This will exit back to the terminal once complete. From the terminal:

```
make
```

If using an IDE, building may be done through the GUI. We recommend using VS Code to build your own plugins with the docker image using Development Containers, as described in the sample plugin README file. With the sample devcontainer.json file (available from the HelloWorld repository), this will handle mounting your code directory, installing the necessary extensions into the container, and permissions for display.

Execution Order and Threads

The main Vurtigo thread is controlled by the message loop from the QApplication. This message loop receives input from three source types:

- The User Interface. For example, when the user clicks on a button, rotates the camera or changes the options for an object. Depending on which interface this is, the message can be sent to the rtMainWindow or to the GUI handler for an individual plugin or to the GUI handler for a

visualization object.
- An Event Timer. The rtTimeManager keeps a number of timers that specify how often certain windows or plugins need to be checked for updates. It is important to note that timers should not be made to fire faster than every 10ms or so.
- A Plugin. Plugins may have separate threads that send messages to the main thread.

The main thread is responsible for handling the events that it receives and sending possible render requests to VTK. Some event handling (for example VTK rendering) may spawn separate threads which run until the event has been processed. When one event has finished processing the main thread regains control and continues by sending the next event.

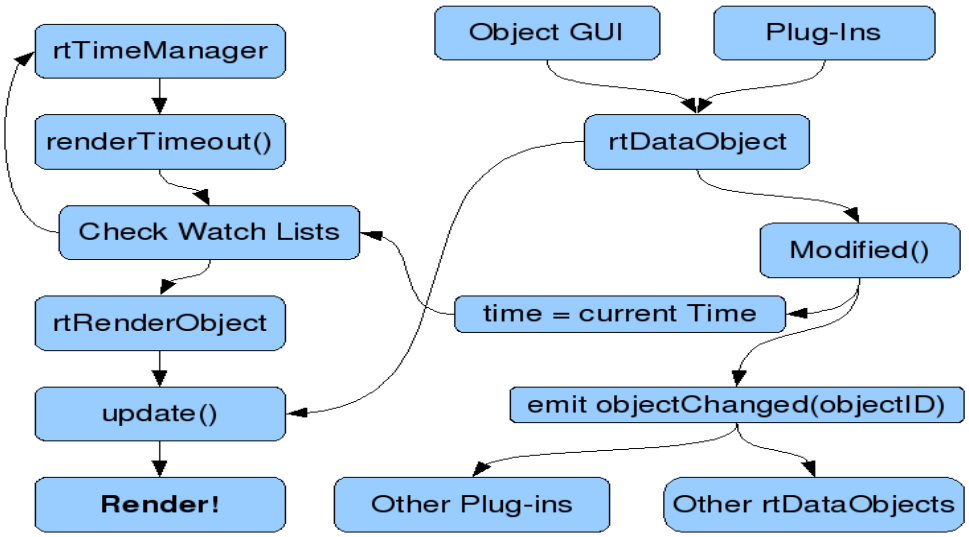
As a general rule the event thread will not regain control until the current event has finished processing. The weakness with this approach is that the GUI remains frozen while the event is being processed. If this processing take more than 20-30 ms the freeze is noticeable by the user. If the task is longer than 1500 ms then this may become a concern to the user. As a result, there is one exception to this rule: QCoreApplication::processEvents(). This function is called to temporarily return the control back to the event loop to allow it to process more events before continuing the current time-consuming task. This function should be used carefully as unfinished tasks may leave objects in inconsistent states and therefore other events that use those objects may cause errors.

Events and Event Handling

Events in Vurtigo serve three main purposes.

1) Communicating between the GUI and each object and/or plugin. This is the most common way for a developer to use the signal and slot functionality from Qt. GUI elements will emit signals and the object or plugin behind that GUI will have a slot that is called.
2) Passing messages between threads. Since rendering can only be completed in the main GUI thread all information must eventually be given to the main thread. The message passing system in Qt is ideal for this because it provides a messaging buffer between that can be filled by one thread and read by another.
3) Broadcasting notifications to all interested objects or plugins. When an object is modified many different other objects or plugins need to adapt to those changes. Similarly, when the user clicks on the 3D scene the interaction widgets, objects and cameras need to know about the click.

All objects in Vurtigo have the same **update pathway**. This is the main event pathway in Vurtigo and a way for changing objects to update the 3D/2D views.



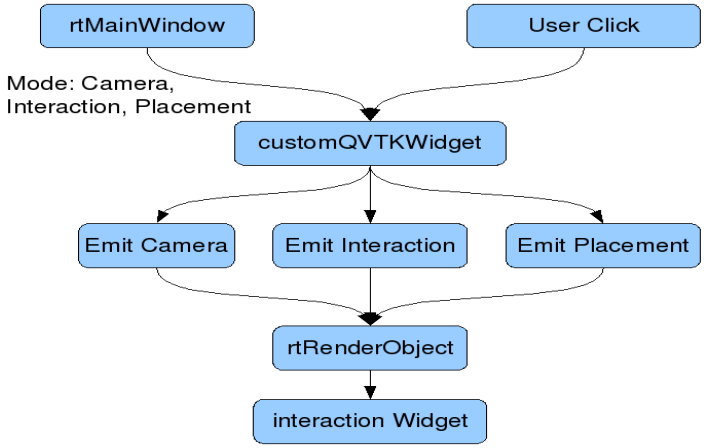
The update pathway is composed of two pieces.

On the left is the render update that starts with the time manager which checks the watch lists at regular intervals. A **watch list** is just a list of rendered objects that are currently visible. Objects that are not being rendered do not need to have their pipelines updated and re-rendered and so are not part of the list. For each object in the list the last modified time is compared against the last render time. If the object has been modified since the last render it will be updated and re-rendered. During the update phase the render object finds the corresponding data object and updates the rendering pipeline accordingly. Once all of the render pipelines have been updated the main window is re-rendered.

On the right, the data object is updated either through the GUI or through one of the plugins. For every change in data the Modified() function is called. For every Modified() call the modified time is updated to the current time and the object emits a objectChanged(objectID) signal. Plugins or other objects can connect to this signal to be notified when the object changes.

Though it may seem that much of the code can be eliminated by connecting update() on the render side with the objectChanged() signal on the data object side, there are two reasons why this was not done. First, it is undesirable to update the render every time an object is changed. Objects can change at a rate higher than the possible frame rate which will create a backlog of calls. Multiple objects may be changed simultaneously which will result in a render call for each object when a single call at the end would be enough. Secondly, immediate render calls would slow down plugins and force them to execute in sequence rather than in parallel as only one plugin/thread can access the renderer at one time.

Another important event pathway is the **3D interaction pathway**. It represents the way that mouse clicks pass from the 3D window to the objects that are being interacted with.



The user's click on the 2D or 3D view is transformed by Qt into a QEvent type of object and passed to the customQVTKWidget of the window that generated the Qt event. Each widget can then handle the interaction event itself depending on the interactions currently enabled in the widget. It can also emit an event signal, depending on the type of interaction, that also includes a reference back to itself. Render objects can choose to listen to one or more types of signals from the customQVTKWidget. These signals are sometimes passed on to an interaction widget that can further interact with the user clicks. A good example of an interaction widget is “Plane2DControlWidget”. Once an interaction event has been handled by an object, the event should be accepted using the “accept()” method to indicate that further processing of the event is not required.

Tips and Traps

Object Selection Combo Box

In many cases it is necessary for an object or plugin to ask the user to select an existing object of a particular type to use. The “ObjectSelectionComboBox” is derived from QComboBox and serves this purpose. The programmer sets the object types that can be selected and the box will then list objects of those types. This combo box keeps track of objects as they are created and destroyed by the application, changing the list dynamically. So, if you want to ask the user to “Choose a Catheter to Track”, or “Choose a Point Set to Register”, then use this widget. This needs to be done programmatically by instantiating this widget and adding it to the layout of the plugin’s interface. For example, the following adds this widget to a plugin’s existing layout, allowing selection of 2D slice objects:

```
#include "ObjectSelectionComboBox.h"
...
m_combo = new ObjectSelectionComboBox();
m_combo->addObjectType("OT_2DObject");
my_HBoxLayout->addWidget(m_combo);
```

The vtkRenderWindowInteractor

Vurtigo does not make use of the vtk style of interactor, using Qt mouse events instead. However, the vtk interactor cannot be completely removed from Vurtigo. Both the QVTK render widget and the orientation widget (the little direction marker in the bottom corner) need a valid pointer to a vtk interactor. Inside the rtMainWindow class the valid pointer to the vtk interactor is read from the QVTK widget and given to the

orientation widget. However, it is disabled (and will remain disabled!) to avoid conflicts between the Qt mouse/keyboard events and the vtk interactor.