OpenGL Projects Readme (Linux)

Note: this readme only contains specific instructions to build the accompanied OpenGL projects on Ubuntu Linux (Version 15.04 "Vivid Vervet"). If you are using another distribution, the listed steps will very likely have to be performed in a similar fashion.

Step 1: Check your system's OpenGL capabilities

- Install the Mesa utilities to obtain glxgears and glxinfo: On Ubuntu, run `sudo apt-get install mesa-utils` from your terminal.
- Run `glxgears -info` from a terminal. You should be able to see an OpenGL window containing three colored gears.
- The output of glxgears should look similar to this:
```
GL_RENDERER   = GeForce GTX 670/PCIe/SSE2
GL_VERSION    = 4.2.0 NVIDIA 304.125
GL_VENDOR     = NVIDIA Corporation
GL_EXTENSIONS = [...]
```
- Make sure your graphics card is at least able to run OpenGL 4.0. If that is not the case, but you have a reasonably current card, you might need to install the manufacturer's proprietary driver. On Ubuntu systems, this can be done using apt-get:
  - If you're using an Nvidia card, run `sudo apt-get install nvidia-current`
  - If you're using an ATI (i.e. AMD) card, run `sudo apt-get install fglrx`
  - You'll have to reboot your system after the installation completes.
- The Linux driver for Intel's integrated graphics chips (i965) **does not** support OpenGL 4.0 yet (cf. the Mesa support matrix). If this is your only graphics card, you'll have to complete the course using a Windows system.
  - You can use Linux and Windows alongside each other using a dual-boot setup. Configuring such a system lies outside of the scope of this readme.
  - If you have access to another Windows machine, you can create a portable Windows 8 system using a sufficiently fast USB-Disk and WinToUsb.
  - For both cases, you can obtain the necessary install image for Windows 8 from Microsoft DreamSpark.
- If your system contains an Nvidia optimus configuration (i.e. both an Intel chip and an Nvidia card), installing Bumblebee might help. This step will complicate debugging a bit, though:
  - To run a binary on the Nvidia card use the command `optirun ./<binary>` (depending on your system, you might also be able to run `primusrun ./<binary>`). You will then need to attach your debugger using the PID of the spawned process.
  - You can also execute the whole IDE through optirun and from within it debug the application as usual.
  - Finally, you should be able to force your system to use the discrete card in your BIOS or EFI settings.

Step 2: Install the necessary dependencies

- To build the projects, you will need to install the following software on your system:
  - The basic compiler utilities for your system (GCC/GDB/Make etc.)
  - The CMake build generator
  - Development versions of `libgl`, `libglm`, `libglew` and `libfltk`.
  - The Code::Blocks IDE
- To install those on Ubuntu, run `sudo apt-get install build-essential cmake libgl-dev libglm-dev libglew-dev lifltk1.3-dev codeblocks`

Note: You need a CMake version that is higher than 3.0 in order to generate Code::Blocks projects. If you're using Ubuntu, make sure to use **at least** Ubuntu 15.04. Any older version will not ship with a recent CMake version.

Note: Users of other distributions might have access to more recent CMake versions (check using `cmake --version`). If this is the case, you might consider to use the Ninja build system instead of make, since it speeds up builds considerably. CMake 3.1 and later produces output files for either system.

Step 3: Generate the build files

- Create a build directory, in which all the files generated by CMake are placed. In this example, we will use `<PROJECT_ROOT>/build`.
  - Open a terminal in the root directory containing the first `CMakeLists.txt` file.
  - Run `mkdir build`
  - It is **not recommended** to generate CMake builds in the project's root directory.
- Let CMake generate the build scripts:
  - In the terminal above, run `cd build` to switch to the newly created build directory.
  - From there, run `cmake -G"CodeBlocks - Unix Makefiles" ..`
  - If you would like to test if the build completes, run `make`.
  - To trigger a cleaning action from the terminal, use `make clean`.
  - If you want to reconfigure the build, just delete the whole directory and run the above steps again (`rm -rf build` from a terminal in the root directory).

Note: CMake will by default generate binaries containing debugging symbols (and lacking optimization). To generate a release build, add `-DCMAKE_BUILD_TYPE=Release` to the cmake line.

Note: If you opted for the usage of Ninja (see the notes for step 2), replace the cmake line by `cmake -G"CodeBlocks - Ninja" ..` You can then run builds using `ninja`; cleaning is done via `ninja clean`.

Step 4: Adapt source files

As we do rely on the package management system of your distribution, we do not provide the forked libraries and headers (in contrast to Windows and macOS). Therefore, we rely on the help of you to ensure a proper build: please remove the `FL_OPENGL3` argument of the `glutInitDisplayMode(...)` function call in the `main(...)` function of the respective project. Please be aware that this has to be done for every project you download from the course website.

Step 5: Open the project, configure Code::Blocks

- Start Code::Blocks.
- Upon the first start, you'll see a dialog listing the compilers detected by Code::Blocks. Confirm.
- Click on `File -> Open...` (`Ctrl+O`)., select the `CG-PROJECTS.cbp` file in your new `build` directory.
- CodeBlocks will open the generated workspace file and display its source files on the left side.
- To get a more convenient file structure, right click on the blue `Workspace` icon on the top, and deselect `Categorize by file types`.
- Try to build all projects using `Build -> Build` (`Ctrl+F9`).
- You can build/run/debug single projects using the combo box in the toolbar (left to the red 'Play'-Icon). By default, it is set to 'all'.
- Start debugging the project called `CG-01-D.01_HelloOpenGL`. Select the project name from said combo box and use `Debug -> Step Into` (`Shift+F7`).
- The IDE will start the debugger and switch to a new perspective. The default debugging view is lacking a display for variables and stack frames, though.
  - To correct this, activate `Debug -> Debugging windows -> Call stack` and `Debug -> Debugging windows -> Watches`. The two missing windows will pop up as new floating dialogs.
  - To attach them to the main window, grab the `Watches` window and drag it to the right of the main window, until you see a highlighted docking area (blue stripes). Stop dragging the window to attach it to the docking area.
  - You can place the `Call stack` window above or below the `Watches` by dragging it towards a corner of the same docking area.
  - To make those changes persistent, use `View -> Perspectives -> Save current` and overwrite the preset called `GDB/CDB debugger:Default`. Code::Blocks will automatically switch to this view whenever the debugger is started.

This document

Converted from Markdown to PDF using the service http://www.markdowntopdf.com or the MacDown app from Tzu-ping Chung.

Author and Date: peter.vonniederhaeusern@bfh.ch, 16 March 2017