Game Engine Programming
Lab 1: An Introduction to CMake

Karsten Pedersen
Department of Creative Technology

August 9, 2018

In this unit we aim to build a game engine which will potentially generate a number of libraries and will likely have a number of dependencies itself on 3rd party libraries. We will also be looking at alternative C++ compilers to the one provided by *Microsoft Visual Studio* (***cl***). For this reason we will be using *CMake* to manage our project for us.

The first thing we will need to do is ensure that it is installed and added to our ***PATH*** environment variable. To do this, open up a command prompt or terminal emulator and enter:

```
> cmake --version
```

If *CMake* is correctly installed, you should see output similar to:

```
cmake version 2.8.12.2
```

Hopefully with that in place we can start creating our initial project. The first step is to create a new directory called ***myengine*** (or anything you want to call your game engine). Try to avoid network drives such as your ***H:*** because it is fairly slow to build software. Once you have created this directory, we will populate it with a general framework of the project. **Figure 1** shows the beginning of a standard C/C++ project layout.

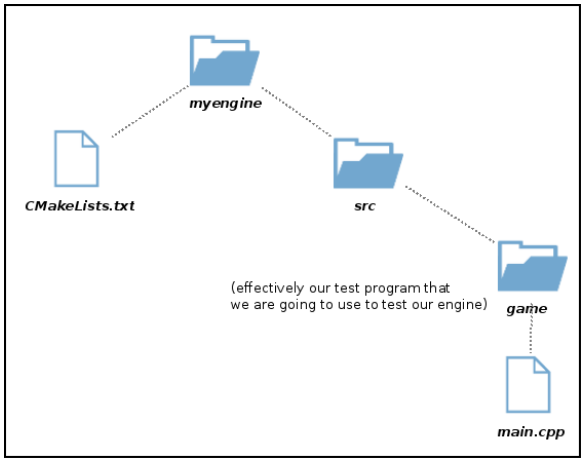

Figure 1: *Diagram showing the intended filesystem structure*

To get started the ***main.cpp*** should be populated with a small *Hello World* program as shown in the listing below.

```
#include <iostream>

int main()
{
  std::cout << "Hello World" << std::endl;

  return 0;
}
```

Next the file called ***CMakeLists.txt*** is what instructs *CMake* how to build your program from the source. It can describe every aspect about building your project from a single file or multiple depending on how complex your build system gets. For example it could be instructed to perform tasks such as running test suites, generating documentation, baking lighting, compiling to multiple platforms, building installers, etc. However, for now lets start with the bare minimum.

```
cmake_minimum_required(VERSION 2.6)
project(MYENGINE)

add_executable(game
  src/game/main.cpp
)
```

The ***cmake_minimum_required*** statement tells *CMake* that it only uses older features of the build tool. This allows for great backwards compatibility with less updated platforms. As we require more modern features, we can bump up the version to unlock them at the expense of supporting old operating systems such as *Solaris 10* or *Microsoft Windows XP*.
The ***project*** statement tells *CMake* what to name the entire project (not just individual libraries or executables that it generates). This is useful when creating scripts later on. We use all capitals here because otherwise variables are named things like ***myengine_VERSION*** which looks a bit horrible.
Finally the ***add_executable*** command instructs *CMake* to create an output executable (**.exe* on *Windows*) with the given name and compiled from the specified source files. We could generate a library (**.lib* for *Microsoft Visual C++*, **.a* for *GNU C/C++ Compiler*) by using ***add_library*** instead. You will encounter this later on.

Once this file is in place, now we build the project. First we create a new directory called ***build*** in the root of the project. This is where *CMake* puts **all** of the generated files. This is an especially nice feature because it doesn't spam your project with clutter. Then you can easily delete the folder to clean the project or simply add it to the ***.gitignore*** file to prevent accidentally committing it into your *Git* repository. We then use *CMake* to generate the project and finally use *CMake* again to build it. The following listing shows this process.

```
> mkdir build     # Create the directory
> cd build        # Go into the newly created directory

> cmake ..        # Generate project in this directory but use the
                  # CMakeLists.txt file from one level up (..)

> cmake --build . # Build from the files in current directory (.)
> Debug\game.exe  # Run the created executable
```

From now on, to trigger rebuilds, simply use the ***cmake –build .*** unless you change machines or delete the ***build*** folder. Remember however to make sure you are in the correct directory. I personally move back into the root of the project and compile from there. For example:

```
> cd ..
> cmake --build build
```

This has the small benefit that you can still use *Git* easily from the root of the project and also any relative paths that a simple naive program may try to open will work correctly.

Finally, the whole point of *CMake* is to use existing tools. If you want to use *Microsoft Visual Studio*, you can do so by simply double clicking on the ****.sln*** file within the created ***build*** directory. You can then pretty much use it as normal. Just remember to use the ***CMakeLists.txt*** file instead to add and remove ****.h*** and ****.cpp*** files.

Note:
Spend this time at the end of the lab setting up a version control repo for your project (i.e *Git*, *Subversion*). Your project will likely grow quite large and source control will help you manage it. Simply check in everything apart from the ***build*** folder. This could be added to your ***.gitignore*** file. With a bit of practice, you will start to see that *Git* and *CMake* work very well together and why they are such common tools to use within the industry.

Once you are happy with using *CMake*. Try adding a new subfolder to the ***src*** directory called ***myengine*** and start adding some initial files in there that you think you might need. Try to compile them into a library using ***add_library*** in your ***CMakeLists.txt***. At this point, your project layout will start to begin looking similar to **Figure 2**.

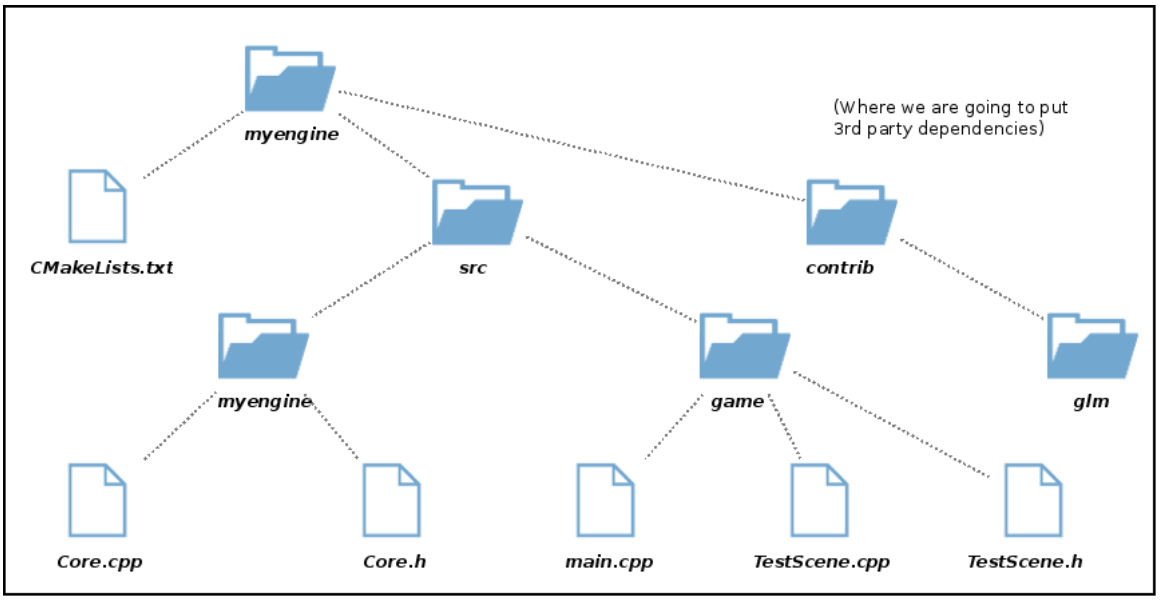

Figure 2: *The structure of a slightly more developed project*