Cmake

An introduction

This document provides a quick overview to **CMake** — a cross-platform alternative to linux Makefiles and make. For help on using **make** and Makefiles without **CMake**, please see **Prof. Newhall's documentation**. This document was written for cmake version 2.6, which is installed on the lab machines via stow (as of Jan 2009).

Why CMake

I personally like CMake because is it seems more intuitive than writing a standard Makefile. Basic and common things are pretty easy and harder things are possible, though most of the stuff I build is pretty basic. To demonstrate CMake, I have included some sample code in the file **cmake.tgz** (4 KB). Save the file and unpack the contents anywhere in your directory. For example:

```
$ tar xzvf cmake.tgz
$ cd cmake
$ ls
CMakeLists.txt  w01-cpp/
```

CMakeLists.txt

CMake is controlled by writing instructions in **CMakeLists.txt** files. Each directory in your project should have a**CMakeLists.txt** file. What is nice about **CMake** is that **CMakeLists.txt** files in a sub-directory inherit properties set in the parent directory, reducing the amount of code duplication. For our sample project, we only have one subdirectory: **w01-cpp**. The **CMakeLists.txt** file for the top-level **cmake** directory is pretty simple but demonstrates a few key features.

```
cmake_minimum_required(VERSION 2.6)
project(CMAKEDEMO)

#There are lots of scripts with cmake
#for finding external libraries.
#see /usr/local/share/cmake-2.6/Modules/Find*.cmake for more examples
find_package(GLUT)
find_package(OpenGL)

set(CMAKE_CXX_FLAGS "-g -Wall")
add_subdirectory(w01-cpp)
```

This demo code includes code that requires external OpenGL libraries for graphics. In a typical Makefile configuration we would likely need to specify where the OpenGL header files are e.g., **-I/usr/local/include**, which libraries to use, e.g., **-lGL -lGLU -lglut**, and the location of the libraries. The location of the header files and libraries likely varies from install to install and from platform to platform. There are usually some standard places to look however, and **CMake** automates this search using **find_package** macros.
Thus**find_package(GLUT)** and **find_package(OpenGL)** finds the location of the header files and libraries and sets up the most common libraries that are linked to in a typical OpenGL project. **CMake** has support for finding lots of packages. See **/usr/local/share/cmake-2.6/Modules/Find*.cmake** for more examples. To use one of these scripts, use **find_package(PKGname)** if the **CMake** script is named **FindPKGname.cmake**.

For more on finding external libraries or creating your own find package macros, see the **KitWare wiki** on the subject.

You can set common flags that you would set in a typical Makefile using **set(VARNAME VALUE)**. In this example I enabled debugging symbols (-g) and all warnings (-Wall). I also gave this CMake project a name using**project(CMAKEDEMO)**. You can name your project whatever you want. We'll use this info in the**CMakeLists.txt** file in the **w01-cpp** directory. Finally we indicate that most of code is really in the **w01-cpp** by telling our top-level **CMakeLists.txt** file to **add_subdirectory(w01-cpp)**

Let's take a look at the **CMakeLists.txt** file in the **w01-cpp** subdirectory:

```
include_directories(${CMAKEDEMO_SOURCE_DIR}/w01-cpp)
link_directories(${CMAKEDEMO_BINARY_DIR}/w01-cpp)
```

```
#the one C file
add_executable(cdemo cdemo.c)
target_link_libraries(cdemo m) #link the math library

#these are all compiled the same way
set(PROGRAMS oglfirst pointers)
set(CORELIBS ${GLUT_LIBRARY} ${OPENGL_LIBRARY} m)

foreach(program ${PROGRAMS})
  add_executable(${program} ${program}.cpp)
  target_link_libraries(${program} ${CORELIBS})
endforeach(program)

#building just a library.
add_library(geometry geometry.cpp)

add_executable(test_geometry test_geometry.cpp)
#linking against a custom library
target_link_libraries(test_geometry ${CORELIBS} geometry)
```

The **include_directories** macro tells CMake where to look for source files. By declaring the project name**CMAKEDEMO** in the top-level **CMakeLists.txt** file, the variables **CMAKEDEMO_SOURCE_DIR** and**CMAKEDEMO_BINARY_DIR** are set depending on the current location of your code and the current location of your build directory (more on this latter location later). Adding a target for a new binary executable is easy. Just add a line **add_executable(cdemo cdemo.c)**. CMake will automatically figure out the compiler based on the file type extension. If additional libraries are needed, you can tell CMake to link against them using**target_link_libraries(cdemo m)**. This tells CMake that the **cdemo** program needs to link against the math library. You can link against more than one library by specifying the libraries in a list. See for example**set(CORELIBS ${GLUT_LIBRARY} ${OPENGL_LIBRARY} m)**. The variables **GLUT_LIBRARY** and**OPENGL_LIBRARY** are set by CMake when we used the **find_package(GLUT)** and **find_package(OpenGL)**. To figure out which variables are set when using find_package, you will probably need to open the appropriate**Find*.cmake** file and read the usually helpful but terse documentation.

Since several programs in this **w01-cpp** folder are compiled the same way and linked against the same libraries, we can process them all with a simple loop shown above using the **foreach** macro in CMake. Finally we show how to create our own library using **add_library(geometry geometry.cpp)** and linking against this library in**target_link_libraries(test_geometry ${CORELIBS} geometry)**.

Building with CMake

Setting up a bunch of **CMakeLists.txt** files will not immediately allow you to build your project. CMake is just a cross platform wrapper around more traditional build systems. In the case of linux, this means **make**. A quick preprocessing step will convert your **CMakeLists.txt** description into a traditional make build system automatically. One nice and highly recommended feature of **CMake** is the ability to do out of source builds. In this way you can make all your **.o** files, various temporary depend files, and even the binary executables without cluttering up your source tree. To use out of source builds, create a build directory in your top-level folder (technically, this can be anywhere, but the top-level project folder seems to be a logical choice). Next, change into your build directory and run **cmake** pointing it to the directory of the top-level **CMakeLists.txt**. For example:

```
cumin[~]$ cd cmake/
cumin[cmake]$ ls
CMakeLists.txt  w01-cpp/
cumin[cmake]$ mkdir build
cumin[cmake]$ ls
CMakeLists.txt  build/  w01-cpp/
cumin[cmake]$ cd build/
cumin[build]$ cmake ..
```

Remember to be in your build directory and point cmake only to the **directory** containing the top-level**CMakeLists.txt** file, not the file itself. If all goes well, **cmake** will process your **CMakeLists.txt** files, find the location of all libraries and include paths and spew a bunch of configuration information including a traditional**Makefile** in your **build** directory. (If you have any familiarity with autotools/autohell, this **cmake** process is similar to **./configure**). You are now ready to build using the traditional **make** system. Run **make** in your build directory to compile and link everything. **CMake** even tosses in some nice colors, progress bars, and suppresses a bunch of **gcc** output. If you want verbose output, you can type **VERBOSE=1 make** (helpful if something goes

wrong).

Because of the way we set up the out of source build and the**link_directories(${CMAKEDEMO_BINARY_DIR}/w01-cpp)**, our freshly compiled binaries are in the **w01-cpp** folder in the **build** directory where we just ran make.

```
cumin[build]$ cd w01-cpp/
cumin[w01-cpp]$ ls
CMakeFiles/  cdemo*              libgeometry.a  pointers*
Makefile     cmake_install.cmake  oglfirst*      test_geometry*
cumin[w01-cpp]$ ./cdemo
Sqrt(2) = 1.4142
This concludes a short C demo
cumin[w01-cpp]$ ./oglfirst
cumin[w01-cpp]$ ./test_geometry
```

Press ESC to quit the OpenGL demos.

Wrapping up

If you modify code in your source directory, including even a **CMakeLists.txt** file and re-run make in the **build**directory, **make** and **cmake** will recompile and rebuild necessary changes. If you are only making changes in a subdirectory, you can simply run **make** in the corresponding subdirectory in the build tree to process updates.

An initial source of confusion with out of source builds is that you basically have two copies of your source tree, one with actual source code, and one with **Makefiles** and binary executables (in the build tree). It is probably best to keep two windows open with one in the build tree for making and running your programs, and one window in the source tree for modifying source files.

One nice thing about out of source builds is that cleaning up object files, makedepend files, binaries, and other miscellaneous build cruft can be done by simply deleting the entire build directory because there is no source. You can also use **make clean** to clean up the actual object and binary files, but when you are planning to tar up your source or distribute your code to the masses, you can simply do

```
cumin[~]$ cd cmake
cumin[cmake]$ ls
CMakeLists.txt  build/  w01-cpp/
cumin[cmake]$ rm -rf build/
cumin[cmake]$ ls
CMakeLists.txt  w01-cpp/
```

to clean up and package your code.

More to explore

This only touches the surface of what CMake can do. Check out the **CMake Wiki** for more info. I hope to update this page if we run into more complex examples, and I'm happy to take feedback or tips from other courses that might be using CMake. For very advanced users, CMake has a companion program **CPack** for automatically packaging binaries and source for Ubuntu/Debian (.deb), Red Hat (.rpm), OSXX11, tgz, CygWin, and Nullsoft systems.