EEEE2076 - Software Development Group Design Project

Worksheet 7 - 3D Rendering with VTK, in Qt and VR

P Evans

February 10, 2025

Contents

1 Visualisation Toolkit (VTK)

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

1.1 Prerequisite - Getting the Valve Software OpenVR Library

VTK will access the VR hardware via an intermediate interface library called OpenVR. The library allows software like VTK to access different VR/MR/AR hardware without needing to understand exactly how the hardware works, OpenVR effectively acts as a translator.

Although you will not use the VR hardware on your system, you need to be able to compile VR enabled code to allow you to fix basic programming errors before you transfer it over to the VR systems. This means you need to download the OpenVR library.

OpenVR and VTK are pre-installed on the VR PCs at the locations indicated in this document.

- Download a zip of the OpenVR Software Development Kit (SDK) from Valve's Github repository: https://github.com/ValveSoftware/openvr.
- Unzip the file, find the parent directory containing the *bin*, *codegen*, *headers*, etc sub folders and rename this something sensible like *OpenVR*. Copy the renamed directory somewhere where it is easy to find, e.g. I used *D:\OpenVR*
- Remember where you put the folder, you'll need it in a minute.
- Add the relevant binary subdirectory to your path, for example: If you have installed OpenVR at *D:\OpenVR*, the binary subdirectory is *D:\OpenVR\bin* however, you then need to select another subdirectory according to the operating system you are using. This is most probably Windows-64bit so the folder that needs adding to the path is: *D:\OpenVR\bin\win64*

2 Obtaining the VTK library

Visit the VTK website and download the latest source zip file (note: the latest version might not be available in a zip, you might need to download the .tar.gz archive and use a utility like WinRAR to extract). Extract this somewhere, this your source folder for CMake.

Then you need to create a build folder somewhere, it is a good idea to name this according to the VTK version you have downloaded and also the version of Visual Studio you will be compiling with., E.g. *VTK9.2_MSVC2022*.

Now it is just a case of going through the CMake *Configure, Generate, Build* process. Open CMake GUI and browse to your source and build folders.

Click configure, making sure the correct version of Visual Studio has been selected as the compiler. If you just want to build a standard version of VTK then you could immediately go on to Generate and Build in Visual Studio but you need more than a standard VTK build as the VR capabilities are not included by default. **To make VTK compatible with VR and Qt you just need to select a few options in the list of options than CMake has provided before generating. See below and Fig. 1/2. If you don't do this step now, you will end up recompiling VTK later!**

- In the list of options, example the VTK group
- Find the VTK_MODULE_ENABLE_VTK_RenderingOpenVR option
- change the option from Default to YES

You will also need to configure VTK so that it is compatible with Qt as Qt will be used as the GUI toolkit for the base-station, follow the instructions below and see Fig. 2.

- Find *VTK_Group_Enable_Qt* and change it to *Yes*.
- Click Add Entry in the CMake Window, select Path, enter the Name as CMAKE_PREFIX_PATH and value as D:/Qt/6.2.4/msvc2019_64 (**adapt this path to suit your Qt install location**).
- Click Add Entry in the CMake Window, select File Path, enter the Name as QT_QMAKE_EXECUTABLE and value as D:\Qt\6.2.4\msvc2019_64\bin\qmake6.exe (**adapt this path to suit your Qt install location**).

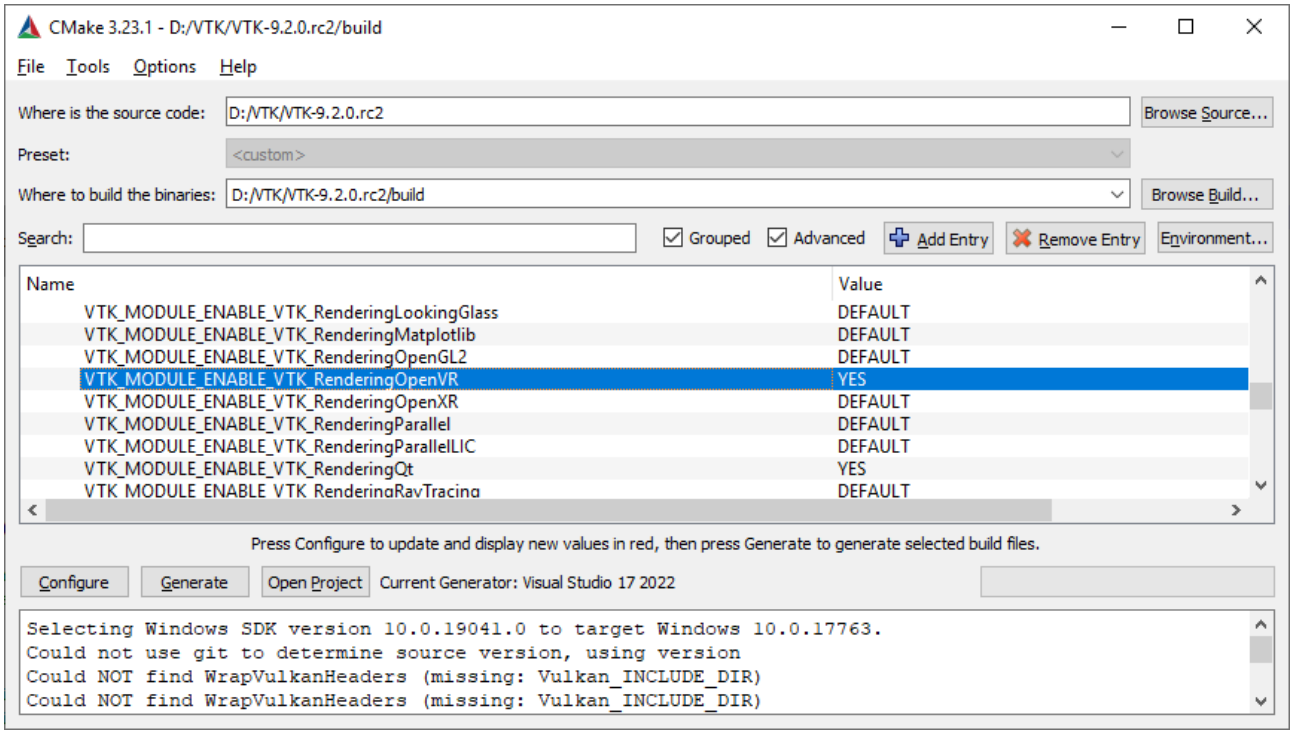

Figure 1: CMake Option to Enable OpenVR Rendering for VTK

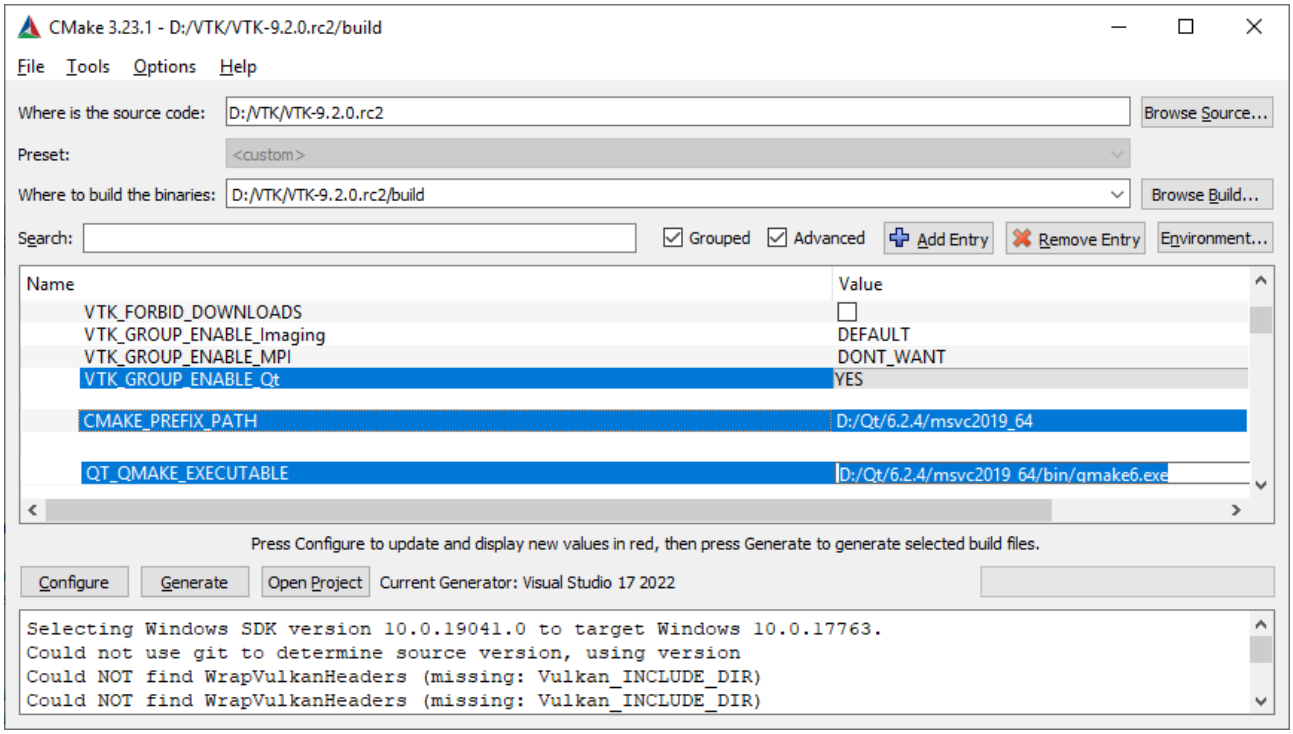

Figure 2: CMake Option to Enable Qt Rendering for VTK

Then click *Generate*, CMake will have tried to find the OpenVR library but will probably have failed. There will be two new 'path' options for the OpenVR library which will be listed as 'not found'. These options are shown indicated in Fig. 3 along with the paths I had to set to get it to work. My OpenVR was installed at *D:\OpenVR* and the paths reflect this, you may need to edit the paths depending on where you put OpenVR.

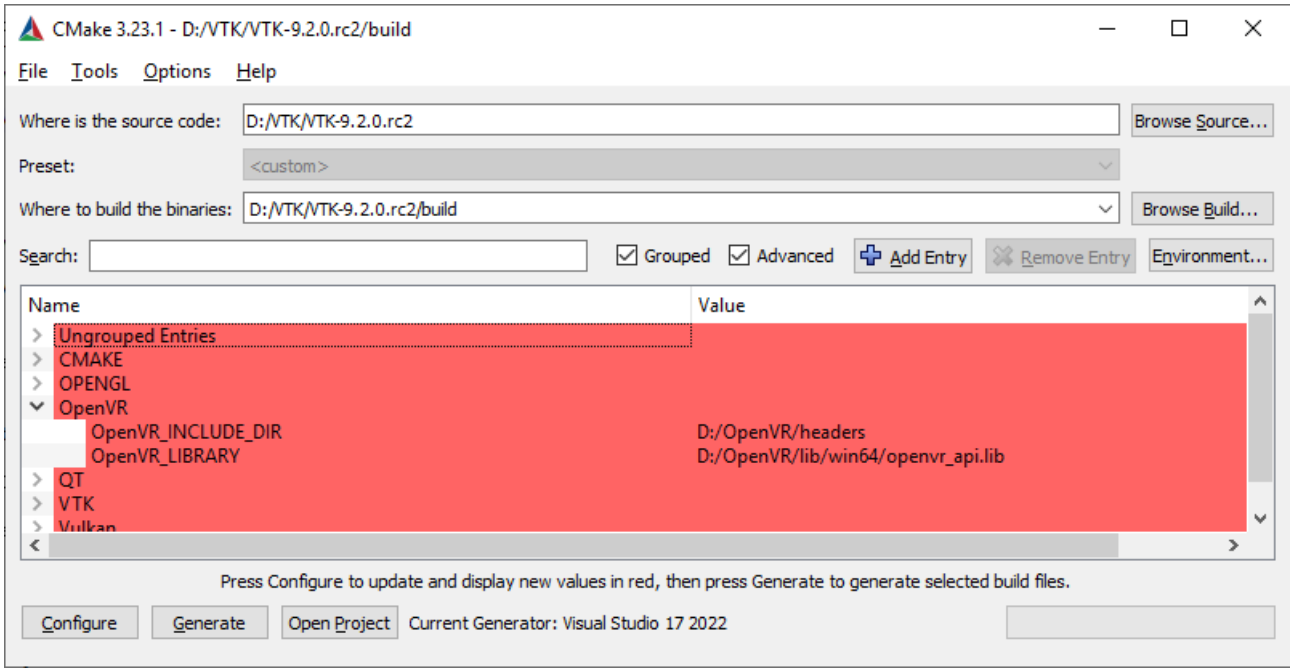

Figure 3: Paths for OpenVR

Then click *Generate* again and if the paths are ok, CMake will generate your Visual Studio project. Finally clock *Open Project* to open the newly created Visual Studio project. In Visual Studio, do a *Build All*. This build can take over half an hour on a reasonable spec PC, and much longer on an old or low spec machine.

You need to install VTK when it has finished building. There is an *INSTALL* target in the Visual Studio project that will do this for you. Right click on it and select build, this will probably fail with an error about the *SetLocal* command. The error is because Visual Studio is trying to copy files to the Program Files directory which requires administrator privileges. Close Visual Studio, run as an administrator, and try again. Make sure you know where VTK was installed to (it should be *C:\Program Files\VTK*) but check.

Add the newly installed binary directory *C:\Program Files\VTK\bin*) to the system path, otherwise your VTK applications will fail to find the VTK DLLs and will not run.

3 Initial Examples to Check Setup

3.1 A Basic VTK Application

First you need to check that your VTK library is working properly. We'll use of the basic VTK examples provided on the VTK examples page to do this:

https://kitware.github.io/vtk-examples/site/Cxx/GeometricObjects/CylinderExample/

3.1.1 Exercise 1

Download the example, you should have a C++ source file and a CMakeLists.txt. Put these in a *Worksheet7/Exercise1* folder in your individual repository. Run CMake and Visual Studio to build the project, then run it. You should see a cylinder like in Figure 4. If you have problems, think about the following:

- When *configuring*, CMake needs to find your VTK library. It might not be able to do this if the *bin* sub-directory of your VTK install (usually *C:\Program Files\VTK\bin*) isn't on the path. If you add this folder to the path now, you will need to restart CMake before it picks up the change.
- When you click *Run* or *Debug* in Visual Studio, it needs to know what you want to run or debug. You tell it this by clicking on the relevant sub-project and selecting *Set as Startup Project*
- If your project links to a dynamic library like VTK (remember that most of VTK is a set of DLLs), your program needs to find these at run-time. This means the DLL location needs be on the path!

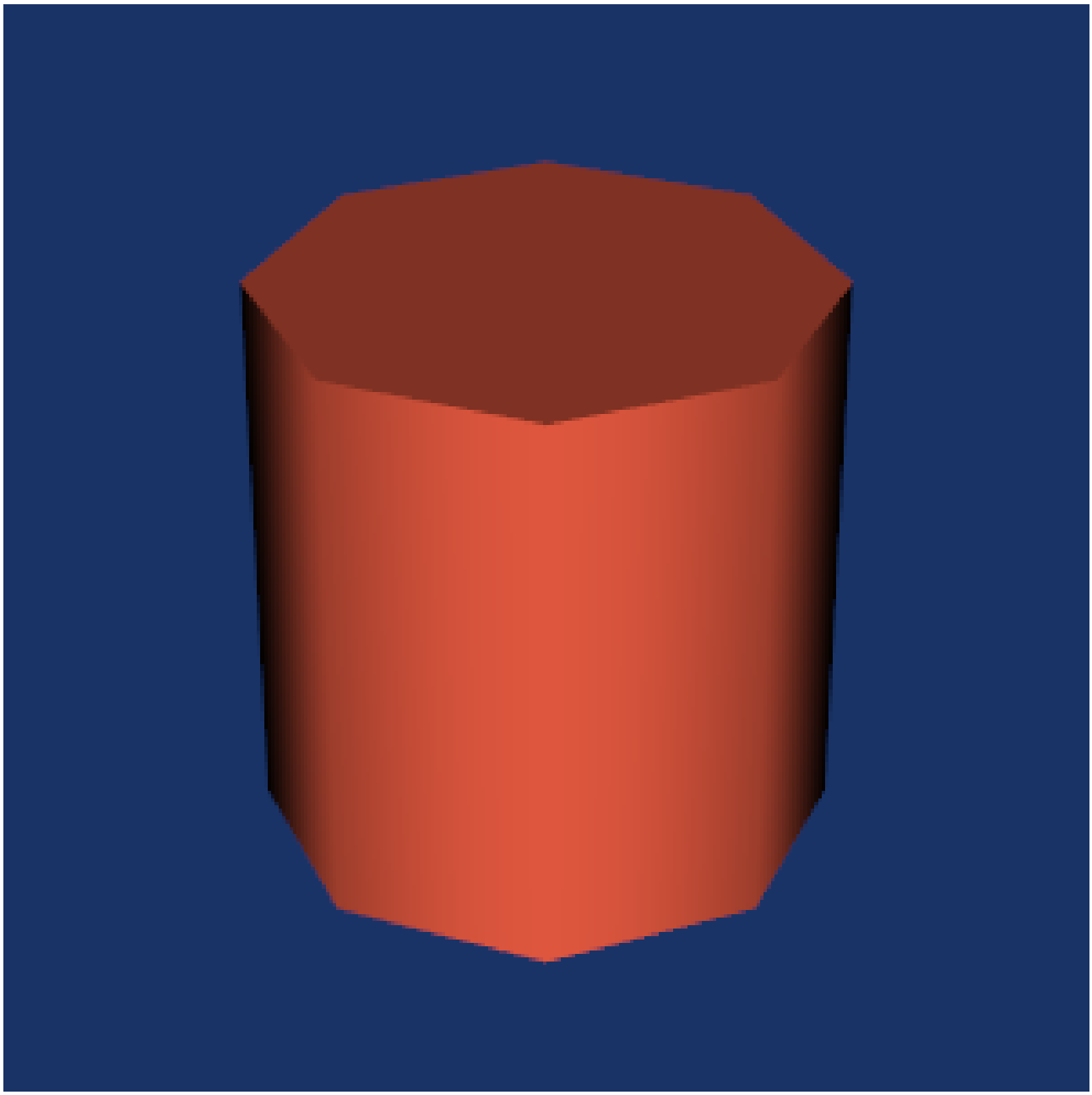

Figure 4: The VTK Cylinder Example

3.2 VR visualisation with VTK

A modified version of the VTK Cylinder application is provided in my Github repositories. This replaces the standard VTK Renderer and RenderWindow classes with OpenVR equivalents in the source code. It also makes a change to the CMakeLists.txt to tell CMake that the VTK_RendererOpenVR packages is needed. Perform the following steps:

3.2.1 Exercise 2

- Download / clone the modified VR Cylinder Repository here.
- Look at the source and CMakeLists.txt. Make sure you understand the differences as you will need to create your own projects later.
- *Configure* the project in CMake GUI. You are likely to get errors relating to VTK and OpenVR. Remember that CMake's main job is just to find libraries that are needed to build your project so it will be telling you that it can't find VTK / OpenVR.
  - For VTK - making sure your VTK bin folder (usually *C:\Program Files\VTK\bin*) is on the path is usually enough. If you add this folder to the path now, you will need to restart CMake before it picks up the change.
  - For OpenVR - after the first attempt to build your software, CMake will tell you what it tried to find and couldn't. Go to the OpenVR options that have appeared and set them to the relevant paths for your system - the paths I used are shown in Figure 5.
- When the project generates correctly, open in Visual Studio and try to build and run. For build errors you will need to interpret the error messages but because the code is known to be ok, errors are likely to be due to libraries or headers that cannot be found and will related to the CMake config. Errors when running the program are probably because VTK or OpenVR DLLs cannot be found, these are usually path errors ('bin' folders not added to system path).
- Once you are happy that you have a valid VTK OpenVR project that will compile on your computer:
  - Commit and push to Github.
  - Go to one of the VR systems, clone your individual repository (Use the documents folder rather than somewhere like *C:* as it will prevent other users of the PC being able to access your code)
  - Run CMake, Configure, Generate, Build in Visual Studio. OpenVR, VTK and Qt are all installed on the PCs, either in the Program Files directory or on the *D:* drive.
  - The application should output on the VR system when you run it, this cylinder application isn't particularly interesting but you are just checking that you are able to develop code, transfer it to the VR PCs, and successfully run it.

Important: Look at the CMakeLists.txt for both of the previous examples - there are some extra lines that are required to find and link against the VTK library (find_package(VTK), include(VTK_USE_FILE), target_link_libraries(...)). Make sure you remember to copy these to your own CMakeLists in the next exercise.

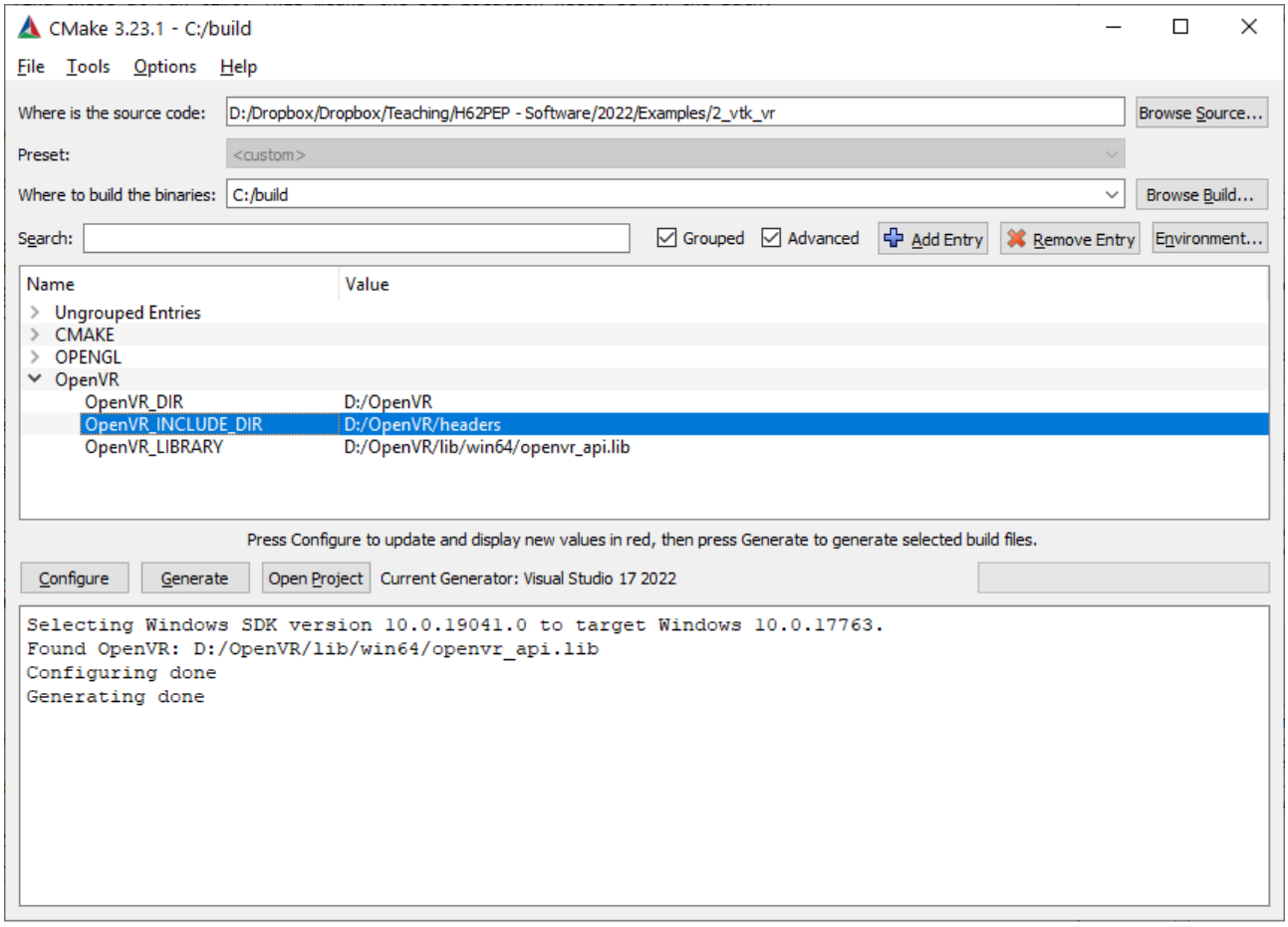

Figure 5: Setting OpenVR Options for CMake

3.3 VTK from within Qt

The aim of this project is to create a VTK enabled PC application that uses Qt for PC GUI. VTK will be used in two ways, one to allow the CAD models to be visualised and configured on the PC, and subsequently for these CAD models to be rendered using VR. You will notice from the Examples 1 and 2 that the VTK rendering process for a native VTK application and a VT application are basically the same - data source - mapper - actor - renderer - renderwindow, add a camera to define the viewpoint, add an interactor to allow user interaction. The only real difference is that the exact data type of some of these classes changes (e.g. the renderer or render window). The same is true for the Qt enabled VTK application - same process, a few minor differences.

3.3.1 Exercise 3 - Adding VTK and OpenVR libraries to your Qt GUI appliction

The aim here is to add the cylinder rendering code to your Qt application, and to modify the CMakeLists.txt to support VTK and OpenVR.

The first thing that needs to be done is to 'promote' the container 'QWidget' we added as a placeholder to a 'QVTKOpenGLNativeWidget'. To do this:

- Create a copy of your final exercise from Worksheet 6 in a new Worksheet7 subfolder
- Copy the *vrbindings* folder from the previous example into this example
- Open the mainwindow.ui file in QtCreator, select the QWidget in your user interface design, right click, select 'Promote to...'
- Make the promoted class name 'QVTKOpenGLNativeWidget' and allow Qt to automatically set the header filename (you might want to check the capitalisation on the file name, the file will be in your VTK\include folder (in Program Files) and if you look you will see capital letters are used. This won't make any difference on Windows but if you were to compile on another system it could cause problems. (Fig. 6)

Now you need to modify the CMakeLists.txt that was copied from Worksheet6. Three additions are required:

- Add *find_package* line to the top of the file, this tells CMake that it needs to search for VTK on your system:

```
# Include all of VTK - previous example specified individual components but this
    is just likely to cause problems
find_package( VTK REQUIRED )
```

- Add the variable ${VTK_LIBRARIES} to the list of libraries that CMake will link to your application, find your *target_link_libraries* statement and modify by adding ${VTK_LIBRARIES as an argument:

```
target_link_libraries(BaseStation PRIVATE Qt6::Widgets ${VTK_LIBRARIES})
```

  The first parameter to *target_link_libraries* is the name of your application and must match what is specified in the *add_executable* statement - I have called mine *BaseStation*, your name may differ.

- Add the following to the end of CMakelists.txt file so that the VR controller bindings are copied to the build folder.

```
# Copy across Open VR bindings that map controllers
# The program will expect to find these in the build dir when it runs
add_custom_target(    VRBindings )
add_custom_command(  TARGET VRBindings PRE_BUILD
COMMAND ${CMAKE_COMMAND} -E
                copy_directory ${CMAKE_SOURCE_DIR}/vrbindings ${CMAKE_BINARY_DIR}/ )
```

All of the above is just merging the VTK/Open VR lines from the previous example into the Qt application. There is an example CMakeLists.txt here if you want a reference.

Try to build. If you get error messages from CMake/Visual Studio then try to work out what they mean and fix the problem yourself. Two common errors and possible reasons for them are given below as some initial help:

- *Cannot find QVTKOpenGLNativeWidget.h* - CMake has either not found VTK, or has not found the relevant **.h*. This could be because you:
  - have not added *find_package(VTK REQUIRED)*
  - specified the specific VTK components with *find_package(VTK component_name1 component_name2 ...)* and not included the correct ones
  - didn't compile VTK with the Qt options correctly set, which means the Qt VTK Widget classes were not built.
- Errors relating to a *Custom Build*
  - you haven't set the paths to OpenVR in CMake
  - you haven't copied across the OpenVR bindings folder

If it all works, your container widget should now have a black background (Fig. 7).

3.3.2 Exercise 3 - Rendering the cylinder in your application

The following exercise just asks you to merge the cylinder rendering with your Qt application. It will also divide the VTK code into essential renderer setup, and object related class (mapper/actor) for the next stage.

Note: I have changed the name of my vtkOpenGLNativeWidget from 'widget' to 'vtkWidget' to avoid confusion, the following code reflects this. Remember, anywhere in the code where you see ui->xxxxxx: the xxxxxx part refers to the name that you have given an widget in Qt creator. If you have a different widget name, your code will need to be slightly different!

The code you need to add to the constructor is below. You will also need some #include statements, the basic rule is that for every VTK class with the name vtkXXXXX, you need to #include <vtkXXXXX.h>. You can work out exactly what needs to be included.

```
/* This needs adding to MainWindow constructor */
/* Link a render window with the Qt widget */
renderWindow = vtkSmartPointer<vtkGenericOpenGLRenderWindow>::New();
ui->vtkWidget->setRenderWindow(renderWindow);

/* Add a renderer */
renderer = vtkSmartPointer<vtkRenderer>::New();
renderWindow->AddRenderer(renderer);

/* Create an object and add to renderer (this will change later to display a CAD
    model) */
/* Will just copy and paste cylinder example from before */
// This creates a polygonal cylinder model with eight circumferential facets
// (i.e, in practice an octagonal prism).
vtkNew<vtkCylinderSource> cylinder;
cylinder->SetResolution(8);

// The mapper is responsible for pushing the geometry into the graphics
// library. It may also do color mapping, if scalars or other attributes are
// defined.
vtkNew<vtkPolyDataMapper> cylinderMapper;
cylinderMapper->SetInputConnection(cylinder->GetOutputPort());

// The actor is a grouping mechanism: besides the geometry (mapper), it
// also has a property, transformation matrix, and/or texture map.
// Here we set its color and rotate it around the X and Y axes.
vtkNew<vtkActor> cylinderActor;
cylinderActor->SetMapper(cylinderMapper);
cylinderActor->GetProperty()->SetColor(1., 0., 0.35);
cylinderActor->RotateX(30.0);
cylinderActor->RotateY(-45.0);

renderer->AddActor(cylinderActor);

/* Reset Camera (probably needs to go in its own function that is called whenever
    model is changed) */
renderer->ResetCamera();
renderer->GetActiveCamera()->Azimuth(30);
renderer->GetActiveCamera()->Elevation(30);
renderer->ResetCameraClippingRange();
```

```
/* This needs adding to MainWindow class definition */
private:

   vtkSmartPointer<vtkRenderer> renderer;
   vtkSmartPointer<vtkGenericOpenGLRenderWindow> renderWindow;
```

4 Opening and Rendering a CAD File

This examples is intended to be the basis of your final application (excluding VR capability). The main features that need to be added are:

- Load CAD files in STL format and render.
- Use TreeView to represent loaded CAD files.

4.0.1 Exercise 4 - Opening a CAD file, updating the renderer

There are three required modifications:

1. In the function that is called when someone clicks the *Open File* action:
   - Get the filename selected.
   - Add a new item to the tree view (see code in constructor) - ideally you'd add the item as a child of the currently selected item.
   - Call the loadSTL() function of the newly created item to ask it to load from the STL file.
2. In the loadSTL() function of the ModelPart class
   - Take the filename provided as an argument and create a vtkSTLReader
   - Create a mapper and link it to the STL reader
   - Create an actor and link it to the mapper
   - If you look at the template for the ModelPart class (in header file) you will notice that there are variables that can be uncommented to store pointers to the STL reader, mapper and actor. Uncomment and use these.
   - the VTK ReadSTL example might be useful as a guide.
3. In the MainWindow class:
   - You need a function that can be used to refresh the VTK rendering each time, something like the code listed below. This: removes any existing actors from the renderer, calls a function to add any actors from the ModelParts, forces an immediate update of the renderer.

```
void MainWindow::updateRender() {
    renderer->RemoveAllViewProps();
    updateRenderFromTree(partList->index(0, 0, QModelIndex() ) );
    renderer->Render();
}
```

   - The missing component here is the updateRenderFromTree() function. This needs to iterate through the tree and add the actor for each ModelPart to the renderer. A basic framework for the function is shown below but you'll need to modify to get it to work.

```
void MainWindow::updateRenderFromTree( const QModelIndex& index ){
    if( index.isValid() ) {
        ModelPart* selectedPart = static_cast<ModelPart*>(index.internalPointer
    ());
        /* Retrieve actor from selected part and add to renderer */
    }

```

```
7     /* Check to see if this part has any children */
8     if( !partList->hasChildren(index) || (index.flags() & Qt::
      ItemNeverHasChildren) ) {
9        return;
10    }
11
12    /* Loop through children and add their actors */
13    int rows = partList->rowCount( index );
14    for (int i = 0; i < rows; i++) {
15       updateRenderFromTree(partList->index(i, 0, index));
16    }
17 }
```

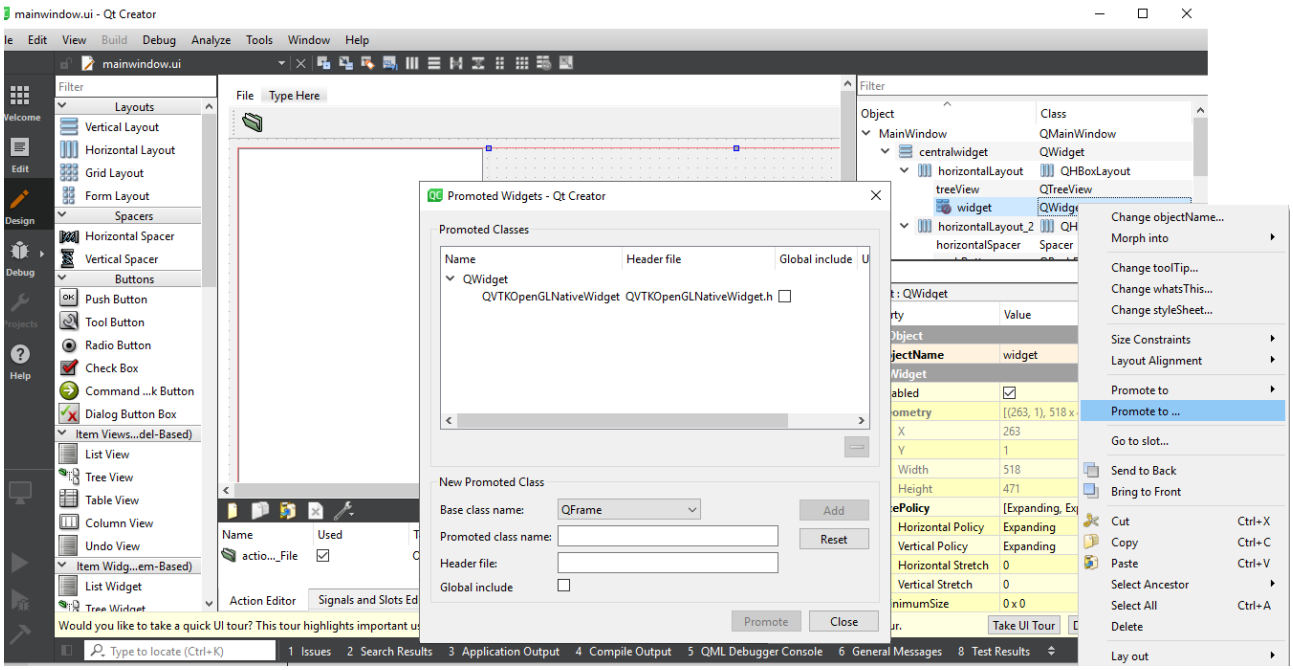

Figure 6: Promoting the empty container QWidget to a QVtkOpenGLNativeWidget

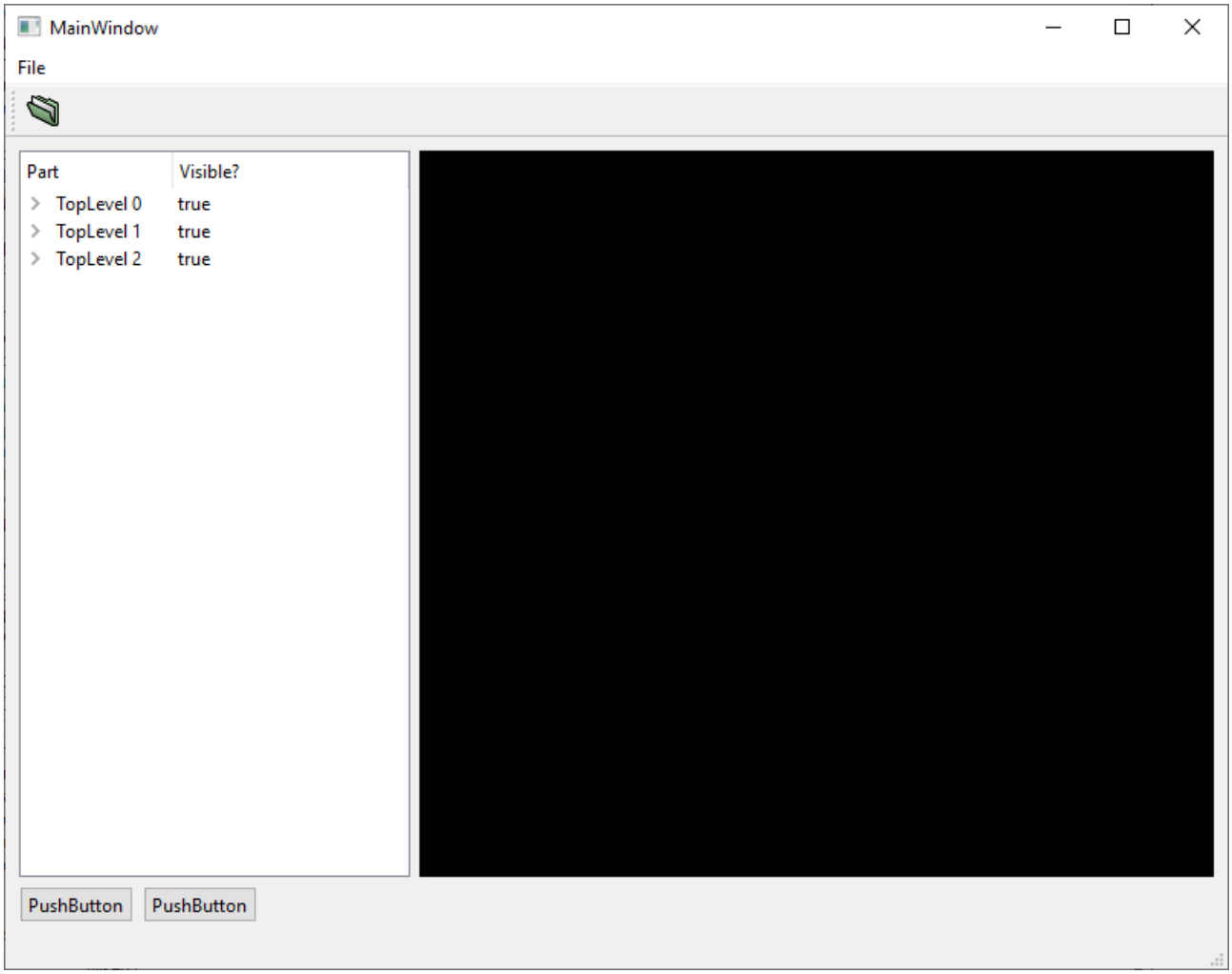

Figure 7: Application should look like this if VTK integration is working

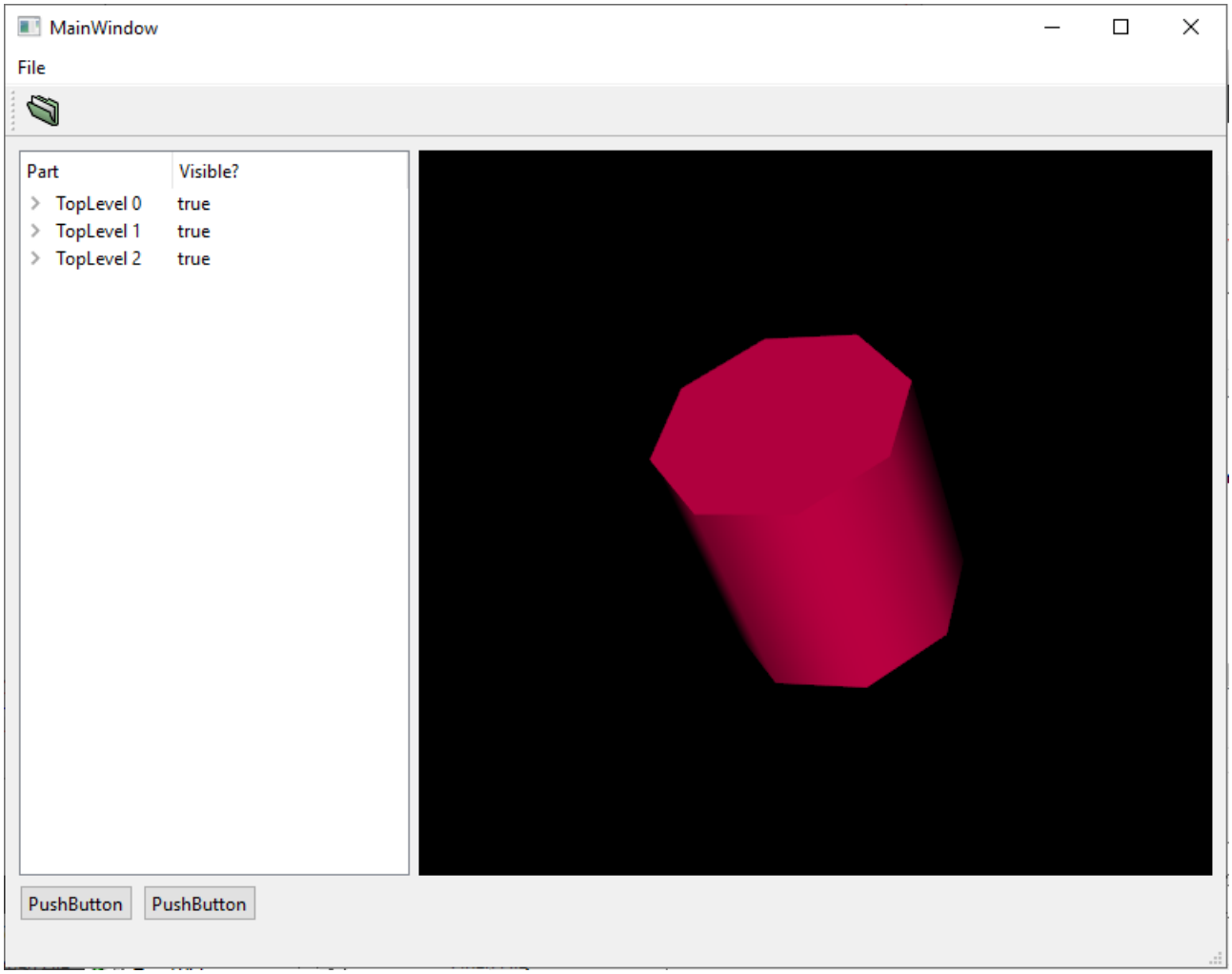

Figure 8: Qt application with embedded VTK widget