COMP3811 – Exercise G.1
Getting started

Contents

In this exercise, you will familiarize yourself with the coding environment used in COMP3811. You will experiment with a sample project that later exercises and courseworks build on. This exercise provides a step-by-step guide to get up and running, and includes some tasks that help you get acquainted with the software stack.

In COMP3811, the software stack consists of C++, OpenGL through the *Glad* loader, and *GLFW* for abstracting OS services such as window management and event handling. COMP3811 targets the C++17 standard, and OpenGL version 4.3. (OpenGL will play a larger role later, in the second half of the module. This first part still uses OpenGL under the hood, but you will not yet need to interact with it directly.)

Exercise 1 covers the following parts:

- Project overview and building the first time
- The main loop and event handling in GLFW
- Manipulating images

```
.
├── exercise1
│   └── main.cpp
├── draw2d
│   ├── forward.hpp
│   ├── color.{hpp,inl}
│   ├── draw.{hpp,cpp}
│   └── surface.{hpp,cpp,inl}
├── support
│   ├── checkpoint.{hpp,cpp}
│   ├── context.{hpp,cpp}
│   └── error.{hpp,cpp}
├── vmlib
│   ├── vec2.hpp
│   └── empty.cpp
├── third_party
│   ├── premake5.lua
│   ├── glad
│   │   └── ...
│   ├── glfw
│   │   └── ...
│   └── stb
│       └── ...
├── premake5*
├── readme.md
└── third_party.md
```

Listing 1: *Project layout.*

Note: You are allowed to reuse code from your solutions to exercises in the courseeworks, *if and only if* you are the sole author of the code in question. Keep this in mind when working on the exercises. While you are encouraged to discuss problems with your colleagues, you should not directly share any code. If you find solutions to your problems online, study the solution and then implement it yourself. Never copy-paste code, and always make sure you understand the code that you write!

1 First Setup

Download the Exercise 1 project files and unpack them. Take a brief look around the project; you can find an overview of the directory structure in Listing 1. The most important parts are:

`exercise1` Source code specific to this exercise. All code in this directory is compiled into a single executable file, found in the `bin` directory after building. This is the main executable that you will run.

`draw2d` 2D drawing functions and related classes used in the first part of COMP3811. You will be asked to implement some methods here in Exercise 1.

`support` Support code for transferring the results of the 2D drawing into OpenGL, such that they can be shown on screen. You won't have to touch this code yet (but everything in here will be covered in the second part of COMP3811).

`third_party` Third party code used by the exercises (and, later, courseworks). See section "Third part software" below.

`premake5*` Files related to Premake, a meta build-system, not unlike CMake. Premake does not require any installation (all files that you need are included in the provided project files), which makes it convenient to distribute in stand-alone projects like this. Detailed instructions follow in this document.

Third party software

Exercise 1 uses a few third party libraries. They are briefly introduced in the list below. The source for the libraries is included in the project, and the libraries are automatically built as part of the normal building process (this means that you do not have to install them separately).

Glad Glad on Github Glad online generator The modern OpenGL API is typically not directly available by just linking against the system's OpenGL library (especially prominent in Windows, whose `OpenGL32.dll` only exposes functions from a very old standard). Instead, the OpenGL API must be loaded dynamically at runtime, by asking the system for the various entry points (functions) that OpenGL defines. The Glad OpenGL loader takes care of this, and provides a standard-OpenGL–looking programming experience to us.

As a consequence, one *must always* include Glad's generated header (`<glad.h>`) *instead* of the standard OpenGL header (`<GL/gl.h>` on most systems; `<OpenGL/gl.h>` on MacOSX).

GLFW GLFW To render anything on screen, we must interact with the operating system. Minimally, we must ask for a window into which we can draw, and we must be able to receive input events from the system. GLFW provides a simple cross-platform interface for this. It plays a similar role as Qt in this respect, but is much simpler and does not provide all the features of Qt. GLFW is designed with graphics applications and games in mind, and specifically supports OpenGL rendering.

Premake Premake is a meta build-system, not unlike CMake. Premake does not require any installation, which makes it convenient to distribute. The generated project files do not depend on Premake, so they can also be distributed easily.

stb Select single header libraries by Sean Barret, specifically `stb_image.h` and `stb_image_write.h`. The former includes functions to load images from disk. The latter defines a few functions to store images to disk. Both libraries will be used later.

You can find additional information in the `third_party.md` file, including information on the licenses of the various third party components.

Debug vs. Release build configurations

It's a common practice to have multiple *build configurations*, or *builds* for short. Different builds typically use different compiler options, but may also subtly change the code that is compiled. In some cases, different builds will also link against different libraries.

The most common setup is to have a *debug build* and a *release build*. The debug build typically disables (or limits) compiler optimizations and asks the compiler to generate debug information. This makes it easier to debug the code using debugging tools. The release build, in contrast, enables optimizations and does not generate debug information. This makes it more difficult to debug a program, but can significantly boost runtime performance. Compilation of debug builds is frequently a bit faster, compared to release builds (aside from optimizations taking time, tools like Visual Studio may utilize incremental builds to cut down build times further).

The project defines both a debug and a release build. Instructions for switching them are included below. You typically want to do most of your work with debug builds (such that you can easily use debuggers and similar tools). You should nevertheless test the release build occasionally.

In addition to the differences listed above, the handed-out code changed behaviour slightly between debug and release builds. First, the code uses the standard `assert()` macro to perform additional run-time checks in debug builds. The macro is ignored (compiled out) in release build. Secondly, in debug builds, the code sets up additional OpenGL debugging by creating a "debug OpenGL context" and enabling OpenGL's debug output. Debug output is feature that was added into core OpenGL in version 4.3 and significantly improves the debugging experience in OpenGL. (However, this becomes more relevant in later exercises.)

Building on Linux (command line)

If you are in the UG teaching lab (2.05) in Bragg, you should start by loading the `gcc` module:

```
> module load gcc
```

Note: don't type the '>' symbol! This is used to distinguish input lines (commands that you type) from output generated by the command. This will give you a more recent GCC compiler version (11.2.0). Verify that this is the case by checking the version of the C++ compiler, `g++`:

```
> g++ --version
g++ (Spack GCC) 11.2.0
Copyright (C) 2021 Free Software Foundation, Inc.
(...)
```

The output should contain a line similar to something like `g++ (Spack GCC) 11.2.0`. Verify that the version number at the end is correct. You will need to do this in every terminal/shell (but, of course, you may choose to automate it by adding it to your `.bashrc` or similar).

If you are using a non-standard machine (e.g., own laptop), you probably don't need to do this. However, you should still verify that you have a recent GCC version available on your machine.

For the following, make sure you are in the directory that contains the premake files (e.g., `premake5`, ...). The first step is to run `premake5` to generate makefiles for the projects:

```
> ./premake5 gmake2
Building configurations...
Running action 'gmake2'...
Generated Makefile...
(...)
Done (48ms).
```

You will need to rerun this command if you add new source files or rename/move source files around. However, for this first exercise, there is little need to do so. You should consider the generated makefiles to be build artifacts, and, if you use version control, you probably want to exclude them from the repository (like any other generated files).

Next, you can use `make` to build the project:

```
> make -j6
==== Building x-glad (debug_x64) ====
==== Building x-glfw (debug_x64) ====
(...)
Linking exercise1
```

The `-j6` flag indicates to make that `make` should run up to six build processes in parallel. You can adjust this number, somewhat depending on the number of CPU cores that you have in your machine (check `/proc/cpuinfo`).

The default is to perform a *debug* build. This is reflected in the name of the final binary, which you can find in the `bin` directory. Run it:

```
> ./bin/exercse1-debug-x64-gcc.exe
```

(The generated binary gets a `.exe` extension even on Linux. This is non-standard, but has proven helpful in the past.)

Launching the application should open a empty (black) window. You can close it by pressing the `Escape` key (or through any standard means).

To perform a *release* build, use:

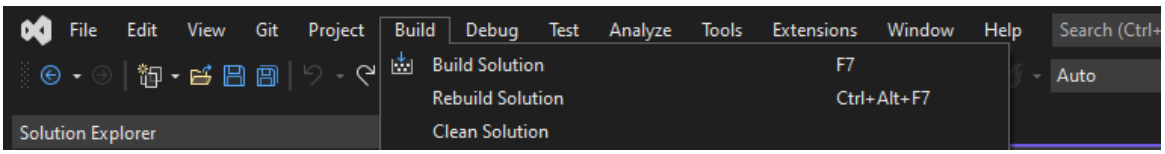

Figure 1: *Visual Studio's* Build *menu. There are further options in the menu that might be relevant to you – check them out.*

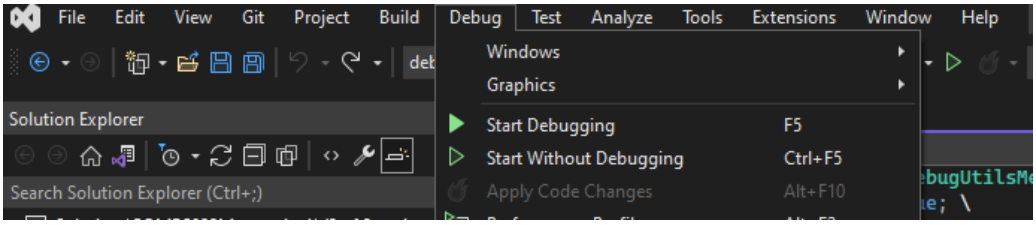

Figure 2: *Visual Studio's* Debug *menu. From here, you can launch the currently selected project ("startup project") either in the Visual Studio debugger or without it.*

```
> make -j6 config=release_x64
==== Building x-glad (release_x64) ====
==== Building x-glfw (release_x64) ====
(...)
Linking exercise1
```

The `x64`-suffix refers to the platform. In this case, the build targets a 64-bit x86 architecture.

Building on Windows (Visual Studio)

If you own a Windows machine, you can opt to use Visual Studio for your work. Visual Studio has in recent years gained a very competent and up-to-date C++ compiler. It also includes one of the best and easy-to-use debuggers. The Visual Studio community edition is available for free for individuals and for academic use. For details, see the separate document that describes how to get and install Visual Studio on your own machine. The following assumes that you have set up Visual Studio 2022 for C++ development on your machine.

To generate the Visual Studio solution and project files, use `premake` from the command line:

```
> .\premake5.exe vs2022
```

Similarly to the Linux build above, you will need to rerun this command if you add a new source file or rename/move source files around.

The main Visual Studio solution file (`COMP3811-exercise1.sln`) will be placed in the root directory of the downloaded code. Open it to launch Visual Studio.

To build the project(s) in Visual Studio, you can either explicitly issue a build command via the *Build* menu (Figure 1), or by launching a program from the *Debug* menu (Figure 2). The latter will try to build (or ask if it should try to build) the program before running it.

In the debug menu, you can launch the program either in the debugger (*Start Debugging*, `F5`) or "normally" (*Start Without Debugging*, `Ctrl-F5`). The latter option is useful for quick tests, as it launches the program without switching over the Visual Studio interface to the debugger. Either option will try to launch the project that is currently designed as the *Startup Project*. To change the project, right-click on any project in the *Solution Explorer* and designate it as *Startup Project*. (Note that trying to start a library project, rather than an application project will not do much.)

The options *Rebuild Solution*/*Clean solution* in the *Build* menu can be useful if you run into mysterious errors that cause linking (and sometimes even compilation) to fail. This doesn't happen very often, but it's a relatively cheap option to try for smaller projects. Note that cleaning a solution will *not* get rid of all of Visual Studio's temporary files or build artifacts. You will still have to pay attention what files you include in e.g. coursework submissions.

To switch between debug and release build (Visual Studio refers to these as "configurations"), use the small pull-down menu that is by default located below the *Build* and *Debug* menus.

Notes

While you are encouraged to use the School of Computing machines in the UG teaching lab, you are of course free to work and solve COMP3811 exercises on your own machine. Similarly, you are free to use your IDE of choice. However, COMP3811 instructors can only provide support on the standard machines and when building according to the instructions above (we will still try to help if we can).

Note that the courseworks will need run in the standardized environment of the school, so you should test things on the school machines regularly.

2 GLFW Main Loop and Event Handling

You are likely familiar with the Qt framework. Qt is a rather large framework and exposes quite a large amount of functionality. Nevertheless, one of the core components of Qt is its functionality relating to development of GUI applications. As such, Qt contains mechanisms to create and manage windows, and to receive and process events ("signals" in Qt-terminology) from the operating system/graphical environment. Any graphical application that can react to user input will need this functionality.

COMP3811 uses GLFW, which is a light-weight cross-platform library. GLFW provides functions to create and manage on-screen windows, and allows the application to receive and process events, but not much more. GLFW was designed as a supporting library for OpenGL-based applications (it now also supports the OpenGL|ES and Vulkan rendering APIs).

One major difference between Qt and GLFW is that Qt typically "takes over" the application to run its own main loop (e.g. `QApplication::exec`). GLFW instead lets the programmer write their own main loop. This is especially useful for interactive and real-time applications that want to render at a constant frame-rate.

Open up the `exercise1/main.cpp` file. It contains the `main()` function, which is the entry point of the program. In the `main()` function, you can find a `while()` loop. This is the main loop of the application. Most of the application's lifetime will be spent in this loop. We will exit the loop only when the application's main window is closed.

Study the main loop. The main loop is kept relatively simple: each iteration of the main loop produces one image/frame. The main loop can be split into roughly four parts:

1. GLFW's event processing
2. Updating the application's state
3. Drawing the next frame
4. Displaying the frame on screen

Similar main loops can be found in many games and other interactive programs. (The main loops can get quite a bit more complex in more advanced applications. These might need to perform additional tasks, and typically want use threads to utilize multiple threads. Furthermore, processing events multiple times per frame and other clever tricks can reduce input latency, which is very desirable in fast-paced games.)

Event processing

Event processing in GLFW involves two components. First, we need to tell GLFW to process events. This is done through either the `glfwWaitEvents()` or `glfwPollEvents()` functions. Both functions will cause GLFW to check for events from the operating system/window environment, process events if such are available, and, finally, pass the events on to our application (if we have requested this). The main difference is that `glfwWaitEvents()` will wait (=block) until an event is available. In contrast, `glfwPollEvents()` will check for events, and if no events are available, it will return immediately. The waiting behaviour is useful for applications that mainly react to user input. The polling behaviour is useful if we want to render images at a fixed frame rate.

GLFW reports events to our application via callback functions. These correspond roughly to Qt's "slots". Callbacks are simple functions that we register with GLFW, such that GLFW can call them when appropriate. You can find additional information in GLFW's Input Guide.

The program currently defines one callback function, called `glfw_cb_key_`. It is declared towards the top of the `main.cpp` source file, and defined towards the end. It is registered with GLFW after window creation with the `glfwSetKeyCallback` function. The key callback is called whenever a keyboard-related event occurs, such as a key being pressed or released. Study the definition of the `glfw_cb_key_` function. Currently, it checks for

the case when the `Escape` key is pressed, at which point it asks GLFW to close the current window (which ultimately exits the main loop and the application).

Task: Define a cursor position callback function. This callback should be called whenever the mouse is moved. The prototype for the function is as follows:

```
void glfw_cb_motion_( GLFWwindow* aWindow, double aMouseXPos, double aMouseYPos );
```

Declare and then define the function. For now, just output the mouse position (x and y coordinate) to the console. To register the function with GLFW, use the `glfwSetCursorPosCallback` function from GLFW.

Study the output. What is the minimum and maximum mouse position that you can observe? How is the coordinate system for the mouse positions oriented relative to the window/screen?

Task: Next, define a mouse button callback. The callback should be called when one of the mouse buttons is pressed. The prototype of the function is:

```
void glfw_cb_button_( GLFWwindow* aWindow, int aButton, int aAction, int /*aMod*/ );
```

Declare and define the function, and register it with GLFW using the `glfwSetMouseButtonCallback`. Now change the program such that it prints the mouse position whenever the left mouse button is pressed. The `aButton` argument identifies which of the mouse buttons is pressed. GLFW provides a set of named constantants, such as `GLFW_MOUSE_BUTTON_LEFT`, to identify common mouse buttons. Check the documentation for the full list of constants. The `aAction` argument identifies the action that took place. A value of `GLFW_PRESS` indicates that the button was pressed, whereas `GLFW_RELEASE` indicates that it was released. (The final argument, `aMods` is unused, and left unnamed to avoid warnings. It would indicate if any modifier keys – shift, control or similar – are held down at the time of the event.)

You will need to figure out a way to pass information between the different callbacks. For this exercise, you can opt for the simple solution of passing the data through global variables.

The more elegant solution involves defining a custom structure (`struct`) that holds the relevant data. An instance of this structure can then be registered with GLFW through the user pointer mechanism with the `glfwSetWindowUserPointer`. The pointer can then later be retrieved using `glfwGetWindowUserPointer`.

There are few more types of input that GLFW provides through callbacks. Refer to the GLFW documentation for details. (GLFW also supports input from gamepads and joysticks. These work slightly differently.)

Updates

After processing events, an application will typically want to update its state based on events and other input, as well as on the time that has passed since the last frame. In a game, this would typically involve computing the next step in a simulation, running AI tasks, updating animations, updating player position and so on.

In this simple example, there is nothing that needs to be updated, so this step is left empty.

Drawing

After updates, the application will draw a new frame. Graphics applications, such as games, frequently draw the whole frame from scratch. This is somewhat in contrast to UI toolkits, which may try to limit redrawing to only components that require this at the time.

At the moment, the application isn't drawing anything interesting. However, it will still clear the main *surface* to black. (Computer graphics frequently uses the term "surface" to refer to a drawable image.) We will add a bit more drawing in Section 3.

Displaying on screen

After the application has drawn the frame, it needs to be shown on screen. Currently, the application simply holds the image data in memory (in the `Surface` object, which is defined in `surface.hpp` and implemented in `surface.inl` and `surface.cpp`). To show the image on screen, we first hand it over to OpenGL, such that we can use OpenGL can draw it to the back *framebuffer* of the application's window. Next, the application asks the OS to show the back framebuffer on screen, via `glfwSwapBuffers`.

There are multiple reasons for this process. On a modern computer with a modern operating system, we typically cannot directly draw to the screen. There are multiple applications that share the screen's contents. One application's window may partially obscure another. Further, the memory that represents the image shown on screen may not be directly accessible by the CPU (e.g., it may be owned by the GPU, and reside in

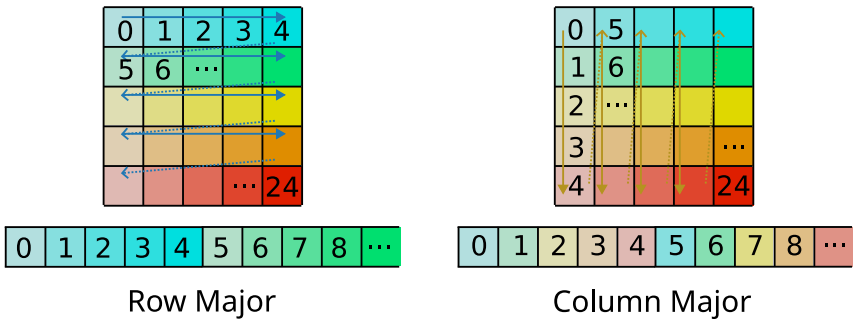

Figure 3: *Row-major vs. column-major storage. Top part of the shows a* 5×5 *image. Bottom part illustrates how the image is stored in linear memory.*

dedicated VRAM on said GPU). The operating system/windowing environment may also apply transforms to the image before showing it (e.g., while compositing it with other windows' contents).

We will discuss this mechanics further when we discuss the OpenGL pipeline. Using multiple framebuffer images is frequently referred to as "double buffering". On modern systems, this is the default – in fact, systems may use more than just two buffers (front and back). Single buffering is frequently no longer supported directly (but drivers may emulate its behaviour for older OpenGL applications).

3 Drawing things

Study the `Surface` class. It simply stores an image in (system) memory. The image is stored in an RGBX format in the sRGB color space, with 8 bits per channel, for a total of 32 bits (= 4 bytes) per pixel. The fourth component is ignored (hence the name 'RGBX' instead of 'RGBA', where the fourth component would be used as an alpha channel). The image is stored in row major form in a single "array"/buffer. In row major form, each rows of pixels are stored consecutively in memory (Figure 3).

The `Surface` class exposes the `Surface::set_pixel_srgb` method to modify the image. This method is currently not implemented. It takes three arguments: `std::size_t aX`, `std::size_t aY` and a `ColorU8_sRGB` color value. The latter is a custom type defined in `draw2d/color.hpp`, and holds a single sRGB color value with three 8-bit channels (`r`, `g` and `b`). The `Surface::set_pixel_srgb` method is supposed to set the pixel at (aX, aY) to the specified color.

Task: Implement the `Surface::get_linear_index` method (`draw2d/surface.inl`). It takes two arguments, `std::size_t aX` and `std::size_t aY`, and should return the linear memory index of the pixel inside the `mSurface` array.

Both methods are declared `inline`. We want the compiler to be able to inline the functions when they are called (at least in optimized builds). This avoids overheads related to calling functions. We expect these functions to be called very frequently -for each pixel that we draw-, so reducing overheads is key. (The exact meaning of the `inline` keyword in C++ is a bit weird, and technically only tangentially related to inlining. Nevertheless, we must use the keyword with the function definitions here, as the function definitions will show up in multiple compilation units.)

Task: Implement the `Surface::set_pixel_srgb` method (`draw2d/surface.inl`), using the `Surface::get_linear_index` method. *Note: the* `assert()` *statement should be left as the first statement in the definition of* `set_pixel_srgb` *intact. Only add code below it.*

Verify that you can set some pixel's colors by adding a few calls `set_pixel_srgb` inside of the main loop, where drawing takes place (e.g., after the call to `Surface::clear` and before the call to `Context::draw`. For example:

```
// ...
surface.set_pixel_srgb( 10, 100, { 255, 255, 255 } ); // Set pixel (10,100) to white (255,255,255)
// ...
```

Study the position of the pixels in the window. Compare the pixel's coordinates to the mouse coordinates reported by GLFW. What do you observe?

Drawing a rectangle

Open up `draw2d/draw.hpp`. The header currently declares two functions, `draw_rectangle_solid` and `draw_rectangle_outline`. Both take the same arguments. The first argument is a mutable `Surface` reference. The purpose of the functions is to draw axis-aligned rectangles into the passed-in surface. The next pair of arguments, `aMinCorner` and `aMaxCorner`, are coordinates for the minimum and maximum corners of the axis-aligned rectangle. They use the type `Vec2f`, which is defined in `vmlib/vec2.hpp`. The final argument is of type `ColorU8_sRGB` and specifies the color in which the rectangle should be drawn.

One might call `draw_rectangle_solid` as follows:

```
draw_rectangle_solid(
    surface,
    {  50.f, 300.f },
    { 200.f, 450.f },
    { 255, 0, 0 } // red
);
```

This should draw an axis-aligned red rectangle of size 150×150 pixels ($200 - 50 = 150$, $450 - 300 = 150$) spanning from the coordinate $(50, 300)$ to $(200, 450)$ (Figure 4).

Task: Implement the function `draw_rectangle_solid` (`draw2d/draw.cpp`). It should use `Surface::▽ ▷ set_pixel_srgb` to set affected pixels of the surface to the provided color.

Try drawing a few rectangles by adding a few calls to `draw_rectangle_solid` to the main loop. What happens when you swap the min and max coordiantes of the box, such that (for example) `aMinCorner.x` is larger than `aMaxCorner.x`. Can your implementation handle that?

Task: Implement the function `draw_rectangle_outline` (`draw2d/draw.cpp`).

Drawing many rectangles

Task: Extend the application such that you can draw rectangles using the mouse. A rectangle should be defined by two consecutive clicks with the left mouse button. The first click defines one of the corners for the rectangle, and the second click defines the other corner. Store the rectangles in a list (e.g. `std::vector`), and draw them each frame.

The teaser image (first page) is drawn that way. It shows somewhere around 16 rectangles, and demonstrates what one would refer to as "programmer art".

Figure 4: *Red rectangle. The bottom left corner of the rectangle is located at* $(50, 300)$*. The top right corner is at* $(200, 450)$*. (Note that the image scales with the PDF, so it might not be the same size as your window.)*