Contents

1 The Build Syndrome 1
1.1 Introduction 1
1.2 Using Utility Libraries 3
1.3 OpenGL Agnosticism 5
1.4 Configuration Spaces 8
1.5 Meta-builds and CMake 9
1.6 CMake and the Configuration Space 10
1.7 CMake and Platform Specifics 15
1.8 Conclusion 21
Bibliography 21

The Build Syndrome

Jochem van der Spek and Daniel Dekkers

1.1 Introduction

In the current age of desktop, mobile, and console computing the number of platforms, operating systems, and OpenGL versions that are in active use has become so large that developing and deploying our application for all those different configurations has become a trying and time consuming part of development. When a game studio wants to release their latest title on as many platforms as possible, they will need to manage a combinatoric explosion of all the different configuration parameters.

	OpenGL(1.0-4.2)	OpenGL|ES(1.0/2.0)
OSX desktop	GLUT, QT, wxWidgets, X11	QT
iOS embedded	N/A	CoreAnimation (iOS2.0/3.0)
SGI desktop	GLUT, QT, X11	EGL
Windows desktop	GLUT, QT, wxWidgets, EGL	EGL
Windows embedded	N/A	EGL, QT
Unix desktop	GLUT, QT, wxWidgets, EGL	EGL
Linux desktop	GLUT, QT, wxWidgets, X11, EGL	EGL
Linux embedded	N/A	EGL/QT
Android embedded	N/A	Android(1.0/2.2), EGL
Symbian embedded	N/A	EGL, QT
Blackberry embedded	N/A	BlackberryOS(5.0/7.0)
Webbrowsers	N/A	WebGL(ES2.0 only)

Table 1.1. Overview of the main OpenGL implementations on the various platforms.

We recognize two steps in the process to reduce the complexity of this task. The first is to write OpenGL agnostic code, meaning that the code encapsulates the platform- and OpenGL version specific details into classes that are fully transparent to any combination of platform and OpenGL version. The second method is to use a meta-build system that wraps all that code into a usable project for many different IDE's on many different platforms. Each platform comes with it's own set of APIs for creating a window to draw. Some of these APIs support OpenGL|ES for Embedded Systems, even on non-embedded desktop platforms such as the iPad simulator on OSX, while some only support the OpenGL version that is enabled by the OpenGL drivers on that platform. Different OpenGL implementations on various platforms can be seen in Table 1.1. A complete list can be found on the OpenGL.org website [implementations]. A completely different way of achieving the same goal is to use JavaScript with WebGL and is described in detail in **??**. In this article we will focus on C++/Objective-C.

	OpenGL1.0-4.2	OpenGL|ES1.0-2.0
iOS	N/A	CoreAnimation(iOS2.0/3.0)
Windows	GLUT, QT, EGL	EGL
OSX	GLUT, QT	N/A

Table 1.2. The selection of APIs and subset of platforms used in this article.

As a demonstration, we show how to implement a very minimalistic OpenGL program on a subset of all the possible platforms, as can be seen in Table 1.2. For the sake of simplicity, we further limit ourself to considering only APIs that interface the creation of the so-called OpenGL Drawing Context, which specifies to the operating system how a pixel is to be drawn to the screen. See http://www.opengl.org/wiki/Creating_an_OpenGL_Context for a more extensive discussion of this topic.

In describing the use of our selection of APIs on the subset of platforms we draw from the experience of writing the 'RenderTools' [RenderTools] software library. The library was created over the course of the past three years (2008-2011) to serve as a codebase to create any conceivable OpenGL application on many platforms. We tried to keep the classes lightweight and the library in it's simplest form depends as little as possible on external libraries. We used the whimsically named Extension Wrangler Library, or GLEW [GLEW], to manage the various OpenGL extensions on each platform. Many deprecated math functions such as glRotate, glOrtho, glPerspective were implemented in OpenGL compliant Matrix classes. Currently only GLfloat types are supported, though abstraction of the type to a compiler setting is on the wish-list. There are many such open issues, but we believe that the design of the library is sound and can be built upon and extended by the community, hence we release it under the GNU Public License (GPL) which ensures open source distribution but we also allow binary distribution under licenses that are free for artists, charities, contributors and educators.

1.2 Using Utility Libraries

Where the predecessor to OpenGL, Silicon Graphics' IrisGL, had functions to create a window to draw in, OpenGL does not. This makes OpenGL portable to different operating systems and Windows-APIs but also makes it difficult to set up without intimate knowledge of the underlying Windows-API of the platform at hand. Each platform has it's own platform-specific implementation of the for creating and OpenGL Context and a window, such as 'WGL' [WGL] on Windows, 'GLX' [GLX] on XWindows systems and Cocoa on OSX. Fortunately, there are many cross-platform software libraries that unify those platform-specific APIs, one of the most prominent being the OpenGL Utility Toolkit, or 'GLUT' [Kilgard] for short. GLUT was originally written by Mark J. Kilgard to accompany the first OpenGL Programming Guide in 1994, the so called red book, and even though it is no longer supported, it has been in use ever since. Because of licensing issues GLUT is no longer maintained, but a re-implementation that is more-or-less actively maintained called 'FreeGlut' [Olszta] is also available. GLUT is standard with OSX/XCode, and can be easily downloaded and installed for Windows and Linux. An extensive list of the various toolkits for different platforms can be found on the OpenGL website [toolkits].

```
#include <glut.h>
void displayFunc( void ){
  glClearColor(0.0, 0.0, 0.0, 1.0);
  glClear(GL_COLOR_BUFFER_BIT);
  glViewport(0, 0, 400, 400);

  glColor4f(1.0, 0.0, 0.0, 1.0);
  GLfloat vertices[8] = {-0.1,-0.1,0.1,-0.1,0.1,0.1,-0.1,0.1};
  glEnableClientState(GL_VERTEX_ARRAY);
  glVertexPointer(2, GL_FLOAT, 0, vertices);
  glDrawArrays(GL_QUADS, 0, 4);

  glutSwapBuffers();
}

void main( int argc, char ** argv ){
  glutInit( & argc, argv );
  glutInitWindowSize( 400, 400 );
  glutInitDisplayMode( GLUT_DOUBLE | GLUT_RGBA );
  glutCreateWindow( "GLut" );
  glutDisplayFunc( displayFunc );
  glutMainLoop();
}
```

Listing 1.1. A minimal OpenGL example using GLUT.

1.2.1 Hello World With GLUT

In the example in Listing 1.1, the main routine initializes the GLUT library and then tells the windowing system to create an OpenGL window that is double-buffered and has an RGBA pixel format. Then it registers a display callback that is called the first time the window is displayed on the screen, and when a previously obscured part of the window is shown again. Finally, it calls glutMainloop which is a bit typical, because GLUT never returns from this function. GLUT's work is now done, and it gives control to the window it has created. This example can be built and run, provided that the compiler or development IDE knows the include- and linker paths in order to find glut.h and link against the correct library (Glut.a on linux, GLUT.framework on OSX, glut32.lib on Windows and so on). The rather amazing thing is that this code runs exactly as it is printed here on all the desktop systems listed in 1.1, and has done so since it's inception for the systems that were available at that time.

1.2.2 Hello World With Qt

The same example can be written for Qt [QT], which can hardly be categorized as a 'Utility library' as it is a complete Graphical User Interface framework including a GUI-designer, audio facilities, etc. but in the context of drawing OpenGL content we can regard the QtOpenGL component of the Qt suite as similar to GLUT in that it facilitates the creation of an OpenGL context and window for us. Qt originated from Quasar Technologies, later TrollTech, in 1992 and was bought by Nokia in 2008. The library is now available under the LGPL Open Source license, and also under a commercial license from Nokia.

Compiling and linking the Qt example in Listing 1.2 is not quite as simple as building the GLUT example, because it requires the installation of the entire Qt suite, but the developers have made it as easy as possible by providing a configuration utility that lets us select which options we want to include and then generates the build-scripts for us.

1.2.3 Hello World With EGL

Finally, the same example written in 'EGL' [EGL] can be seen in Listing 1.3. EGL interfaces OpenGL|ES with the native windows-API on a wide variety of platforms, including mobile and desktop. However, using EGL is a bit more involved because unlike Qt and GLUT, EGL does not provide a mechanism for creating a window in a platform-independent way. EGL allows us to create the rendering context and a drawing surface, and connect it to an existing window or display, but no more. We can create a native display ourselves, or we can use the EGL_DEFAULT_DISPLAY flag

```
#include <QtCore/QtCore>
#include <QtGui/QtGui>
#include <QtOpenGL/QtOpenGL>

class MyView : public QGLWidget {
  Q_OBJECT
  public:
    MyView( QWidget * parent = 0 ) :
    QGLWidget( QGLFormat( QGL::DoubleBuffer | QGL::Rgba ), parent ){
      resize( 400, 400 );
}
~MyView(){}

protected:
  void paintGL( QGLPainter *painter ){
    makeCurrent();

    glClearColor(0.0, 0.0, 0.0, 1.0);
    glClear(GL_COLOR_BUFFER_BIT);
    glViewport(0, 0, 400, 400);

    glColor4f(1.0, 0.0, 0.0, 1.0);
    GLfloat vertices[8] = {-0.1,-0.1,0.1,-0.1,0.1,0.1,-0.1,0.1};
    glEnableClientState(GL_VERTEX_ARRAY);
    glVertexPointer(2, GL_FLOAT, 0, vertices);
    glDrawArrays(GL_QUADS, 0, 4);
  }
};

int main( int argc, char **argv ){
  QApplication app( argc, argv );
  MyView view;
  return app.exec();
}
```

Listing 1.2. A minimal OpenGL example using Qt.

to obtain the default display for the current system. Either way, we need to create a native window on that display.

Since we are now building for OpenGL|ES, the convenient glOrtho, glMatrixMode, etc. functions are not in the API, and we need to re-implement them in our own library.

1.3 OpenGL Agnosticism

In order to transparently differentiate the request from the programmer to draw a red square to the different GL APIs we can abstract the request completely away from any OpenGL specifics but would like to offer the ease and flow of the immediate-mode fixed-function API. We also want to stay

```
void main( void ){
  EGLint attribList [] = { EGL_BUFFER_SIZE, 32,
               EGL_DEPTH_SIZE, 16,
               EGL_NONE };

  // even though we may obtain the EGL_DEFAULT_DISPLAY, we need
  // to create a handle to a window in which we create the drawing
  // surface, a HDC on windows, a Display on X11, etc.
  EGLNativeDisplayType nativeDisplay = EGL_DEFAULT_DISPLAY;
  EGLNativeWindowType nativeWindow = platformSpecificCreateWindow();
  EGLDisplay iEglDisplay = eglGetDisplay( nativeDisplay );
  eglInitialize( iEglDisplay, 0, 0 );

  EGLConfig iEglConfig;
  EGLint    numConfigs;
  eglChooseConfig(iEglDisplay,
      attribList,
      & iEglConfig,
      1,
      &numConfigs );
  EGLContext iEglContext = eglCreateContext( iEglDisplay,
          iEglConfig,
          EGL_NO_CONTEXT,
          0 );
  EGLSurface iEglSurface = eglCreateWindowSurface(  iEglDisplay,
            iEglConfig,
            & nativeWindow,
            0 );

  // For brevity, we omit a display function similar to
  // the one in the GLUT and Qt examples
}
```

Listing 1.3. A minimal OpenGL example using EGL.

close to the OpenGL naming conventions so that when we think about the objects that we use, we hear the same names as those that are used in the OpenGL registry. Thus, we want a Vertexbuffer class that is completely transparent to the underlying implementation.

The code in Listing 1.4 can be used for all the different dialects of the GL but internally this seemingly simple piece of code fragments into at least three different code-paths.

1. using a VBO (available in all versions).
2. using a VBO with a VAO (from OpenGL3.0).
3. use programs (available in OpenGL2.0 and ES|2.0)

Implementation of the simple example becomes far from trivial. To accommodate for the various versions of OpenGL within the same code-base, and to allow different implementations of the codebase for different

```
Vertexbuffer quad;
quad.color( 1.0, 0.0, 0.0 );
quad.begin( GL_QUADS );
quad.vertex( -10.0, -10.0 );
quad.vertex(  10.0, -10.0 );
quad.vertex(  10.0,  10.0 );
quad.vertex( -10.0,  10.0 );
quad.end();
```

Listing 1.4. Using a RenderTools::Vertexbuffer to emulate the immediate-mode API.

platforms, we made extensive use of 'selective compilation' by defining compiler flags that specify the platform and OpenGL version that we compile for. A typical example of such conditional compilation is the RenderTools::ViewController class, that encapsulates the OpenGL versions 1.1 through 4.x, OpenGLES|1.x and 2.x, the different APIs Qt, GLUT, EGL, Cocoa, EAGL, and even the different languages C++ and Objective-C. To accommodate for communication between the different APIs and languages that are used simultaneously at runtime on different platforms we implemented the ViewController as a global static singleton that can be accessed from anywhere in the system. This is the way that events from the Objective-C based iOS are passed along to the C++ hierarchy of RenderTools. When a ViewController is instantiated, it starts life as a platform specific class such as the IOSViewController on iOS, but is exposed to the developer simply as a 'ViewController' class by means of conditional compilation. If the RenderTools library is compiled for iOS, RT_IOS is defined and the ViewController class will be a typedef of IOSViewController, for GLUT on windows, OSX or Linux, RT_GLUT will be defined and ViewController will be a typedef of GLUTViewController, etc. The different implementations of the ViewController class are wrapped in #ifdef/#endif blocks so as to include or exclude the code from the compilation. Combining these insights resulted in the 'HelloWorld' example that can be found in the RenderTools/examples directory and that can be compiled and run on all the platforms and OpenGL versions supported by RenderTools, without changing one single letter of code, and in fact with just one single configuration action as we shall see in the next section.

All of this together leads to a massive number of possible configuration states. We have the various libraries that we need to include, conditional compilation flags, different OpenGL libraries, possibly third-party libraries depending on the platform, and finally different IDE's on different platforms for which we need to create and maintain project files in order to

```
#include <RenderTools.h>
class HelloWorldView : public RenderTools::RendergroupGLView {
  public:
  static PropertyPtr create( const RenderTools::XMLNodePtr & xml ){
    boost::shared_ptr< HelloWorldView > p( new HelloWorldView() );
    return( boost::dynamic_pointer_cast< RenderTools::AbstractProperty↩
        , HelloWorldView >( p ) );
  }
  virtual const std::string getTypeName( bool ofComponent ) const{ ↩
      return( "HelloWorldView" ); }
  virtual void onInitialize( void ){
    m_buffer = RenderTools::Vertexbuffer::create()->getSharedPtr< ↩
        Vertexbuffer >();
    m_buffer->begin( GL_TRIANGLES );
    m_buffer->color( Vec3( 1.0, 0.0, 0.0 ) );
    m_buffer->vertex( Vec2( -10.0, -10.0 ) );
    m_buffer->vertex( Vec2(  10.0, -10.0 ) );
    m_buffer->vertex( Vec2(  10.0,  10.0 ) );
    m_buffer->vertex( Vec2(  10.0,  10.0 ) );
    m_buffer->vertex( Vec2( -10.0,  10.0 ) );
    m_buffer->vertex( Vec2( -10.0, -10.0 ) );
    m_buffer->end();
  }
  virtual void onRender( const RenderTools::ComponentFilterPtr & ↩
      components ){
      m_buffer->render( GEOMETRIES );
  }
  RenderTools::VertexbufferPtr m_buffer;
};

int main( int argc, char ** argv ){
  RenderTools::initialize( argc, argv );
  RenderTools::Factory::registerContainerType( "HelloWorldView", ↩
      HelloWorldView(), HelloWorldView::create );
  RenderTools::run( "<app type=\"Application\" ><viewcontroller type↩
      =\"HelloWorldView\" /></app>" );
}
```

Listing 1.5. The platform-independent and OpenGL-version agnostic minimal example.

build all the various combinations.

1.4 Configuration Spaces

Now that we have defined an abstraction layer over the different OpenGL versions a next logical step is to further investigate platform-independence. In order to build the ‘agnostic’ OpenGL example we have to introduce different platforms and a coupling between OpenGL versions and platform specifics. To start administrating this increasing complexity, we introduce the concept of a *configuration space*. A configuration space is an exhaustive enumeration of all the possible configuration flags, where a configuration flag may involve the current platform, OpenGL version, and/or external library, either related to OpenGL like GLUT or EGL or independent of the rendering like Boost [Boost] or Bullet physics [Couwmans]. We create names for these configuration flags within the namespace of RenderTools by prefixing the flag with RT_, such as RT_APPLE, RT_WIN32, RT_GLUT, RT_IOS, RT_ES1 etc. which are treated as standard C preprocessor defines. Some RT_[VALUE] definitions depend on context and are implied by the platform the build is performed on (RT_APPLE, RT_WIN32), some are dictated by the OpenGL version that is targeted (RT_ES1 or RT_ES2) and some are required for including the third-party external libraries (RT_GLUT, RT_BULLET). Any one combination of flags out of the entire configuration space is called a *configuration state*.

Examples of configuration states are:

- <RT_WIN32, RT_GLUT, RT_DEBUG, RT_BULLET>A Windows build, using GLUT for the windowing interface, in debug mode, using Bullet as an external library.
- <RT_APPLE, RT_GLUT>A Mac OS X release build, using GLUT for the windowing interface.
- <RT_WIN32, RT_EGL, RT_ES2>A Windows release build, using EGL as the interface between OpenGL|ES2.0 and Windows.
- <RT_APPLE, RT_IOS, RT_ES1, RT_DEBUG>An iOS build for iPhone and iPad, using OpenGL|ES1 fixed function pipeline, to support earlier devices, in debug mode.

Not all configuration states are valid. We can not simultaneously build for RT_ES1 and RT_ES2 and some third-party libraries are mutually exclusive (e.g. RT_BULLET and RT_BOX2D).

1.5 Meta-builds and CMake

One of the most time consuming aspects of platform independent programming is the cumbersome task of defining all the individual settings for the different IDE's: Visual Studio on Windows, Xcode on Apple or makefiles on Unix-based systems. Recently, several so called *meta-build*, or *build automation* systems have been gaining popularity to aid in this task. Examples of meta-build systems are 'premake' [premake] or 'waf' [waf].

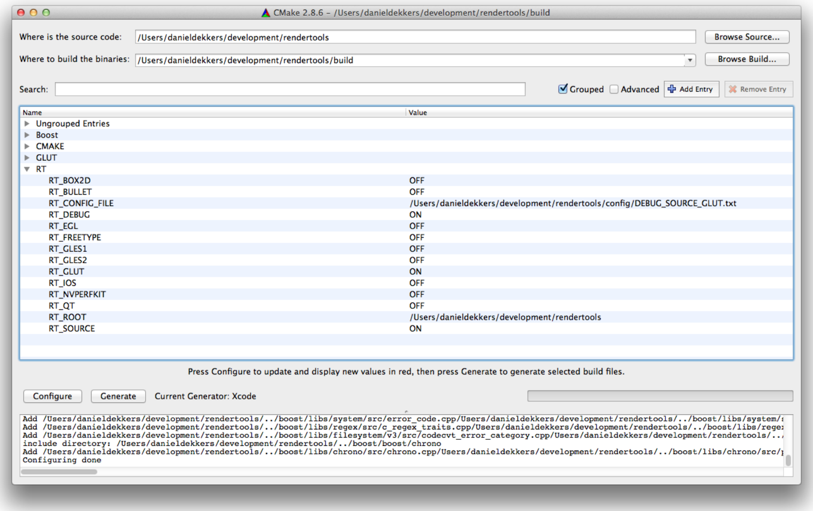
The sheer quantity of settings in IDE's can be overwhelming. The meta-build system creates sensible defaults for all of them and if we want to adjust, we adjust locally via the configuration files. In this way, the exceptions are clearly visible in isolation, instead of hidden in a long enumeration of settings in the IDE.

An advantage of a meta-build system is that it gives the opportunity to migrate back and forth between different versions of an IDE. In almost every IDE it is a very painful process to go back to a previous version, when all the project files have been converted to a newer version. Another advantage of a meta-build system is the relative ease in which projects can be shared between different developers. Every developer typically has slightly different paths to their sources or has adjusted a few settings in the IDE to achieve a local successful build. This makes exporting project files directly to other developers undesirable. With meta-build systems, developers generate fresh project files themselves, out of the source tree, after having adjusted a few absolute paths clearly stated in configuration files in the build system. A rather unexpected advantage we found while developing, the latest version of Xcode, version 4.0.2 on Mac OS X Snow Leopard, proved to be quite unstable and we were a lot more productive developing in Visual Studio, even though the final target would be an iOS application. Meta-builds let the developer choose his favorite development IDE.

One of the more popular and well established tools is CMake [CMake], a free, platform-independent, open-source build system. CMake works with human readable configuration files, always named “CMakeLists.txt”, that contain CMake scripts and exist in directories of the source tree. These CMake configuration files link to each other via the CMake ADD_SUBDIRECTORY() command. A tree traversal is performed starting from a top-level CMake configuration file, passing through the sources, creating project setups for libraries and executables as it goes. After this so called *configure* process, CMake *generates* IDE project files (Visual Studio, Xcode) or generate makefiles on Unix based systems. In daily practice, we typically lose our fixed, static platform dependent project files and generate them dynamically every time a change is made in the build configuration. The CMake structure and especially the syntax takes some time getting used to, but the advantage is that we only have to learn a single language. Traditional makefiles are not much easier to read and only give platform/compiler specific results.

1.6 CMake and the Configuration Space

In a top-level CMake configuration file we can fit our concept of the configuration state. The RT_[VALUE] elements of the configuration state can be mapped directly onto so called *options* in CMake. These options are communicated to the developer and can be adjusted in the CMake GUI (Figure 1.1).



Where is the source code: /Users/danieldekkers/development/rendertools

Where to build the binaries: /Users/danieldekkers/development/rendertools/build

Search:

[✓ Grouped] Advanced

Name	Value
▶ Ungrouped Entries	
▶ Boost	
▶ CMAKE	
▶ GLUT	
▼ RT	
RT_BOX2D	OFF
RT_BULLET	OFF
RT_CONFIG_FILE	/Users/danieldekkers/development/rendertools/config/DEBUG_SOURCE_GLUT.txt
RT_DEBUG	ON
RT_EGL	OFF
RT_FREETYPE	OFF
RT_GLES1	OFF
RT_GLES2	OFF
RT_GLUT	ON
RT_IOS	OFF
RT_NVPERFKIT	OFF
RT_QT	OFF
RT_ROOT	/Users/danieldekkers/development/rendertools
RT_SOURCE	ON

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

Current Generator: Xcode

```
Add /Users/danieldekkers/development/rendertools/../boost/libs/system/src/error_code.cpp/Users/danieldekkers/development/rendertools/../boost/libs/system/
Add /Users/danieldekkers/development/rendertools/../boost/libs/regex/src/c_regex_traits.cpp/Users/danieldekkers/development/rendertools/../boost/libs/rege
Add /Users/danieldekkers/development/rendertools/../boost/libs/filesystem/v3/src/codecvt_error_category.cpp/Users/danieldekkers/development/rendertools/..
include directory: /Users/danieldekkers/development/rendertools/../boost/boost/chrono
Add /Users/danieldekkers/development/rendertools/../boost/libs/chrono/src/chrono.cpp/Users/danieldekkers/development/rendertools/../boost/libs/chrono/src/
Configuring done
```

Figure 1.1. The CMake (2.8.6) GUI.

We define the configuration state in a CMake includable file "configurationspace.cmake". In this file, we not only set the various RT_[VALUE] options, but also, based on these settings, set include directories and add definitions. A boolean RT_[VALUE] that is either "ON" or "OFF" in CMake, will be passed on to the compiler as a preprocessor definition with the CMake commands:

```
IF(RT_[VALUE]) ADD_DEFINITION(-DRT_[VALUE])
```

Furthermore, we have to locate third-party libraries such as GLUT, Bullet, Boost, etc., that are needed for this particular configuration state. These paths are developer dependent, so they can not be known in advance. CMake provides a find_package() mechanism to find the packages after a root path is set. It also searches in platform specific standard locations, in case the libraries are installed system-wide. If the package finding is omitted in this phase, the individual CMake configuration files of RenderTools will invoke find_package() calls themselves for the libraries it needs. As a whole, configurationspace.cmake creates a context from which the library as well as the application(s) can be built.

A typical main development source tree with CMake configuration files is shown in Figure 1.6. In this directory structure, /rendertools is the

```
+ development
  (CmakeLists.txt)
  + ARenderToolsApp
    CMakeLists.txt
    + src
    + rsrc
    + config
  + rendertools
    CMakeLists.txt
    + src
    + examples
      CMakeLists.txt
      + HelloWorld
        CMakeLists.txt
        + src
        + rsrc
        + config
      + CameraTest
        CMakeLists.txt
        + src
        + rsrc
        + config
      + ...
    + config
      configurationspace.cmake
  + bullet (external)
    CMakeLists.txt
    + src
  + boost (external)
  + glut (external)
  + ...
```

Listing 1.6. A typical main development source tree with CMake configuration files.

directory that contains the "suite" containing the library and the examples. It contains a CMakeLists.txt that is a logical start of a build, see the "Where is the source code" entry in the screenshot of the CMake GUI, Figure 1.1. With the project that is generated from this CMakeLists.txt we can build the library based on the RT_[VALUE] settings we choose and build the examples that are depending on this library.

The general structure of a top-level CMakeLists.txt in a RenderTools context looks like:

1. Include configurationspace.cmake to define the configuration state <RT_APPLE, RT_DEBUG, RT_IOS,...>, set the paths to the (external) third-party libraries, add include directories and pass on definitions that are needed for this particular configuration state.

2. Recurse into the actual source code directory of the RenderTools library by invoking ADD_SUBDIRECTORY(rendertools/src). If we

don't want to build RenderTools from source we can omit this step and link to a binary pre-built RenderTools from the applications directly.

3. Recurse source code directory of a RenderTools dependent application, or an intermediate directory representing a set of applications as is the case with the examples.

The CMakeLists.txt of the RenderTools library itself, located in rendertools/src, has a simple structure:

1. Create a new project for the lib

   ```
   PROJECT(RenderTools)
   ```

2. Gather the RenderTools sources

   ```
   FILE(RT_SOURCES ... )
   ```

3. Combine with sources from third-party libraries, depending on the ones defined for inclusion by the RT_[VALUE] options, if RT_SOURCE is defined. Or, if RT_SOURCE is not defined, link third-party libraries directly.

4. Create a library with these sources

   ```
   ADD_LIBRARY(RenderTools ${RT_SOURCES})
   ```

One could imagine RenderTools being distributed as a set of prebuilt binaries, which would actually be a large collection for all the different configuration states on all the different platforms. Instead, we chose to let developers build RenderTools from source. We feel we can do so because we supply the sources, assist in the build process via the CMake configuration files, and provide sensible defaults for common configurations.

The CMakeLists.txt of an intermediate directory that contains various applications simply recurses into these source code directories. As an example, the CMakeLists in /Examples looks like:

1. Create a new project for this application suite

   ```
   PROJECT(Examples)
   ```

2. Recurse into lower level source directories:

```
ADD_SUBDIRECTORY(HelloWorld)
ADD_SUBDIRECTORY(CameraTest)
...
```

Finally, the CMakeLists.txt in the directories of individual applications have the following structure:

1. Create a new project for the application

   ```
   PROJECT(HelloWorld)
   ```

2. Gather application specific sources and resources

   ```
   FILE(APP_SOURCES ...)
   FILE(APP_RESOURCES ...)
   ```

3. Create an executable with these sources and resources

   ```
   ADD_EXECUTABLE(HelloWorld
     ${APP_SOURCES} ${APP_RESOURCES})
   ```

4. Link in the RenderTools library

   ```
   TARGET_LINK_LIBRARY(HelloWorld RenderTools)
   ```

5. Set the dependency

   ```
   ADD_DEPENDENCY(HelloWorld RenderTools)
   ```

For applications residing outside the /rendertools directory, e.g. ARenderToolsApp, we can write a CMakeLists.txt for the higher level in /development that includes ARenderToolsApp and the RenderTools library. Such a top-level CMakeLists.txt file looks like:

```
PROJECT(MyDailyWork)
INCLUDE(rendertools/config/configurationspace.cmake)
ADD_SUBDIRECTORY(ARenderToolsApp)
# Recurse directly into the library, avoiding double
inclusion of configurationspace.cmake and the examples.
ADD_SUBDIRECTORY(rendertools/src)
```

Note that this is a volatile file and changes regularly, depending on the projects you are working on at that particular moment.

1.7 CMake and Platform Specifics

The CMake structure mentioned above is the general structure that most CMake-based builds follow. There are of course a lot of platform specific peculiarities that have to be dealt with. Some of the non-trivial ones we encountered are listed below.

1.7.1 Windows

Windows is fairly straightforward. GLUT or EGL handles the windowing interface. OpenGL as binary library is available with the operating system or via a dynamic link library (dll) provided by the hardware manufacturer of the video card. The OpenGL header and library file are shipped with Visual Studio.

- Resources. On Windows we simply copy all the resources the application needs to the build directory, avoiding a more involved search mechanism. Unlike Apple, Windows doesn't use application bundles so some work on placing resources in the final distribution will have to be done in the installer. CMake has a post build command mechanism in which tasks can be specified that have to be performed after the build. The CMake script fragment in Listing **??** copies the resources to the directory where the executable resides.

```
FOREACH(NAME ${APP_RESOURCES})
  GET_FILENAME_COMPONENT(NAMEWITHOUTPATH ${NAME} NAME)
  ADD_CUSTOM_COMMAND(
    TARGET ${APP_NAME}
      POST_BUILD
      COMMAND ${CMAKE_COMMAND} -E copy
              ${NAME}
              ${PROJECT_BINARY_DIR}/
                ${CMAKE_CFG_INTDIR}/
                ${NAMEWITHOUTPATH})
ENDFOREACH()
```

  The CMake variable PROJECT_BINARY_DIR is the build directory of the project, CMAKE_CFG_INTDIR is the current configuration, e.g. Debug, Release, etc. so together they form the actual path to the executable. The ${CMAKE_COMMAND} -E copy is CMakes platform independent copy command.

1.7.2 Mac OS X

The procedure for Mac OS X is similar to Windows. Again, GLUT handles the windowing interface. We have no OpenGL|ES build for Mac OS X at the moment of writing.

- Resources. Our Mac OS X application bundles have the standard hierarchy where the application itself is represented by a bundle (e.g. HelloWorld.app) containing a /Contents directory, that in turn contains a /MacOS directory with the actual executable, e.g. HelloWorld, and a /Resources directory containing the resources. Using a similar CMake post-build construct as we did in Windows, lead to conflicts with the copying that Xcode performs internally. We chose to just let Xcode do its job. In CMake we present sources and resources to ADD_EXECUTABLE():

  ```
  ADD_EXECUTABLE(${APP_NAME} MACOSX_BUNDLE
    ${APP_SOURCES}
    ${APP_RESOURCES})
  ```

  We make sure the resources are labeled as resources, so Xcode will treat them correctly during the build. That is, make them visible in the /Resources folder in the Xcode IDE and copy them to the correct location in the application bundle:

  ```
  SET_TARGET_PROPERTIES(${APP_NAME} PROPERTIES
    RESOURCE "${APP_RESOURCES}")
  ```

- Information property list files. Mac OS X application bundles need an information property list file that enumerates various aspects of your application in the application bundle, which is located directly in the root of the bundle. CMake lets you identify a template file with wildcards, which we name Info.plist.in.

  ```
  SET(APP_PLIST_FILE
    ${APP_ROOT}/config/apple/osx/Info.plist.in)
  SET_TARGET_PROPERTIES(${APP_NAME} PROPERTIES
    MACOSX_BUNDLE_INFO_PLIST ${APP_PLIST_FILE})
  ```

  CMake will read the Info.plist.in, substitute the wildcards and generate a Info.plist in a private section of its build directory, e.g. path/to/build/CMakeFiles/HelloWorld.dir/Info.plist. This file is linked and added to the Xcode project files automatically, after which Xcode will copy it to the root of the application bundle as a pre-build step.

- Objective-C(++). In order to use C++ code in Objective-C, we need to compile all sources as Objective-C++ which we can do with:

  ```
  SET(CMAKE_CXX_FLAGS "-x objective-c++")
  ```

  We need to add the .mm files to the sources in order for them to be built, so we include those with:

  ```
  FILE(GLOB APP_SOURCES ${APP_ROOT}/src/[^.]*.[mmcpph]*)
  ```

1.7.3 iOS

iOS is a lot more involved. Only OpenGL|ES is supported on the devices. OpenGL|ES 2.0 is supported only on newer models (iPhone 3GS and up, iPad 2). OpenGL|ES 1.1 is supported on all models. Furthermore, a distinction is made between applications that run on the simulator on the Intel architecture or on the device itself with the ARM architecture. The signing and provisioning of applications needs special attention and resource management is a bit more involved than on Mac OS X.

- Configurating targets. CMake presents a method of cross-compiling when the build platform is different from the target platform, which was our first approach to creating iOS applications, resulting in different so called toolchain files for device and simulator. This turned out to be an unnecessary in-between step. The compiler can be the default compiler (Apple LLVM compiler 3.0 for Xcode 4.2) also used for Mac OS X builds. The base SDK is selected via the CMAKE_OSX_SYSROOT variable. The CMake script fragment for iOS 5.0:

  ```
  SET(IOS_BASE_SDK_VER "5.0"
    CACHE PATH "iOS Base SDK version")
  SET(IOS_DEVROOT
    "/Developer/Platforms/iPhoneOS.platform/Developer")
  SET(IOS_SDKROOT "${IOS_DEVROOT}/SDKs/
    iPhoneOS${IOS_BASE_SDK_VER}.sdk")
  SET(CMAKE_OSX_SYSROOT "${SDKROOT}")
  ```

  The architecture has to be set:

  ```
  SET (CMAKE_OSX_ARCHITECTURES
    "$(ARCHS_STANDARD_32_BIT)")
  ```

which will result in the standard armv7 setting. We set the target to create universal applications for both iPad and iPhone. The representation “1,2” will be translated correctly to “iPhone/iPad” in Xcode:

```
SET_TARGET_PROPERTIES(${APP_NAME} PROPERTIES
  XCODE_ATTRIBUTE_TARGETED_DEVICE_FAMILY "1,2")
SET_TARGET_PROPERTIES(${APP_NAME} PROPERTIES
  XCODE_ATTRIBUTE_DEVICES "Universal")
```

We can set the minimal iOS deployment version, iOS 4.3 in this case:

```
SET_TARGET_PROPERTIES(${APP_NAME} PROPERTIES
  XCODE_ATTRIBUTE_IPHONEOS_DEPLOYMENT_TARGET 4.3)
```

- Frameworks. The different frameworks needed for linking can be added via linker flags. They will not be clearly visible in the Xcode IDE, but the applications link correctly:

```
# Enumerate frameworks to be linked to on iOS...
SET(IOS_FRAMEWORKS ${IOS_FRAMEWORKS} OpenGLES)
SET(IOS_FRAMEWORKS ${IOS_FRAMEWORKS} UIKit)
SET(IOS_FRAMEWORKS ${IOS_FRAMEWORKS} Foundation)
SET(IOS_FRAMEWORKS ${IOS_FRAMEWORKS} CoreGraphics)
SET(IOS_FRAMEWORKS ${IOS_FRAMEWORKS} QuartzCore)
...
FOREACH(NAME ${IOS_FRAMEWORKS})
  SET(CMAKE_EXE_LINKER_FLAGS
    "${CMAKE_EXE_LINKER_FLAGS} -framework ${NAME}")
ENDFOREACH()
```

- Effective platforms. New in the latest version of CMake, 2.8.6, is the concept of effective platforms. Setting this parameter makes sure that if we switch between device or simulator schemes in the Xcode IDE, the correct build path to the corresponding library is automatically selected by Xcode. This will be either path/to/build/[config]-iphoneos or path/to/build/[config]-iphonesimulator, where config represents your current configuration, Debug, Release, etc.

```
SET(CMAKE_XCODE_EFFECTIVE_PLATFORMS
  -iphoneos;-iphonesimulator)
```

- Information Property List Files. As in Mac OS X builds, an Info.plist property list file is needed in the bundle. In CMake, they are treated similar to the Mac OS X builds, we only provide a different template because iOS has some additional properties like orientations of the device and minimal device requirements, e.g. gyroscope, gps.

  ```
  SET(PLIST_FILE
    ${APP_ROOT}/config/apple/ios/Info.plist.in)
  ```

- Interface Builder. Xib files are Interface Builder User Interface files. As a pre-build step Xcode compiles them into binary nib files and adds them to the bundle, but only if they are properly identified as resource. The CMake script fragment:

  ```
  FILE(GLOB XIB_FILES
    ${APP_ROOT}/config/apple/ios/*.xib) # Gather xib files
  SET_TARGET_PROPERTIES(${APP_NAME} PROPERTIES
    RESOURCE "${XIB_FILES}")
  ```

- Provisioning and code signing. Provisioning and code signing is one of the more error prone new aspects of iOS development. After subscribing to the Apple Developer Program a developer will have to spend quite some time in the "iOS Provisioning Portal" on the Apple Developer website. First, developer and distribution certificates have to be generated that can be added to the personal keychain. For each application, we generate an application identifier called the AppID and generate developer, AdHoc and AppStore distribution provisioning profiles called *.mobile-provisioning files that need to be linked with our bundle. Without those we can only run applications in the simulator. With a developer provisioning profile, we can run and debug our application on the device that is tethered to our development machine directly from Xcode. With the AdHoc distribution, we can create a so called archive that can be sent around and installed locally on a limited set of trusted devices via iTunes. We need to know and enumerate the unique UID-keys of these devices in advance. The AppStore distribution allows us to distribute our application via the AppStore after it is approved by Apple. The App ID in reversed domain notation is set through the CMake MACOS_BUNDLE_GUI_IDENTIFIER variable. This entry will also be substituted in the Info.plist file via a wildcard as value for the CFBundleIdentifier key:

```
SET(IOS_APP_IDENTIFIER nl.cthrough.helloworld)
# this has to match to your App ID (case sensitive)
SET(MACOSX_BUNDLE_GUI_IDENTIFIER ${IOS_APP_IDENTIFIER})
```

The provision profile is a separate setting:

```
SET(IOS_CODESIGN_ENTITLEMENTS
  ${APP_ROOT}/config/apple/ios/
  entitlements/EntitlementsDebug.plist)
  # replace with EntitlementsDistributionAdHoc.plist or
  # EntitlementsDistributionAppStore.plist
SET_TARGET_PROPERTIES(${APP_NAME} PROPERTIES
  XCODE_ATTRIBUTE_CODE_SIGN_ENTITLEMENTS
  ${IOS_CODESIGN_ENTITLEMENTS})
```

- Archiving. Archiving consists of creating an archive for distribution, either AdHoc or AppStore. To create a successful archive we have to make sure that in Xcode the skip install property is not set for the application, and set for static libraries which is the CMake default. Furthermore we make sure that the path to an installation directory is not empty:

```
SET_TARGET_PROPERTIES(${APP_NAME} PROPERTIES
  XCODE_ATTRIBUTE_SKIP_INSTALL NO)
SET_TARGET_PROPERTIES(${APP_NAME} PROPERTIES
  XCODE_ATTRIBUTE_INSTALL_PATH "/Applications")
```

We also have to make sure that in the code signing field a valid iPhone Distribution as opposed to an iPhone Developer profile is set. Unfortunately, with the latest version of CMake, it is not yet possible to set different values for different configurations in Xcode, but this feature is on the road map for the next version (2.8.7). We hope to be able to do the following:

```
SET( IOS_CODE_SIGN_IDENTITY_DEVELOPER
  "iPhone Developer"
  CACHE STRING "code signing identity" )
  # For developing
SET( IOS_CODE_SIGN_IDENTITY_DISTRIBUTION
  "iPhone Distribution"
  CACHE STRING "code signing identity" )
  # AdHoc or AppStore distribution
```

```
SET_TARGET_PROPERTIES( ${APP_NAME} PROPERTIES
  XCODE_ATTRIBUTE_CODE_SIGN_IDENTITY[variant=Debug]
  ${IOS_CODE_SIGN_IDENTITY_DEVELOPER} )
SET_TARGET_PROPERTIES( ${APP_NAME} PROPERTIES
  XCODE_ATTRIBUTE_CODE_SIGN_IDENTITY[variant=Release]
  ${IOS_CODE_SIGN_IDENTITY_DISTRIBUTION} )
```

Instead of manually changing the value of the code sign identity, like we have to do now:

```
SET( IOS_CODE_SIGN_IDENTITY "iPhone Developer"
  CACHE STRING "code signing identity" )
  # Change to iPhone Distribution for archiving
SET_TARGET_PROPERTIES( ${APP_NAME} PROPERTIES
  XCODE_ATTRIBUTE_CODE_SIGN_IDENTITY
  ${RT_CODE_SIGN_IDENTITY} )
```

1.8 Conclusion

Fortunately, the last decade has shown an incredible increase in the number of community-driven software projects aiming to help deal with the complexities of cross-platform development (Boost, CMake) and help to avoid overly complex codebases when targeting different OpenGL versions (GLEW, GLUT). Unfortunately, the task of selecting the best set of these projects is a difficult one. There are many alternatives to the selection we made but we believe that we have made the most sensible choice at this time.

We would like to thank contributors from the very active CMake community, especially David Cole and Michael Hertling. And George van Venrooij, for pointing out CMake in the first place.

Bibliography

[Boost] Boost. "Boost C++ Libaries." http://www.boost.org. Portable, peer-reviewed C++ libraries.

[CMake] CMake. "CMake - Cross Platform Make." http://www.cmake.org. A cross-platform, open-source, build system.

[Couwmans] Erwin Couwmans. "Bullet Physics Library." http://bulletphysics.org. A Collision Detection and Rigid Body Dynamics Library.

[EGL] EGL. “EGL - Native Platform Interface.” http://www.khronos.org/egl. A native platform graphics interface for openGL and OpenVG maintained by Khronos.

[GLEW] GLEW. “GLEW - The OpenGL Extension Wrangler Library.” http://glew.sourceforge.net. GLEW, the OpenGL Extension Wrangler by Milan Ikits and Nigel Stuart.

[GLX] GLX. “GLX - Wikipedia, the free encyclopedia.” http://en.wikipedia.org/wiki/GLX. OpenGL Extension to the X Window System.

[implementations] implementations. “OpenGL implementations.” http://www.opengl.org/documentation/implementations. OpenGL implementations.

[Kilgard] Mark Kilgard. “GLUT - The OpenGL Utility Toolkit.” http://www.opengl.org/resources/libraries/glut/. The OpenGL Utility Toolkit.

[Olszta] Pawel W. Olszta. “FreeGLUT - The OpenSourced alternative to GLUT.” http://freeglut.sourceforge.net/. The OpenSourced alternative to GLUT.

[premake] premake. “premake — build script generation.” http://premake.sourceforge.net/. Premake, build script generation.

[QT] QT. “Qt - A cross-platform application and UI framework.” http://qt.nokia.com/products. A cross-platform GUI API by TrollTech/NOKIA.

[RenderTools] RenderTools. “RenderTools.” http://rendertools.dynamica.org. A (lightweight) OpenGL based scenegraph library by J. van der Spek.

[toolkits] toolkits. “OpenGL Toolkits.” http://www.opengl.org/wiki/Related_toolkits_and_APIs#Context.2FWindow/Toolkits. OpenGL Toolkits.

[waf] waf. “waf - The meta build system - Google Project Hosting.” http://code.google.com/p/waf/. Waf - The meta build system - Google Project Hosting.

[WGL] WGL. “WGL (software) - Wikipedia, the free encyclopedia.” http://en.wikipedia.org/wiki/WGL_(software). The windowing system interface to the Microsoft Windows implementation of the OpenGL specification.