CSCI 4110 Lab One

Image Display

Introduction to CSCI 4110 Laboratories

The purpose of the laboratories is to introduce tools and techniques that are useful in graphics programming. The laboratories develop some of the essential skills that are required by graphics programmers. Some of the material in the early labs you should already been familiar with, but we will approach the topics in a different way. The aim is to build a tool set that you can use in the assignments and future projects.

Many of the laboratories are based on OpenGL, which has a very long history and multiple versions. OpenGL evolved from Iris GL, which was developed by SGI around 1984 to support their new graphics hardware. It was a rushed job and had some pretty rough edges. The first version of OpenGL fixed a number of these problems and was released in 1991. OpenGL 1.1 was the first version that wasn't owned by SGI and was released in 1997. To date there have been 18 releases of the main branch of OpenGL development (the latest version is 4.6). There is also OpenGL ES for mobile devices and WebGL for the web. More recently there is OpenGL SC for safety critical applications. All the modern versions of OpenGL have the same basic core API with restrictions and extensions for specific domains.

Needless to say there have been many changes to OpenGL is the past 20+ years and you will still find programs that are written in older versions of OpenGL. For example, Microsoft Windows still ships with OpenGL 1.3 support. Some programs are still written using older versions of OpenGL, since they don't require the new features.

Starting with version 3.2 the OpenGL specification was divided into a core profile and a compatibility profile. The core profile contains the new OpenGL features and completely supports the programmable pipeline. The aim is to evolve the core profile so OpenGL, OpenGL ES and WebGL all have the same core functionality. The compatibility profile includes many of the older features that still need to be supported for legacy code. The compatibility profile is only supported on the desktop version of OpenGL.

There is also the recently released Vulkan standard which is in some ways the successor to OpenGL. It actually goes back to some of the features of previous versions of OpenGL claiming that it will be more efficient (if it's more efficient why did we remove them in the first place?). This continuous re-invention is common in graphics.

In the laboratories we will use both the compatibility and core profiles, so you become familiar with both of them. There are some features in the compatibility profile that are easier to use than the corresponding ones in the core profile.

Note: the compatibility profile will not go away in the near future. Many of the important support libraries used with OpenGL, such as glut, use the compatibility profile. There have been promises of replacements for these libraries, but they have yet to appear.

Introduction to Laboratory One

In this laboratory we will explore how images are displayed in OpenGL and in the process introduce two libraries: FreeImage and GLFW. We will study rendering algorithms in this course, so we need some way of displaying the images that we will produce. This laboratory will introduce the techniques that we will use.

Why display images in OpenGL? Image display is 2D and OpenGL is 3D. One of the benefits of OpenGL is that it is cross platform, so programs using OpenGL run on multiple operating systems. If we use the native image display code in a particular operating system our applications are no longer portable. In addition we might want to display images in a program that largely does 3D graphics and don't want to learn yet another API.

In this laboratory we will start by creating a simple image (see figure 1) and then display this image using OpenGL and GLFW. We will then use FreeImage to store the image in a file and then read it back again into our program.

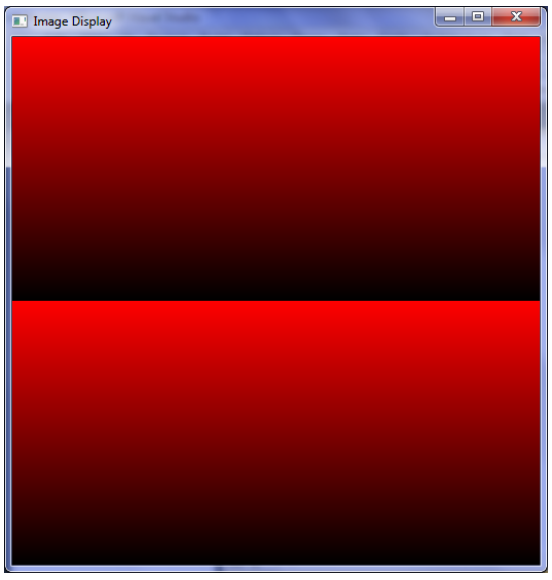


Figure 1 Our image

We will start by creating our image. As you can see it is a simple gradient where the red component increases as we go up the image. We can store an image in a three dimensional array using the following code:

```
#define WIDTH 512
#define HEIGHT 512
```

```
unsigned char image[HEIGHT][WIDTH][3];
```

Note that the first dimension is the height of the image (the number of scan lines), the second dimension is the width of the image (the number of pixels on a scan line) and the third dimension is the colour component. Essentially the first dimension is y and the second dimension is x. Why do we do this? First, this is the order that OpenGL is expecting, so we might as well play along. Second, and maybe more important, it is more efficient. If we examine how this array is laid out in memory, the three colour components of a pixel are next to each other in memory. Similarly, pixels on the same scan line are next to each other. This improves the cache performance of our program, which can be quite significant if we have a lot of large images.

The code for constructing the image is:

```
void makeImage() {
	int i;
	int j;

	for(j=0; j<HEIGHT; j++) {
		for(i=0; i<WIDTH; i++) {
			image[j][i][0] = j % 256;
			image[j][i][1] = 0;
			image[j][i][2] = 0;
		}
	}
}
```

This is pretty straight forward and needs no further description.

GLFW

The interaction between OpenGL and the underlying operating system can be quite complex and varies greatly from one operating system to another. We would like to write programs that are easy to port from one operating system to another and we don't want to worry about the low-level details. Over the years a number of libraries have been produced to serve this purpose. The first one of these was glut, which you may have seen before. Glut hides all of these details and makes it much easier to write OpenGL programs. Glut has a very long history and was written to support the examples in the first Red book (the official book on OpenGL). It was originally written at SGI and while its source code is freely available, its license doesn't allow for the modification of the source code. Thus Glut itself is stuck in the 1990s and cannot be updated for new versions of OpenGL or new operating systems. There have been reimplementation of glut in libraries like FreeGlut, but these libraries are now dated and don't support many of the features of new graphics cards.

A modern alternative to glut is GLFW (https://www.glfw.org/) which supports a wider range of features and APIs. You can download GLFW from its website or the resources part of the web page for the course.

The main part of our program is a while loop that continually displays our image and listens for user input. We could put the graphics code in this loop, but it is better practice to use a separate display function. The display function is:

```
void display () {
	glLoadIdentity();
	glViewport(0,0,511,511);
	gluOrtho2D(0.0, WIDTH, 0.0, HEIGHT);

	glClearColor(0.0,0.0,0.0,0.0);
	glClear(GL_COLOR_BUFFER_BIT);

	glDrawPixels(WIDTH, HEIGHT, GL_RGB, GL_UNSIGNED_BYTE, image);
	glFlush();

}
```

The heart of this function is the glDrawPixels() call. The first two parameters and the last one should be obvious. The third parameter specifies the format of the image; it is a standard RGB image. The fourth parameter states that we are using unsigned bytes to store each pixel component.

Our main program is a bit longer, its code is:

```
int main(int argc, char **argv) {
	GLFWwindow *window;

	glfwSetErrorCallback(error_callback);

	if (!glfwInit()) {
		fprintf(stderr, "can't initialize GLFW\n");
	}

	makeImage();

	saveImage((char*)"image.png");

	//	loadImage("image.png");

	window = glfwCreateWindow(512, 512, "Simple example", NULL, NULL);

	if (!window)
	{
		glfwTerminate();
		exit(EXIT_FAILURE);
	}

	glfwSetKeyCallback(window, key_callback);
```

```
	glfwMakeContextCurrent(window);
	GLenum error = glewInit();
	if (error != GLEW_OK) {
		fprintf(stderr, "Error starting GLEW: %s\n", glewGetErrorString(error));
		exit(0);
	}

	glfwSwapInterval(1);

	while (!glfwWindowShouldClose(window)) {
		display();
		glfwSwapBuffers(window);
		glfwPollEvents();
	}

	glfwTerminate();

}
```

If you have used glut before you will notice that this program is longer. A lot of this is due to better error recovery. The program starts with a call to `glfwSetErrorCallback`, this sets a global error handler for our program. The code for our error handler is:

```
void error_callback(int error, const char* description)
{
	fprintf(stderr, "Error: %s\n", description);
}
```

This isn't particularly fancy, but it is good enough for most of our programs. This is followed by a call to the GLFW initialization function. This is followed by a call to makeImage to build our image, and then a call to save the image (we will examine this function shortly). The next few lines create the window where our image will be displayed and displays a message if window creation fails.

Whenever the user interacts with our program a callback function will be called to process the user input. Our program has very simple input requirements. We need to know when the user wants to terminate the program and we provide a command for flipping the image (more on this later). Our callback function for processing input is:

```
static void key_callback(GLFWwindow* window, int key, int scancode, int action, int mods)
{
	if (key == GLFW_KEY_ESCAPE && action == GLFW_PRESS)
		glfwSetWindowShouldClose(window, GLFW_TRUE);
	if (key == GLFW_KEY_F && action == GLFW_PRESS)
		flip = !flip;
}
```

The call to `glfwSetKeyCallback` informs GLFW of the callback procedure to call. The next set of lines initializes an extension loader library called glew. OpenGL evolves faster than most operating systems so relying on the operating system to provide the appropriate include files isn't

realistic. Instead a number of extension loader libraries have been developed to simplify the process. For this laboratory we really don't need to use glew, but it's a good idea to get used to including it. All our programs will use double buffering, the call `glfwSwapInterval(1)` controls how frequently this occurs. The parameter is the number of screen refreshes to wait between swapping buffers. Finally we have the while loop where we display the image and listen for user input.

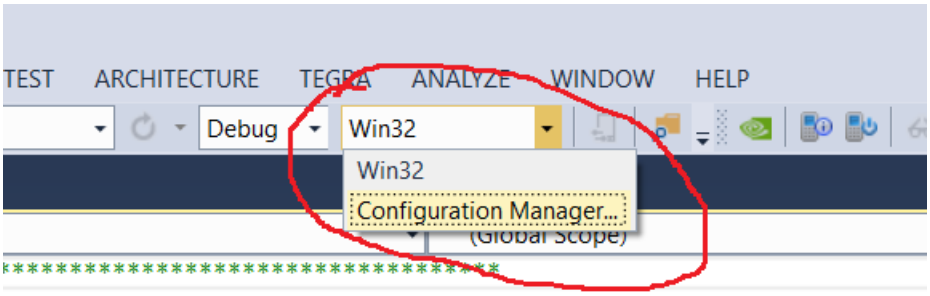
Now we are ready to put everything together and run our program. You will need the following include files:

```
#include <FreeImage.h>
#include <GL/glew.h>
#define GLFW_DLL
#define GLFW_INCLUDE_NONE
#include <GLFW/glfw3.h>
#include <stdio.h>
#include <stdlib.h>
```

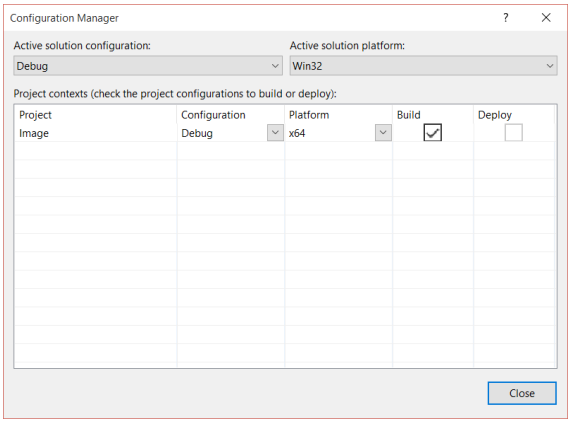
Aside – Visual Studio

The laboratories for this course were developed using Visual Studio 2017 on Windows. Due to the gaming community graphics support on Windows is better than other platforms. I know that some of you are Linux users, unfortunately OpenGL support on Linux tends to be quite variable and usually not as good as Windows. Unfortunately, the graphics card business is quite competitive, therefore, certain card manufacturers are not overly willing to support their latest drivers on Linux. Most of the labs should work okay on Linux given an appropriate build system.

By default Visual Studio produces 32 bit code, but we will be using 64 bit libraries so the first thing you will need to do is change the configuration to 64 bit. Start the configuration manager in the following way:



Once there, select New from under Active Solution Platform and then select x64 in the top pull down list. The result should look like the following:



Note, Visual Studio can be very stubborn about this, so always check the configuration manager. It may say it's doing a 64 bit build when it reality it's really doing a 32 bit build

Now we need to add the location of the include files, the location of the library files and the libraries that we need. To do this right click on the project and select properties. Select C/C++ and the first entry is the location of the include files. Click on this entry, select edit from the drop down menu and then browse to the location of the include files for GLFW, glew and FreeImage (you should have downloaded them from Blackboard). Setting up the libraries is similar. Start by opening Linker in the properties and then selecting General. About half way down you will find Additional Library Directories. Select edit from the drop down menu and add the library directories. Next for the library files, select Input and then Additional Dependencies. This is where you place the library names, by simply typing the name of the libraries. The ones that you need to add are "FreeImage.lib", "glfw3dll.lib", "glew.lib", "opengl32.lib" and "glu32.lib". After you have compile your program make sure to copy the corresponding DLLs to the execution directory, or install them in the system directory.

Laboratory Continued

When you run the program you will notice that it doesn't quite produce that results that we intended. The image is upside down; how did this happen? The first scan line of an image is displayed at the top of the screen and not at the bottom as you might expect. In an image the y coordinate points down the screen and not up like in a regular coordinate system. This is something we need to be careful of in our rendering algorithms.

We can solve this problem by flipping the image, which is quite easy to do in OpenGL:

```
void display () {

	glLoadIdentity();
	glViewport(0,0,511,511);
	gluOrtho2D(0.0, WIDTH, 0.0, HEIGHT);

	glClearColor(0.0,0.0,0.0,0.0);
	glClear(GL_COLOR_BUFFER_BIT);
```

```
	if(flip) {
		glRasterPos2i(0,511);
		glPixelZoom(1.0f, -1.0f);
	} else {
		glRasterPos2i(0,0);
		glPixelZoom(1.0, 1.0);
	}

	glDrawPixels(WIDTH, HEIGHT, GL_RGB, GL_UNSIGNED_BYTE, image);
	glFlush();

}
```

Here we using the global variable flipped to indicate whether the image should be flipped. The glPixelZoom() procedure specifies scaling factors for pixel coordinates. If the two parameters to this function were both 2.0 the image would be twice as big. In this case we are multiplying the y pixel coordinates by -1.0 to flip the image, but when we do this the y coordinate will now all be negative and thus below our window. We solve this problem be setting the origin of the image to (0, 511) using the glRasterPos2i procedure.

Our keyboard callback function already processes the 'f' key and changes the value of the flip variable.

FreeImage

We will use the FreeImage library to save and restore images. There are several good choices in this area; both OpenCV and ImageMagick are good choices. FreeImage has the basic functionality, is reasonably easy to use and the build process is quite straight forward. Since we are only interested in saving and loading images we don't need the extra features of the other libraries.

The saveImage procedure shown below can be used to save an image. The single parameter to this procedure is the name of the file the image will be saved in.

```
void saveImage(char *filename) {
	int i;
	int j;
	FIBITMAP *bitmap;
	BYTE *bits;

	bitmap = FreeImage_Allocate(WIDTH, HEIGHT, 24);
	for(j=0; j<HEIGHT; j++) {
		bits = FreeImage_GetScanLine(bitmap,j);
		for(i=0; i<WIDTH; i++) {
			bits[FI_RGBA_RED] = image[j][i][0];
			bits[FI_RGBA_GREEN] = image[j][i][1];
			bits[FI_RGBA_BLUE] = image[j][i][2];
			bits += 3;
		}
```

```
	}

	FreeImage_Save(FIF_PNG, bitmap, filename, PNG_DEFAULT);

}
```

This procedure starts by calling FreeImage_Allocate to create a new image. The parameters to this function are the width, height and depth (in bits) of the image. It returns a pointer to a FIBITMAP structure. FreeImage attempts to optimize memory layout for performance, therefore image data cannot be accessed in the same way as an array. FreeImage pads scan lines to an optimal number of bytes for memory access, so we need to access images a scan line at a time. Thus, our procedure starts with a for loop over the scan lines in the image. This is followed by a for loop over the pixels in a scan line. The FreeImage_GetScanLine procedure is used to retrieve a scan line. This is basically a simple array of bytes for the pixels in the scan line. As you can see FreeImage has constants for accessing the colour component within a pixel, this allows us to rearrange the colour components of a pixel for different file formats.

After the image data has been transferred to FreeImage we use the FreeImage_Save procedure to save the image to a file. The first parameter to this procedure is a constant that specifies the file format. In this case we are using png, so FIF_PNG is used for this parameter. The second parameter is the image to be stored and the third parameter is the file name. The fourth parameter is used for options in writing the file. The FreeImage documentation has tables that summarize the file formats and options that can be used.

The loadImage procedure is used to load an image from a file. The source code for this procedure is shown below.

```
void loadImage(char *filename) {
	int i,j;
	FIBITMAP *bitmap;
	BYTE *bits;

	bitmap = FreeImage_Load(FIF_PNG, filename, PNG_DEFAULT);
	for(j=0; j<HEIGHT; j++) {
		bits = FreeImage_GetScanLine(bitmap,j);
			for(i=0; i<WIDTH; i++) {
				image[j][i][0] = bits[FI_RGBA_RED];
				image[j][i][1] = bits[FI_RGBA_GREEN];
				image[j][i][2] = bits[FI_RGBA_BLUE];
				bits += 3;
			}
	}
	FreeImage_Unload(bitmap);
}
```

The FreeImage_Load procedure transfers the image data from the file to a FIBITMAP structure, which we can then transfer to our image array. This procedure ignores the extension part of the file name, so we must specify the type of image we are loading. The first parameter is the image

type and the third parameter specifies any optional processing, in the same way as the FreeImage_Save procedure. The second parameter is the name of the file the image is stored on. The FreeImage_Unload function is called at the end of the procedure to free the space used by the FIBITMAP structure.

Add these two procedures to your program and show that you can save and load images. Note you will need to save the image before you try to load it.

Laboratory Report

Once you have finished your program show it to the TA and demonstrate its capabilities. If you cannot finish the work in the lab, email a zip file to the TA containing your program code.