H62PEP Computing - Worksheet 6 - Compiling, Installing and Using the VTK Library

P. Evans

The VTK Library

The VTK library is open-source software system for 3D computer graphics, image processing, and visualization. It consists of a C++ class library and allows visualisation of complex 3D CAD models and simulation results. You will use VTK to enable model visualisation

Building VTK from Source

VTK is built using CMake and so the basic process is very similar to the process you have used to build your own library, but if you need VTK to work with Qt (which you do) there are a few extra configuration steps that are required. The key steps you need to follow are outlined below so you have an overview of the process, and a full and detailed explanation of the build process on Windows and Unix-like platforms is given here (https://www.vtk.org/Wiki/VTK/Configure_and_Build#Configure_with_Qt) - use this as a reference when more detail is needed.

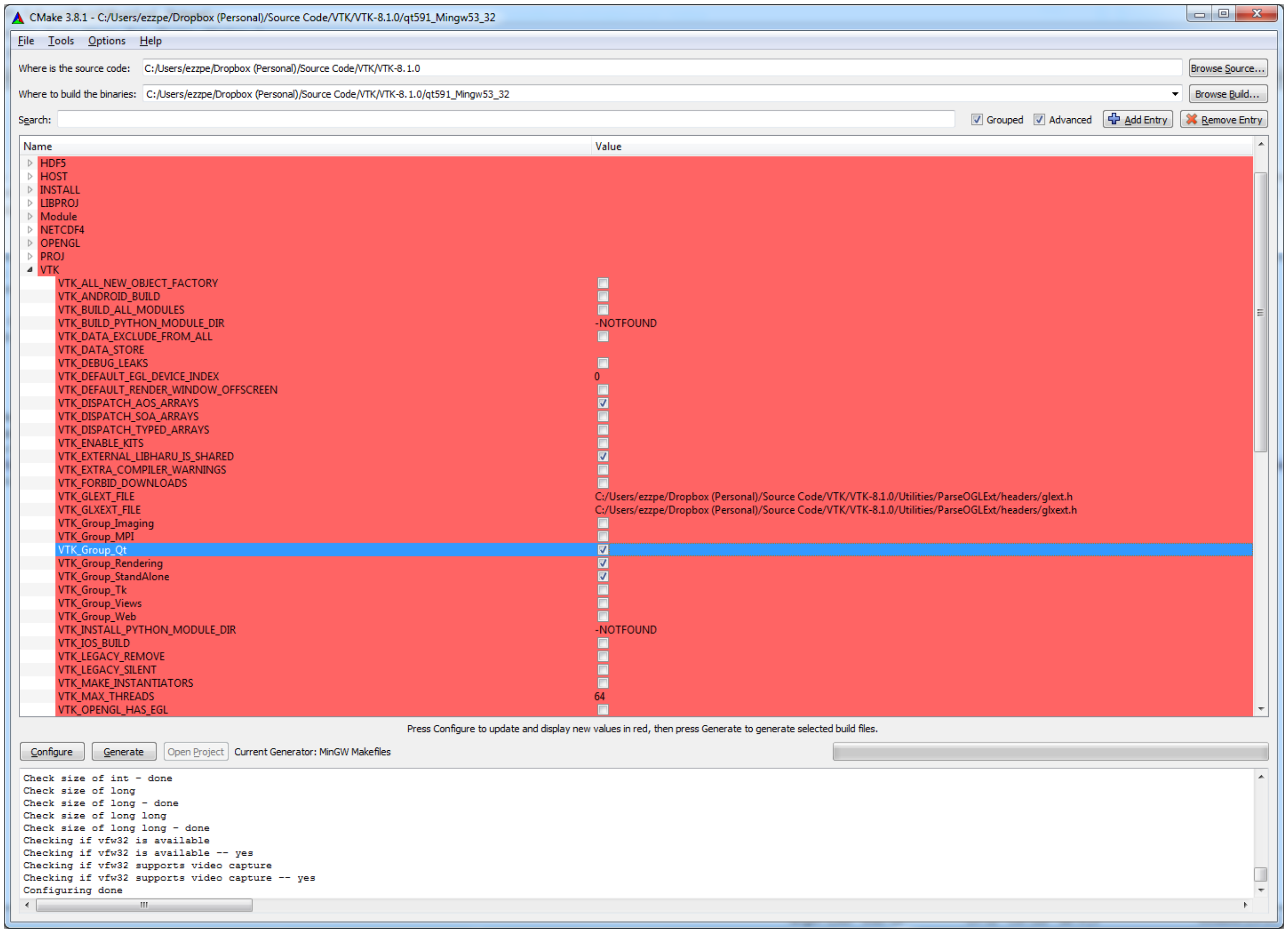
Preparation

- Visit the VTK website and download the latest source code package. Extract the source code to a suitable location (not a network drive).
- Obtain a compiler and install if necessary. Note if you are using MinGW, it should be downloaded as part of the Qt package (next step) to ensure compatibility.
- Obtain Qt and install it on your system if it isn't already present.

On Windows you should use Visual Studio, you can use MinGW but it is more likely to give you problems during compile and requires a bit more configuration effort to get it to work.

Configuring and building VTK (Windows)

- It is easiest to configure CMake using the CMake GUI, so open this and set the path to the source code and a suitable build directory, e.g.:
  - Source code: C:/Users/ezzpe/Source Code/VTK/VTK-8.1.0
  - Build directory: C:/Users/ezzpe/Source Code/VTK/VTK-8.1.0/qt591_Mingw53_32
- Make sure that the relevant *bin* directory for MinGW (if you are using this compiler) is in the system path as this maximises the chance that CMake will find what it needs automatically.
- Click *Configure* and CMake will make an initial guess at a build configuration, this will probably take a while.
- When this completes, you'll need to tell CMake that you want to compile VTK with QT support. Do this by ensuring that the VTK/ *VTK_Group_Qt* option is selected in CMake's variable cache.
- Now you'll need to tell CMake where the correction version of Qt is on your system. Add a *CMAKE_PREFIX_PATH* variable entry and set it to the path to the base directory of your Qt install (specific qt base directory for your compiler).
- Repeat the above process to specify the location of Qmake. Add a *QT_QMAKE_EXECUTABLE* variable entry and set it to the path to qmake.exe in the Qt install (its in the bin directory for the specific version of Qt and compiler you are using).


CMake 3.8.1 - C:/Users/ezzpe/Dropbox (Personal)/Source Code/VTK/VTK-8.1.0/qt591_Mingw53_32

Where is the source code: C:/Users/ezzpe/Dropbox (Personal)/Source Code/VTK/VTK-8.1.0

Where to build the binaries: C:/Users/ezzpe/Dropbox (Personal)/Source Code/VTK/VTK-8.1.0/qt591_Mingw53_32

Search:

✓ Grouped ✓ Advanced

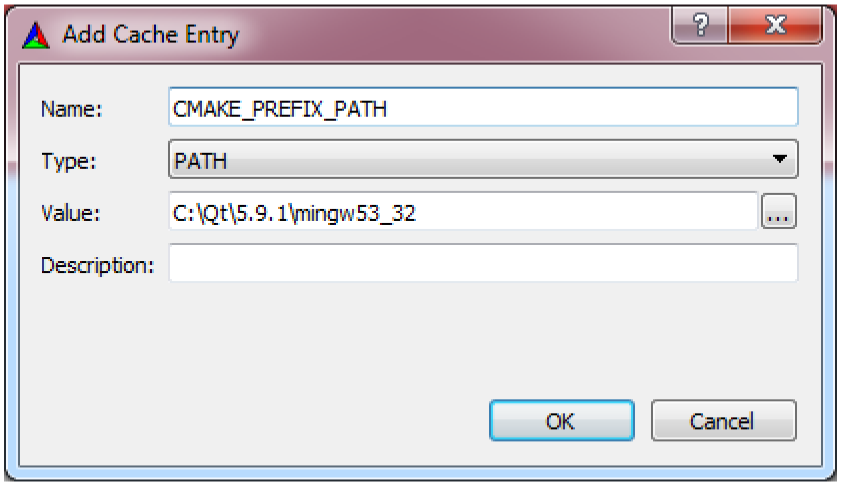
Name	Value
HDF5	
HOST	
INSTALL	
LIBPROJ	
Module	
NETCDF4	
OPENGL	
PROJ	
VTK	
VTK_ALL_NEW_OBJECT_FACTORY	☐
VTK_ANDROID_BUILD	☐
VTK_BUILD_ALL_MODULES	☐
VTK_BUILD_PYTHON_MODULE_DIR	-NOTFOUND
VTK_DATA_EXCLUDE_FROM_ALL	☐
VTK_DATA_STORE	
VTK_DEBUG_LEAKS	☐
VTK_DEFAULT_EGL_DEVICE_INDEX	0
VTK_DEFAULT_RENDER_WINDOW_OFFSCREEN	☐
VTK_DISPATCH_AOS_ARRAYS	✓
VTK_DISPATCH_SOA_ARRAYS	☐
VTK_DISPATCH_TYPED_ARRAYS	☐
VTK_ENABLE_KITS	☐
VTK_EXTERNAL_LIBHARU_IS_SHARED	✓
VTK_EXTRA_COMPILER_WARNINGS	☐
VTK_FORBID_DOWNLOADS	☐
VTK_GLEXT_FILE	C:/Users/ezzpe/Dropbox (Personal)/Source Code/VTK/VTK-8.1.0/Utilities/ParseOGLExt/headers/glext.h
VTK_GLXEXT_FILE	C:/Users/ezzpe/Dropbox (Personal)/Source Code/VTK/VTK-8.1.0/Utilities/ParseOGLExt/headers/glxext.h
VTK_Group_Imaging	☐
VTK_Group_MPI	☐
VTK_Group_Qt	✓
VTK_Group_Rendering	✓
VTK_Group_StandAlone	✓
VTK_Group_Tk	☐
VTK_Group_Views	☐
VTK_Group_Web	☐
VTK_INSTALL_PYTHON_MODULE_DIR	-NOTFOUND
VTK_IOS_BUILD	☐
VTK_LEGACY_REMOVE	☐
VTK_LEGACY_SILENT	☐
VTK_MAKE_INSTANTIATORS	☐
VTK_MAX_THREADS	64
VTK_OPENGL_HAS_EGL	☐

Press Configure to update and display new values in red, then press Generate to generate selected build files.

Current Generator: MinGW Makefiles

```
Check size of int - done
Check size of long
Check size of long - done
Check size of long long
Check size of long long - done
Checking if vfw32 is available
Checking if vfw32 is available -- yes
Checking if vfw32 supports video capture
Checking if vfw32 supports video capture -- yes
Configuring done
```

Figure 1: Qt CMake Cache Group


Add Cache Entry

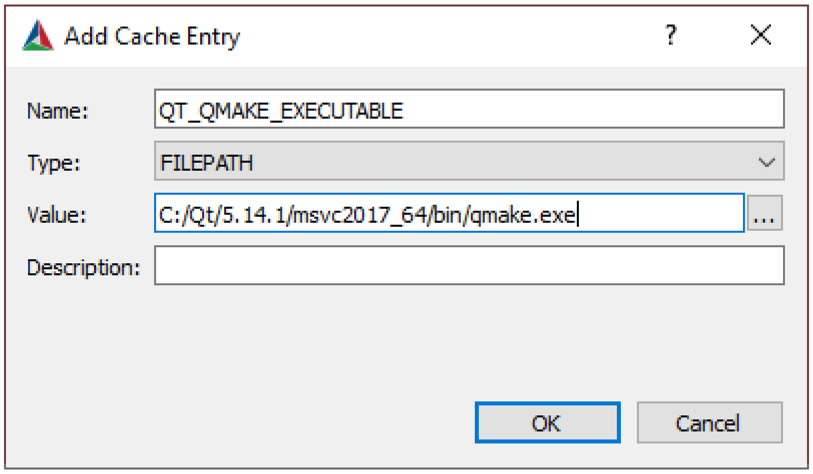
Name: CMAKE_PREFIX_PATH

Type: PATH

Value: C:\Qt\5.9.1\mingw53_32

Description:

Figure 2: CMake Path Variable


Add Cache Entry

Name: QT_QMAKE_EXECUTABLE

Type: FILEPATH

Value: C:/Qt/5.14.1/msvc2017_64/bin/qmake.exe

Description:

Figure 3: QMake Executable Variable

- Now ask CMake to rerun the *configure* process, you should get some additional Qt related settings appear in the Ungrouped Entries section of CMake's cached variables.
- Depending on the compiler(s) that you intend to use, you may need to make some further changes to the CMake Cache variables. See notes in the two sections below.
- Ask CMake to *generate* the Makefiles / Codeblocks project / MS Visual Studio project for you.
- Build VTK using your chosen method. This can take an hour or so to complete.

Changing the default install location (Windows) It may be useful to change the location into which VTK is installed, particularly if you might want to compile it twice. (e.g. create both a Visual Studio and MinGW, or for two different versions of Visual Studio). You can change a variable in the CMake Cache to do this: `CMAKE/CMAKE_INSTALL_PREFIX`. This is the location that VTK will install to after it is built, you might want to choose something like `C:\Program Files\VTK\8.2.0_MSVC2019_Qt5.9` if you were using Visual Studio 2019, VTK8.2 and Qt5.9. This helps you identify specific versions of the library later.

Configuring and building VTK (Other OS)

The process is basically the same as above. See https://www.vtk.org/Wiki/VTK/Configure_and_Build#Configure_with_Qt for more details. The guide in the link above may tell you to build Qt from source, don't do this if you can help it! Try to configure VTK to use the version of Qt that you already have installed. If you are using a Mac and struggling, there are prebuilt versions of VTK available through the Homebrew package manager.

Installing the VTK Libraries

For MinGW type `mingw32-make install`, in Visual Studio build the *INSTALL* target, and on other OSs type `make install`. Be aware that if you haven't changed the install location on Windows, the default is in the *Program Files* directory and install will fail if the build program (i.e. Visual Studio) isn't runnign with administrator privileges.

Using VTK

Overview of VTK

VTK is a set of libraries that allow you to generate advanced computer graphics and visualisations using an object orientated approach. It is a high level library - that is you specify what you want to achieve: the objects you want to draw, their location, the lighting conditions, the camera or viewpoint location, and it handles the rest. It gives you easy to use interfaces to more advanced capabilities such as loading objects from professional 3D model files, make measurements on models and overlay results that might be produced by computer simulations. This is in contrast to a low-level graphics library such as OpenGL or Direct3D where you are responsible for describing exactly how a 3D model should be rendered, down to individual vertex and polygon level.

Fluid flow simulation results visualised using VTK

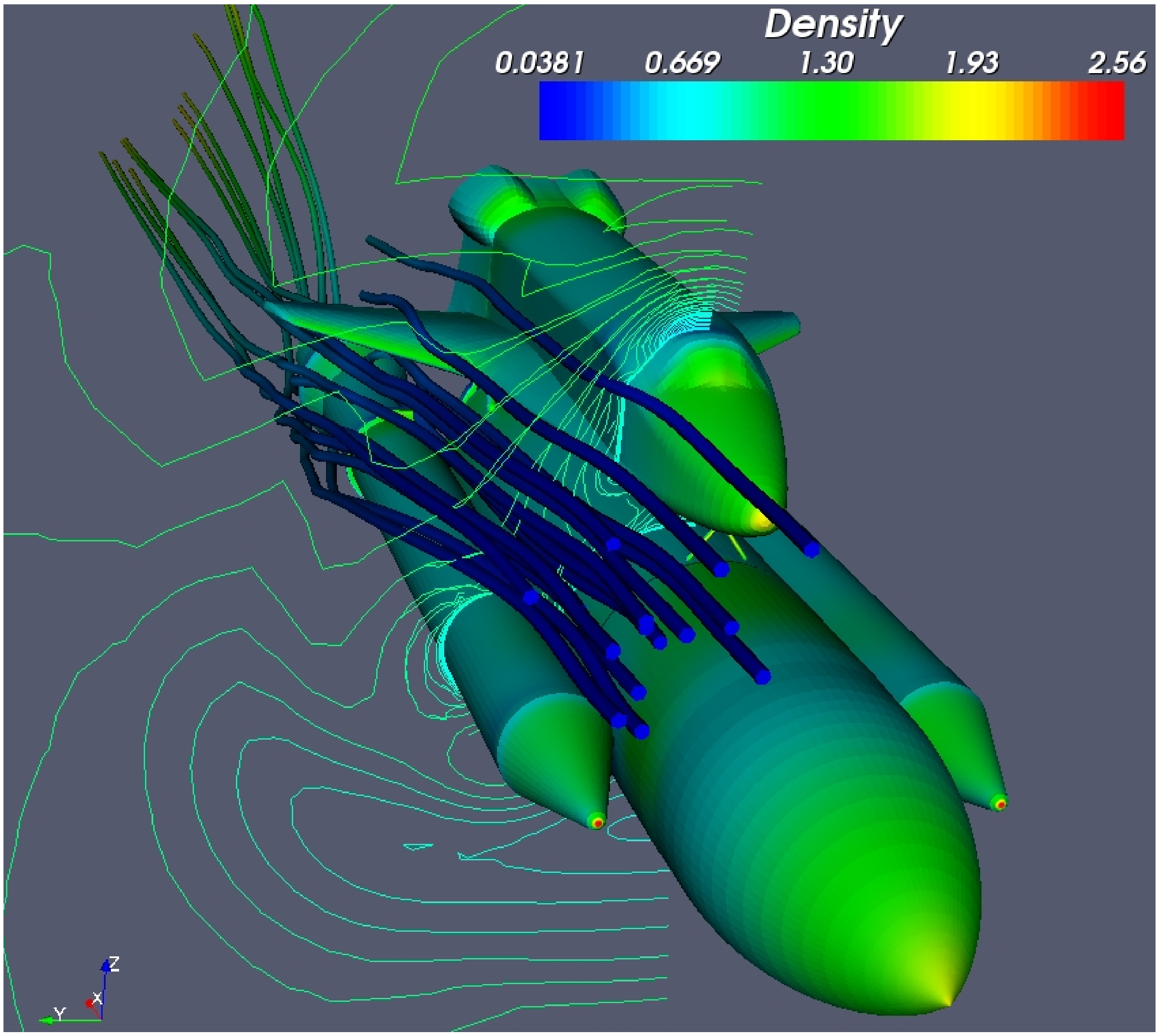


Figure 4: VTK Visualisation of Fluid Floow ARound Space Shuttle

You create a 3D visualisation in VTK by creating and linking a number of fundamental objects (C++ Classes). These objects are[1]:

- ***vtkMapper*** — the geometric representation for a 3D object. It must be created from some source data which could be a `vtkSource` - a class that provides basic geometry data for basic shapes such as cubes, or from a CAD model file.
- ***vtkActor*** — represents how an object, represented by a`vtkMapper`, is rendered in the scene. It holds the objects properties (using a `vtkProperty` class) and also its position in the world coordinate system. (Note: `vtkActor` is a subclass of `vtkProp` and `vtkProp` is a more general form of actor that includes annotation and 2D drawing classes.)
- ***vtkProperty*** — a class that defines the appearance properties of an actor including color, transparency,and lighting properties such as specular and diffuse. Also representational properties like wireframe and solid surface.
- ***vtkLight*** — a source of light to illuminate the scene.
- ***vtkCamera*** — defines the view position, focal point, and other viewing properties of the scene.
- ***vtkRenderer*** — coordinates the rendering process and acts as a container for lights, cameras, and actors.
- ***vtkRenderWindow*** — manages a window on the display device; one or more renderers draw into an instance of *vtkRenderWindow*. The render window can be embedded within a Qt Widget.

A basic VTK program creates objects of these types and links them together to form a pipeline that describes how one or more models, each with geometry (from a *source* that could be a file or a list of coordinates in your code), are loaded into a `vtkMapper`, which are then placed in a world (using `vtkActor` objects), which are then illuminated by `vtkLight` classes, viewed through a `vtkCamera`, and rendered onto your screen in a `vtkRenderWindow` (or a `QVTKOpenGLWidget` in our case). Filters can be inserted into this pipeline to add visual effects to the visualisation.

For more information on how VTK works, see the VTK Textbook (https://www.kitware.com/products/books/VTKTextbook.pdf) Chapter 3 up to 3.10 gives a good introduction to computer graphics. Chapter 3, 3.10 onwards explains the basics of how VTK works.

Example 1 - Simple VTK Application

- Create the following 2 files, vtk_cube.cpp and CMakeLists.txt
- Now open and command prompt, create a build directory, and use CMake to generate the build files for the project. If you are using Windows, remember that now CMake must be able to find the compiler and also the VTK libraries that you installed. Similar to when using Qt, there are two things that you need to do:
  - Make sure that the *bin* directories for VTK, Qt (as we'll be using this also later, and compiler if using MingW, are in the path. This is important to allow the compiled program to run as it must be able to find the DLLs in these folders.
  - Add the base install directory of VTK (and Qt) to the CMAKE_PREFIX_PATH variable by passing it to CMake using the -D command line option, or editing the variable in CMakeGUI.
- If using the MinGW compiler, you'll also need to tell it to expect a newer version of C++ as this is what VTK uses (by defining the cxx_flag *-std=c++11*).

E.g. for MinGW:

```
> set PATH=C:\Program Files\CMake\bin;C:\Qt\5.9.1\mingw53_32\bin;C:\VTK\M
inGW53_32\bin;C:\Qt\Tools\mingw530_32\bin

> cmake -DCMAKE_PREFIX_PATH="C:\Qt\Tools\mingw530_32;C:\VTK\MinGW53_32;C:
\Qt\5.9.1\mingw53_32" -G"MinGW Makefiles" -DCMAKE_CXX_FLAGS="-std=c++11" ..

> mingw32-make
```

or for Visual Studio:

```
> "C:\Program Files (x86)\Microsoft Visual Studio\2017\Community\VC\
                                        Auxiliary\Build\vcvarsall.bat" amd64
```

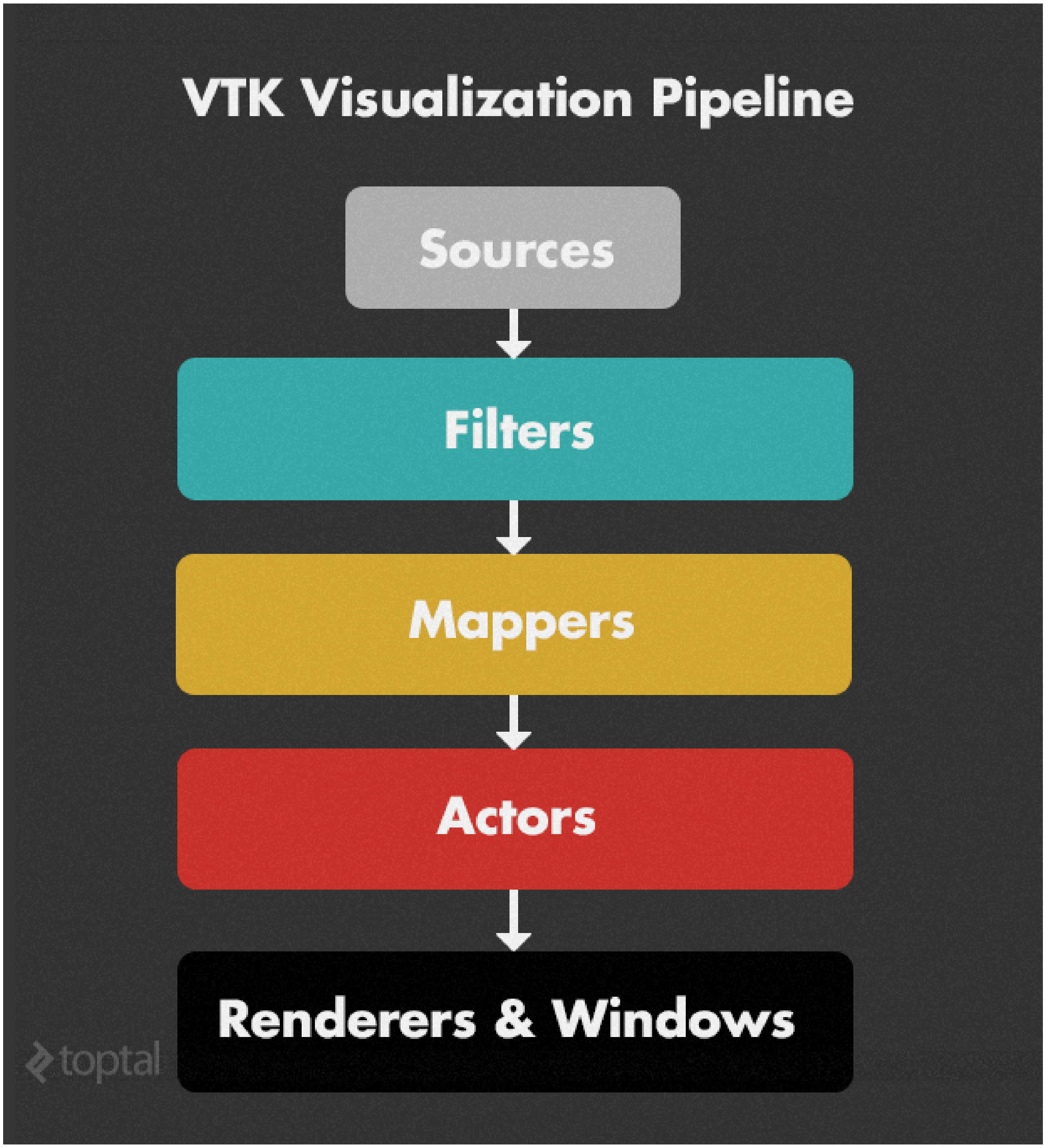


Figure 5: The VTK Pipeline

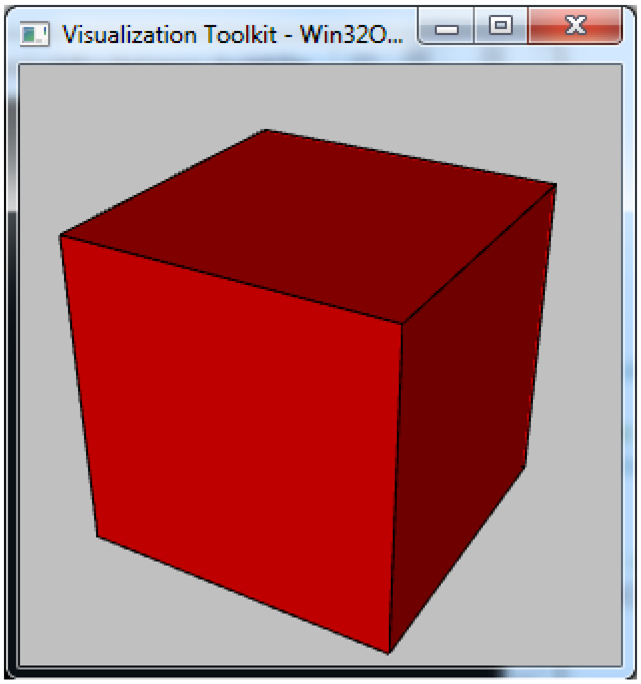


Figure 6: Simple VTK Application

```
> set PATH=%PATH%;C:\Qt\5.9.1\msvc2017_64\bin;C:\VTK\MSVC2017_64\bin

> cmake -DCMAKE_PREFIX_PATH="C:\VTK\MSVC2017_64;C:\Qt\5.9.1\msvc2017_64"
        -G"Visual Studio 15 2017 Win64" ..

> msbuild Cube.sln
```

```cpp
// vtk_cube.cpp-----------------------------------------------------------------
#include <vtkSmartPointer.h>
#include <vtkCubeSource.h>
#include <vtkActor.h>
#include <vtkProperty.h>
#include <vtkCamera.h>
#include <vtkPolyData.h>
#include <vtkDataSetMapper.h>
#include <vtkRenderWindow.h>
#include <vtkRenderWindowInteractor.h>
#include <vtkRenderer.h>
#include <vtkNamedColors.h>

int main(int, char *[])
{
  // Create a cube using a vtkCubeSource
  vtkSmartPointer<vtkCubeSource> cubeSource =
                  vtkSmartPointer<vtkCubeSource>::New();

  // Create a mapper that will hold the cube's geometry in a format suitable for
    // rendering
  vtkSmartPointer<vtkDataSetMapper> mapper =
                  vtkSmartPointer<vtkDataSetMapper>::New();
  mapper->SetInputConnection(cubeSource->GetOutputPort());

  // Create an actor that is used to set the cube's properties for rendering
    // and place it in the window
  vtkSmartPointer<vtkActor> actor =
              vtkSmartPointer<vtkActor>::New();

  actor->SetMapper(mapper);
  actor->GetProperty()->EdgeVisibilityOn();

  vtkSmartPointer<vtkNamedColors> colors =
              vtkSmartPointer<vtkNamedColors>::New();
  actor->GetProperty()->SetColor( colors->GetColor3d("Red").GetData() );

  // Create a renderer, and render window
  vtkSmartPointer<vtkRenderer> renderer =
              vtkSmartPointer<vtkRenderer>::New();

  vtkSmartPointer<vtkRenderWindow> renderWindow
              vtkSmartPointer<vtkRenderWindow>::New();

  renderWindow->AddRenderer(renderer);

  // Link a renderWindowInteractor to the renderer (this allows you to capture
    // mouse movements etc)
  vtkSmartPointer<vtkRenderWindowInteractor> renderWindowInteractor =
              vtkSmartPointer<vtkRenderWindowInteractor>::New();

  renderWindowInteractor->SetRenderWindow(renderWindow);

  // Add the actor to the scene
  renderer->AddActor(actor);
  renderer->SetBackground( colors->GetColor3d("Silver").GetData() );

  // Setup the renderers's camera
  renderer->ResetCamera();
  renderer->GetActiveCamera()->Azimuth(30);
  renderer->GetActiveCamera()->Elevation(30);
  renderer->ResetCameraClippingRange();

  // Render and interact
  renderWindow->Render();
  renderWindowInteractor->Start();

  return EXIT_SUCCESS;
}
// /vtk_cube.cpp-----------------------------------------------------------------
```

```
# CMakeLists.txt----------------------------------------------------------------
cmake_minimum_required( VERSION 2.8 )

PROJECT( Cube )

find_package( VTK REQUIRED )
include( ${VTK_USE_FILE} )

add_executable( Cube vtk_cube.cpp )
```

```
target_link_libraries( Cube ${VTK_LIBRARIES} )
# /CMakeLists.txt---------------------------------------------------------------
```

NOTE 1: The build commands above also tell CMake where the correct version of Qt is installed. You don't actually need Qt for this example but you will need it for subsequent examples, so its easiest just to have these present whenever you use VTK with CMake.

NOTE 2 - vtkSmartPointers The VTK Smart Pointer object is a class that takes the place of a regular C/C++ pointer. The reason for using smart pointers is that it simplifies memory management. If you allocate a an object using a traditional C++ pointer: `vtkActor *actor = new vtkActor();` then you also need to make sure you free the memory pointed to by actor when you've finished using it: `delete actor;` If you don't delete, the memory will remain allocated until the program exits and if you make this mistake multiple times your program could ultimately run out of memory. This is called a memory leak. `vtkSmartPointer`s automatically detect when the memory that you've allocated is no longer needed and delete/free the memory for you. You can find a more detailed explanation of this topic here: https://www.vtk.org/Wiki/VTK/Tutorials/SmartPointers Note that the `vtkSmartPointer` class also uses C++'s template functionallity.

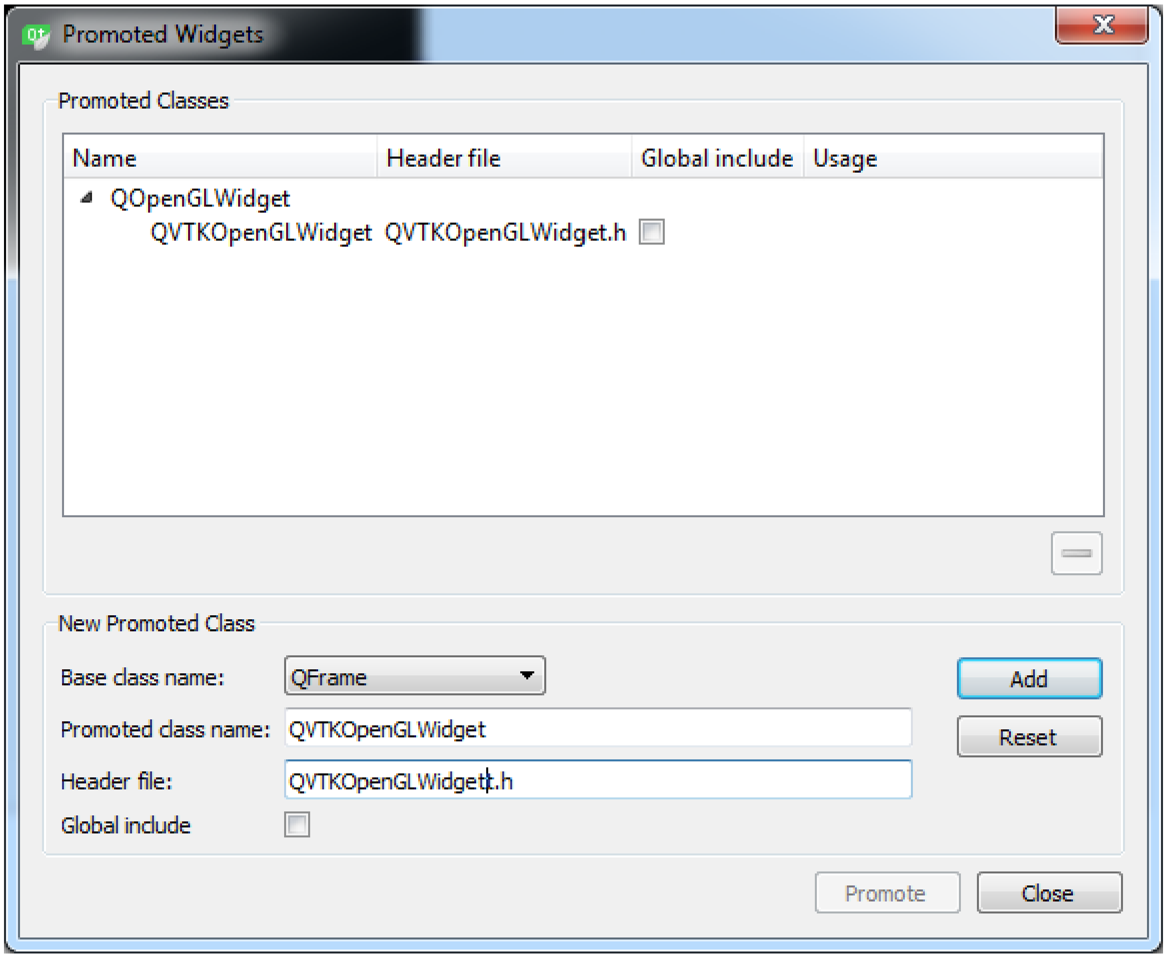
Exercise 2 - Embedding the VTK rendering in a Qt application Exercise 1 is an extremely basic example of the use of VTK for rendering a 3D object which is ultimately what your software application needs to do, exercises 3 and onwards will get you to explore the capabilities of VTK in a bit more detail but first you need to be able to link the capabilities of VTK for 3D model visualisation, with those of Qt for GUI development.

You will need to create a new Qt MainWindow widget that will form the basis of the application. This will consist of a .ui file dscribing the MainWindow, and .cpp/.h source files describing its behaviour. The process is described below, but you can download necessary files from this GitHub repository if you are having difficulties or running short on time: https://github.com/plevans/qtvtk

Or, if you want to create the files yourself, or know how to create them:

- Start by creating opening *Qt Creator* and create a *Designer Form Class" of type* Main Window*. It is important to use Qt Creator (rather then Designer) as Creator will create a ui file and a corresponding class template. Designer will only create the ui file.
- Now you need to add user interface components to the Qt GUI in the same way as you did in previous weeks. One of these components must be a `QtVTKOpenGLWidget`, this is effectively a VTK window embedded within a Qt widget. This widget type doesn't exist in Qt designer so you'll need to insert a suitable base class placeholder and promote the widget to the correct type.
  - First insert a `QOpenGLWidget` as the base class.
  - Right click on this widget and choose "Promote to".
  - Promote the widget to a `QVTKOpenGLWidget` that uses the header file *QVTKOpenGLWidget.h*. (If you look into the VTK source code you'll find that `QOpenGLWidget` is actually the base class for `QtTKOpenGLWidget`)
- You can also add a few more widgets for user input (buttons etc) if you want so that you can use these to experiment with VTK features later. Don't worry too much about the GUI design for now as you can modify it later.
- Now you'll need to:
  - Modify the mainwindow.cpp file and edit it to add some code for the to drive the VTK widget.
  - Create a main.cpp file that contains a program to use your MainWindow class. (below)
  - Create a CMakeLists.txt file.
  - Rerun the CMake and compiler build commands

Remember, these files, including the .ui file can also be found at the following GitHub repository https://github.com/plevans/qtvtk.



Name	Header file	Global include	Usage
QOpenGLWidget			
QVTKOpenGLWidget	QVTKOpenGLWidget.h	☐	

Base class name:	QFrame
Promoted class name:	QVTKOpenGLWidget
Header file:	QVTKOpenGLWidget.h
Global include	☐

Figure 7: Promoting a Widget to a Widget of a Derived Class

```
// main.cpp------------------------------------------------------------------
#include <QApplication>
#include <QSurfaceFormat>
#include <QVTKOpenGLWidget.h>

#include "mainwindow.h"

int main( int argc, char** argv )
{
// needed to ensure appropriate OpenGL context is created for VTK rendering.
    QSurfaceFormat::setDefaultFormat( QVTKOpenGLWidget::defaultFormat() );
    QApplication a( argc, argv );
    MainWindow example;
    example.show();
    return a.exec();
}
// /main.cpp-----------------------------------------------------------------
// mainwindow.cpp-----------------------------------------------------------

#include <vtkSmartPointer.h>
#include <vtkCubeSource.h>
#include <vtkActor.h>
#include <vtkProperty.h>
#include <vtkCamera.h>
#include <vtkPolyData.h>
#include <vtkDataSetMapper.h>
#include <vtkRenderWindow.h>
#include <vtkRenderWindowInteractor.h>
#include <vtkRenderer.h>
#include <vtkNamedColors.h>
#include <vtkNew.h>
#include <vtkGenericOpenGLRenderWindow.h>

#include "mainwindow.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent), ui(new Ui::MainWindow)
{
    // standard call to setup Qt UI (same as previously)
    ui->setupUi( this );

    // Now need to create a VTK render window and link it to the QtVTK widget
    vtkNew<vtkGenericOpenGLRenderWindow> renderWindow;
    // note that vtkWidget is the name I gave to my QtVTKOpenGLWidget in Qt
    // creator
    ui->qvtkWidget->SetRenderWindow( renderWindow );

    // Now use the VTK libraries to render something. To start with you can
    // copy-paste the cube example code, there are comments to show where the
    // code must be modified.

    // Create a cube using a vtkCubeSource
    vtkSmartPointer<vtkCubeSource> cubeSource =
                    vtkSmartPointer<vtkCubeSource>::New();

    // Create a mapper that will hold the cube's geometry in a format
    // suitable for rendering
    vtkSmartPointer<vtkDataSetMapper> mapper =
                    vtkSmartPointer<vtkDataSetMapper>::New();

    mapper->SetInputConnection( cubeSource->GetOutputPort() );

    // Create an actor that is used to set the cube's properties for rendering
    // and place it in the window
    vtkSmartPointer<vtkActor> actor =
                    vtkSmartPointer<vtkActor>::New();

    actor->SetMapper(mapper);
    actor->GetProperty()->EdgeVisibilityOn();

    vtkSmartPointer<vtkNamedColors> colors =
                        vtkSmartPointer<vtkNamedColors>::New();

    actor->GetProperty()->SetColor( colors->GetColor3d("Red").GetData() );

    // Create a renderer, and render window
    vtkSmartPointer<vtkRenderer> renderer =
                        vtkSmartPointer<vtkRenderer>::New();

    // ###### We've already created the renderWindow this time as a qt
    // widget ######
    //vtkSmartPointer<vtkRenderWindow> renderWindow =
    //                  vtkSmartPointer<vtkRenderWindow>::New();

    ui->qvtkWidget->GetRenderWindow()->AddRenderer( renderer );                             // ###

    // Link a renderWindowInteractor to the renderer (this allows you to
    // capture mouse movements etc)  ###### Not needed with Qt ######
    //vtkSmartPointer<vtkRenderWindowInteractor> renderWindowInteractor =
    //                  vtkSmartPointer<vtkRenderWindowInteractor>::New();
    //
    //renderWindowInteractor->SetRenderWindow( ui->vtkWidget->GetRenderWindow() );

    // Add the actor to the scene
    renderer->AddActor(actor);
    renderer->SetBackground( colors->GetColor3d("Silver").GetData() );

    // Setup the renderers's camera
    renderer->ResetCamera();
    renderer->GetActiveCamera()->Azimuth(30);
    renderer->GetActiveCamera()->Elevation(30);
    renderer->ResetCameraClippingRange();

    // Render and interact
    //renderWindow->Render();                   // ###### Not needed with Qt ######
    //renderWindowInteractor->Start();// ###### Not needed with Qt ######
}

MainWindow::~MainWindow()
{
    delete ui;
}
```

```
    // /mainwindow.cpp------------------------------------------------------------

// mainwindow.h------------------------------------------------------------------
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>

namespace Ui {
class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget * parent = 0);
    ~MainWindow();

private:
    Ui::MainWindow * ui;
};

#endif // MAINWINDOW_H
//mainwindow.h-------------------------------------------------------------------

# CMakeLists.txt----------------------------------------------------------------
cmake_minimum_required(VERSION 3.3 FATAL_ERROR)

foreach(p
    CMP0071 # 3.10: Let AUTOMOC and AUTOUIC process GENERATED files
    )
  if(POLICY ${p})
    cmake_policy(SET ${p} NEW)
  endif()
endforeach()

PROJECT( QtVTKExample )

# The CMake build process might generate some new files in the current
# directory. This makes sure they can be found.
set( CMAKE_INCLUDE_CURRENT_DIR ON )


# This allows CMake to run one of Qt's build tools called moc
# if it is needed. moc.exe can be found in Qt's bin directory.
# We'll look at what moc does later.
set( CMAKE_AUTOMOC ON )
set( CMAKE_AUTOUIC ON )

# Find the Qt widgets package. This locates the relevant include and
# lib directories, and the necessary static libraries for linking.
find_package( Qt5Widgets )

set( UIS mainwindow.ui )
qt5_wrap_ui( UI_Srcs ${UIS} )
```

```
# Also link to VTK
find_package( VTK REQUIRED )
include( ${VTK_USE_FILE} )

# Define the executable output and its sources
add_executable( QtVTKExample MACOSX_BUNDLE
                main.cpp mainwindow.cpp
                mainwindow.h
                mainwindow.ui
                ${UI_Srcs}
                ${QRC_Srcs}
                )

# Tell CMake that the executable depends on the Qt::Widget libraries.
target_link_libraries( QtVTKExample Qt5::Widgets )

# Tell CMake that the executable depends on the VTK libraries
target_link_libraries( QtVTKExample ${VTK_LIBRARIES} )
# /CMakeLists.txt--------------------------------------------------------------
```

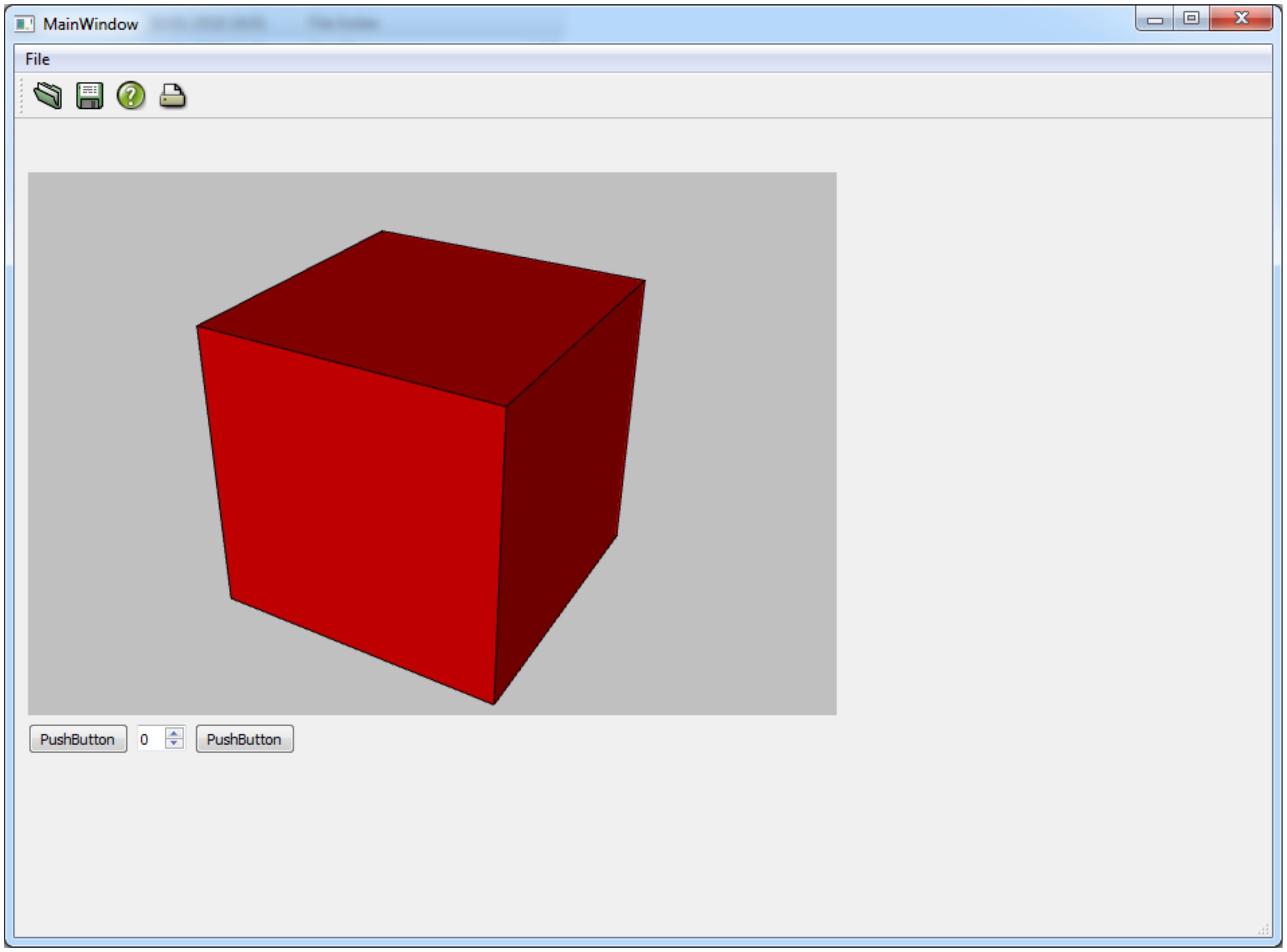


Figure 8: QtVTK Example

You can see that almost all of the VTK code remains the same as for the previous example so you can easily use all of VTK's features from within Qt. The following examples will try to get you to add features to the example program and once you are comfortable with this, you can begin to write your own software application within your groups.

Exercise 3

The examples below explain how to use VTK to implement some of the features your final ModelViewer application should contain. You should attempt these exercises in your individual repository first so that you all learn a bit about these features of VTK. You do not need to have completed everything, see the individual marking scheme for what is actually required. The idea is that you all practice the basics in your individual repo, then move on to implementing the features properly in the group ModelViewer repo.

The examples you have just completed, and most of those below, have been adapted from this VTK Examples website https://lorensen.github.io/VTKExamples/site/. There are lots of examples on this site that you can use to find out what is possible with VTK, or to understand how to use the VTK libraries to achieve certain objectives.

Rendering (and loading) simple models using your software library When you have developed your group's software library (with models consisting of pyramids, tetrahedrons and hexahedrons). Can you build in functionality that allows your software to draw solids represented by classes in your library? Code to do this can be found in the following examples:

- Pyramid https://lorensen.github.io/VTKExamples/site/Cxx/GeometricObjects/Pyramid/
- Tetrahedron https://lorensen.github.io/VTKExamples/site/Cxx/GeometricObjects/Tetrahedron/
- Hexahedron https://lorensen.github.io/VTKExamples/site/Cxx/GeometricObjects/Hexahedron/

If you managed to implement the file parser aspect of the library, can you use VTK to render pyramids, tetrahedrons and hexahedrons loaded from a file? In the previous project week you also developed some Qt code that used a QtFileDialog to save a file. Can you modify this code to allow the user to select the input file using the GUI? Use QFileDialog::getOpenFileName to create an *Open File* Dialog:

```
QString fileName = QFileDialog::getOpenFileName(this, tr("Open STL File"), "./", tr("STL Files
(*.stl)"));
```

You can generate a *C* Style string (that can be passed to file loading functions) from a QString using:

```
const char *c_str = fileName.toLatin1().data();
```

Loading CAD Files (STL Format) A key requirement for your final software application is that it can load models in STL format, a format commonly used by CAD packages and 3D printers. Can you use VTK's STLReader class to load and display an STL file (there is a folder on Moodle containing some example STL files). There is an example of how to load an STL file here: https://lorensen.github.io/VTKExamples/site/Cxx/IO/ReadSTL/. Some further hints:

- As above, use Qt `QFileDialog::getOpenFileName` to run an open file dialog.
- You may need to consider adding some VTK objects/pointers as memmbers of the mainWindow class. For example you'll need to keep track of the vtkRenderer that you add in the constructor so that you can add/remove actors and props when the user opens an new STL model file.
- Refer to the VTK documentation for relevant classes (e.g. vtkRenderer, vtkViewPort to understand how to clear the renderer of existing actors/props.

Let the user modify the VTK visualisation using Qt input widgets Setup the user input widgets so that they modify the model or its rendering in some way. This is the starting point for the development of your software. Some ideas & hints are given below, you don't need to implement all of the features listed here and there are many other alternative features that you could choose to add. Make a decision based on what you think is best for your group's final application.

- When a button is clicked the camera view is preset to a fixed orientation. See documentation for the vtkCamera class, and documentation for the vtkRenderer class for how to get a pointer to the active camera.
- When a button is pressed the colour of the actors in the scene is changes red->blue->red.

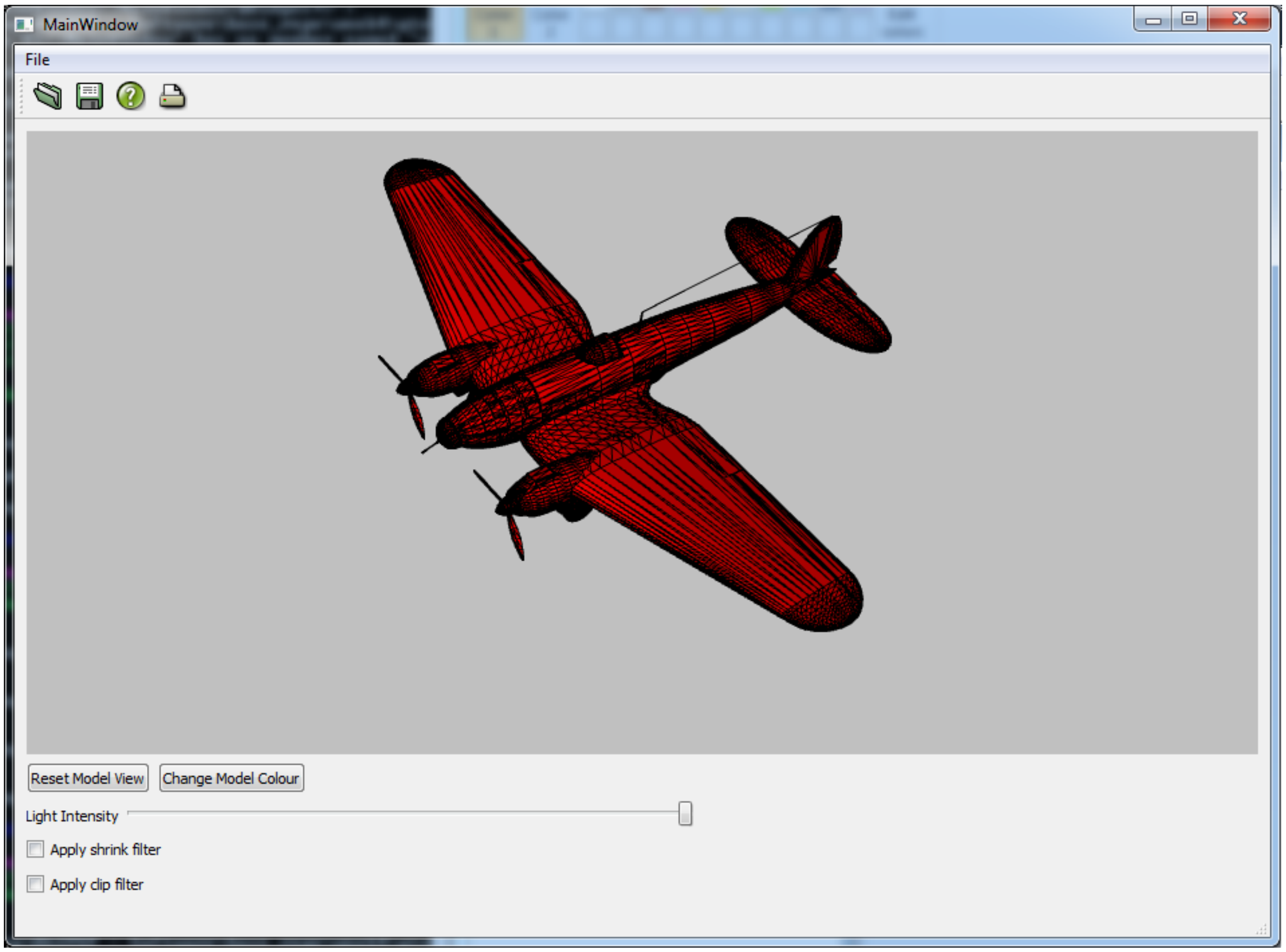


Figure 9: Load and Render an STL Model

- To change the properties of an actor (e.g. colour of the STL model actor), you'll either need to store your own vtkSmartPointer to the actor or retrieve a pointer to the actor from the vtkRenderer. Use vtkActorCollection* vtkRenderer::GetActors() to retrieve pointers to all actors in the scene.
- Can you do the same for the background colour?
- You could allow the user to choose a colour using a QColorDialog.
- Add lights to the rendered and have user input widgets that are able to change the properties of these lights. Lights are added to the renderer in the constructor e.g.:

```
vtkSmartPointer<vtkLight> light = vtkSmartPointer<vtkLight>::New();
light->SetLightTypeToSceneLight();
light->SetPosition( 5, 5, 15 );
light->SetPositional( true );
light->SetConeAngle( 10 );
light->SetFocalPoint( 0, 0, 0 );
light->SetDiffuseColor( 1, 1, 1 );
light->SetAmbientColor( 1, 1, 1 );
light->SetSpecularColor( 1, 1, 1 );
light->SetIntensity( 0.5 );

// now add actors ...

// Render
ui->vtkWidget->GetRenderWindow()->Render();

// add the light to the renderer after Render was called
renderer->AddLight( light );
```

You can use vtkLightCollection* vtkRenderer::GetLights() elsewhere in your code to retrieve a list of the lights in your `vtkRenderer` and make changes to properties such as light position, direction, intensity or colour.

- Filters: VTK allows you to apply filters to the model to change what is rendered. Filters are isnerted into the rendering pipeline between the data source and the mapper. All VTK data sources, filters and are subclasses of `vtkAlgorithm` which means they have `SetInputConnection()` and `GetOutputPort()` methods. In Example 1 and Example 2, the output port of the data source was connected directly to the input connection of the mapper using these functions:

```
MAPPER->SetInputConnection( SOURCE->GetOutputPort() );
```

```
Filter classes can be inserted into this chain to affect what is rendered. The resulting code could look s
```

```
FILTER1->SetInputConnection( SOURCE->GetOutputPort() );
FILTER2->SetInputConnection( FILTER1->GetOutputPort() );
MAPPER->SetInputConnection( FILTER2->GetOutputPort() );
```

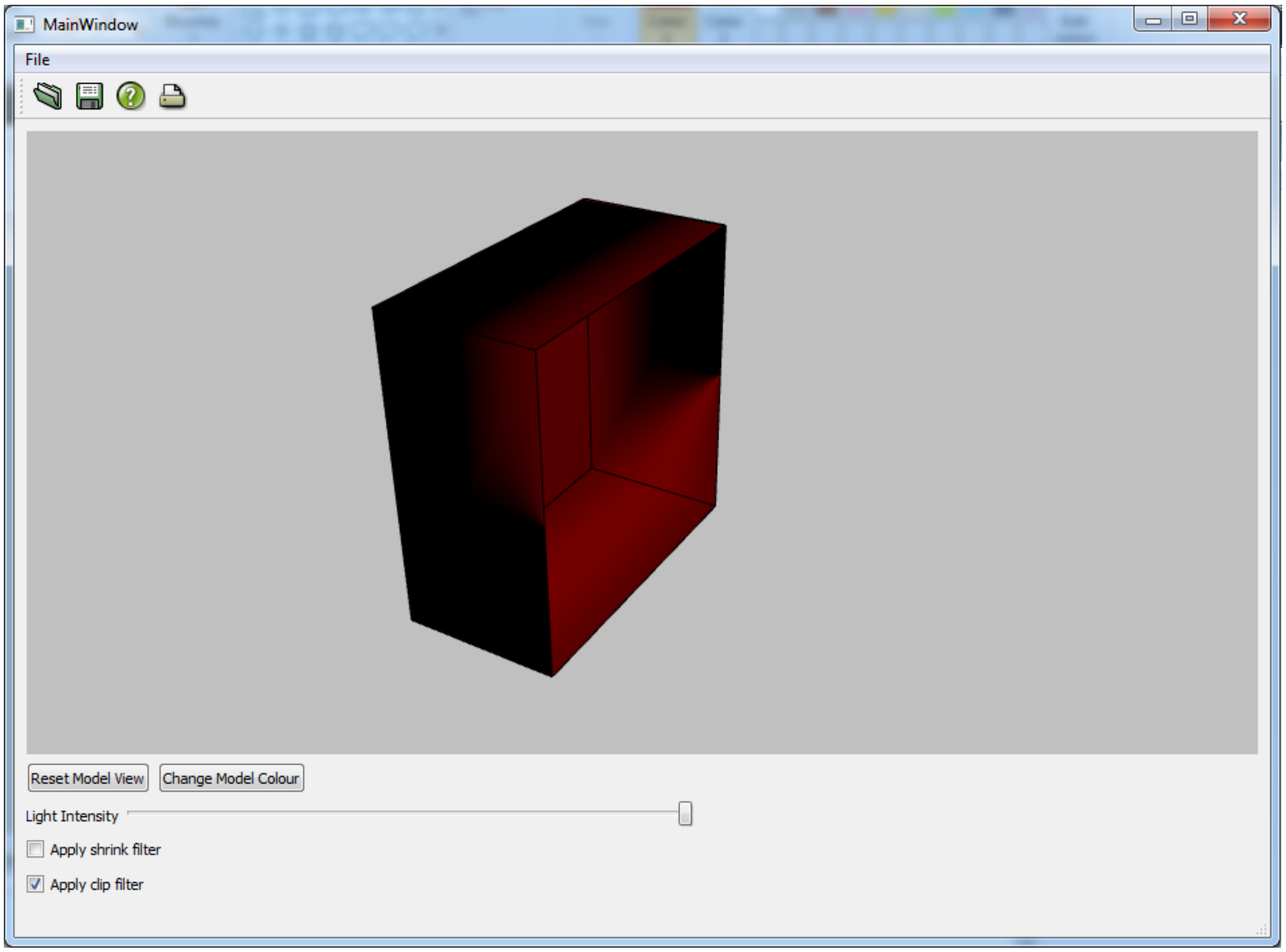
Obviously the filter classes need to be configured and there a lots of different filter options that you can choose from. Examples include:

- Clip Filter

```
// this will apply a clipping plane whose normal is the x-axis that crosses the x-axis at x=0
 vtkSmartPointer<vtkPlane>  planeLeft = vtkSmartPointer<vtkPlane>::New();
 planeLeft->SetOrigin(0.0, 0.0, 0.0);
 planeLeft->SetNormal(-1.0, 0.0, 0.0);

 vtkSmartPointer<vtkClipDataSet> clipFilter = vtkSmartPointer<vtkClipDataSet>::New();
 clipFilter->SetInputConnection( SOURCE->GetOutputPort() ) ;
 clipFilter->SetClipFunction( planeLeft.Get() );

 MAPPER->SetInputConnection( clipFilter->GetOutputPort() );
```



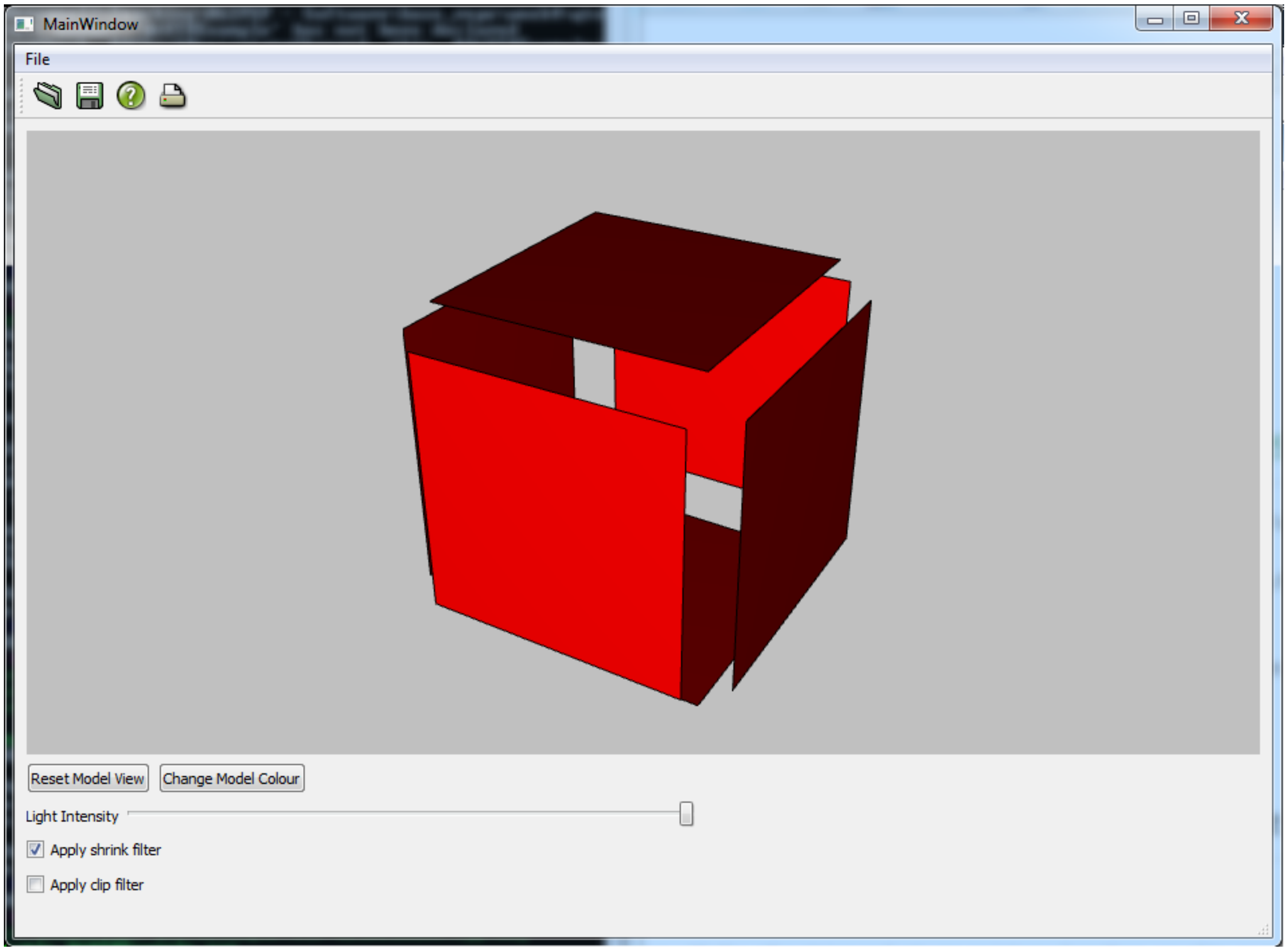
- [ ] Apply shrink filter
- [✓] Apply clip filter

Figure 10: Clip Filter

- Shrink Filter

```
vtkSmartPointer<vtkShrinkFilter> shrinkFilter = vtkSmartPointer<vtkShrinkFilter>::New();
shrinkFilter->SetInputConnection( SOURCE->GetOutputPort() );
shrinkFilter->SetShrinkFactor(.8);
shrinkFilter->Update();

MAPPER->SetInputConnection( shrinkFilter->GetOutputPort() );
```



☑ Apply shrink filter

☐ Apply clip filter

Figure 11: Shrink Filter

Start your Group Software Engineering Exercise

Once you feel confident that you could implement some of these features, and know how you would need to structure the program's code to do so, start to develop your group's application. Start to plan your group's software application and create some source file templates on a Github repo. Divide the work between your group's members and get developing! You may wish to copy some of the the exercise 3 code from one group member into the group repository to serve as a starting point so you have a basic framework for each member to start modifying. See the separate document describing the group ModelViewer development exercise for details of what you need to complete as a group.

Keep in mind the design brief and also the marking criteria, be sure to check the assessment description and mark schemes on Moodle.