Lab 5 – Fonts

In this lab we are going to load some fonts into an OpenGL Texture and then display it on the screen. We then move onto how we can move to 2D rendering to overall some UI elements, this will involve working with some render states.

1. Fonts

OpenGL has no native way to load and render fonts, so again we are going to use an SDL library called SDL_TTF to load a font, then we will use a function from SDL_TTF to render some text to a SDL_Surface, then finally we will convert the surface to an OpenGL Texture.

1.1 SDL_TTF

The first task we will have to do to link against the SDL_TTF library

- You should add **C:\Specialist Applications\SDL2_ttf-2.0.12\include Additional Include Directory**
- You should add for **C:\Specialist Applications\SDL2_ttf-2.0.12\lib\x86** to **Additional Library Directory** for the project
- You should add **SDL2_image.lib** to the **Additional Dependencies**
- **Copy all *.dll** files from **C:\Specialist Applications\SDL2_ttf-2.0.12\lib\x86** to the Debug Directory of the project **(**or follow the instructions from **Debug Envirnomnet.doc)**

1.2 Initialise SDL_TTF

We should first add an include statement for SDL_TTF, you should add the following to the top of **main.cpp**

```
#include <SDL_ttf.h>
```

Navigate to the **main function** in **main.cpp** and add the following just after we initialise SDL

```
if (TTF_Init() == -1) {
        std::cout << "ERROR TTF_Init: " << TTF_GetError();
}
```

1.3 Loading Fonts

We are going to add a function to load a font from file, and then render some text using this font. We will add this to our Texure.cpp and Texture.h.

Before continuing on with the rest of this task you will have to **include** the header file for **SDL_TTF** in Texture.h

Add the following function declaration to **Texture.h**

```
GLuint loadTextureFromFont(const std::string& fontFilename, int pointSize,
                           const std::string& text);
```

Now switch to the **Texture.cpp** and add the following definition for the above function

```
GLuint loadTextureFromFont(const std::string& fontFilename, int pointSize,
                           const std::string& text)
{
        GLuint textureID = 0;
        return textureID;
}
```

This function will take in a string which represents the font file we are going to load, the 2nd parameter is an integer which represents the point size we require for our and finally the last parameter is the text we are going to render.

We are now ready to load a font, we will use an SDL_TTF function to do this, add the following snippet of code just after we set the **textureID** to **zero** and before we return the **textureID**

```
TTF_Font * font = TTF_OpenFont(fontFilename.c_str(),pointSize);
if (!font)
{
        std::cout << "Unable to load font " << fontFilename << " " << TTF_GetError();
        return textureID;
}
```

The above function loads a Font using **TTF_OpenFont**, this returns a pointer to a **TTF_Font**. If the font is not loaded then we return the textureID which is zero.

Now that we have a loaded font, we can use another SDL_TTF function to render some text to an SDL_Surface.

Add the following code snippet just after the closing brace of the above if statement

```
SDL_Surface *textSurface = TTF_RenderText_Blended(font, text.c_str(), { 255, 255, 255 });
```

The above function takes in an pointer to a loaded TTF_Font, the 2nd parameter is the string we are going to render and the last parameter is an SDL_Color which contains red, blue and green colours(0 - 255). This is only one of SDL Render Text functions, visit the following URL for discussion on each method

http://sdl.beuc.net/sdl.wiki/SDL_ttf_Functions_Render

We now have an SDL Surface so we can use pretty much exactly the same code to convert this surface to OpenGL texture as we do for the loading a texture from a file. The only issue we have is that we should decouple the converting code away from the loading the texture.

1.4 Convert Surface to OpenGL Texture

We are going to create a new function which will encapsulate the conversion from an SDL Surface to an OpenGL Texture. Then we will sue function anytime we need to carry out this conversion.

Add the following function declaration to **Texture.h**

```
GLuint convertSDLSurfaceToGLTexture(SDL_Surface * surface);
```

Now switch to Texture.cpp and add the following definition for the above function

```
GLuint convertSDLSurfaceToGLTexture(SDL_Surface * surface)
{
	GLuint textureID = 0;
	return 0;
}
```

The full body of this function is pretty much been implemented in **loadTextureFromFile**. You could cut all the code from this function(from and including **GLint nOfColors line** to the **glTexImage2D** function call) and paste it between the **textureID** and **return 0 call**. Finally you should free the SDL_Surface that has been passed in.

The function should look something like this

```
GLuint convertSDLSurfaceToGLTexture(SDL_Surface * surface)
{
	GLuint textureID = 0;
	 GLint  nOfColors = surface->format->BytesPerPixel;
	GLenum textureFormat = GL_RGB;
	GLenum internalFormat=GL_RGB8;
	if (nOfColors == 4)     // contains an alpha channel
	{
		if (surface->format->Rmask == 0x000000ff){
			textureFormat = GL_RGBA;
			internalFormat=GL_RGBA8;
		 }
		else{
			textureFormat = GL_BGRA;
			internalFormat=GL_RGBA8;
		 }
	}
	else if (nOfColors == 3)     // no alpha channel
	{
		if (surface->format->Rmask == 0x000000ff){
			textureFormat = GL_RGB;
			internalFormat=GL_RGB8;
		}
		else
		{
			textureFormat = GL_BGR;
			internalFormat=GL_RGB8;
		}
	}
	glGenTextures(1, &textureID);
	glBindTexture(GL_TEXTURE_2D, textureID);
	glTexImage2D(GL_TEXTURE_2D, 0, internalFormat, surface->w, surface->h, 0,
			textureFormat, GL_UNSIGNED_BYTE, surface->pixels);

	SDL_FreeSurface(surface);

	return textureID;
}
```

1.5 Finishing of Font Rendering

We can now use this utility function to convert the surface we generate in the font loading function, switch back to **loadTextureFromFont** and after the call to render to the SDL Surface we should call the above conversion function

```
textureID=convertSDLSurfaceToGLTexture(textSurface);
```

We now want to set some initial OpenGL texture states, add the following to setup some initial filtering and texture address modes

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
```

And finally we can close the font, this will clean up any meemory allocated when we opened it

```
TTF_CloseFont(font);
```

2. Rendering Fonts

Now that we have a function which renders a font and converts it to a OpenGL we can then map this onto any surface like any normal texture.

2.1 Font Directory and Font download

Create a new directory called **fonts** inside the assets directory, now download the font from GCU learn and copy this to the directory you just created.

2.2 Adding Font Path

Add a new string called **FONT_PATH** to main.cpp, this should have the values **fonts**. This should be similar to the texture and shader paths.

2.3 Global Variable for Font Texture

Add a new global unsigned integer called **fontTexture**, this should be very similar to the existing texture contained in the file.

2.4 Loading the Font

Add a new function called **createFontTexture** this should be pretty much exactly the same as the **createTexture** function but instead of load a texture we are going to use our font loading function and set the return value of this call to the above **fontTexture.**

You should now call this **createFontTexture** from the **main** function, you should place this at the same place as the call to **createTexture**.

2.5 Bind the Font Texture

Switch down to the render function and instead of binding the normal texture change the texture to **fontTexture.**

2.6 Cleanup

Inside the **CleanUp** function make sure you delete the texture, this should be exact same as the previous deletion of the other texture.

2.7 OpenGL Renderstates

If you run the application at this point you will probably see nothing, the reason for this that the majority of the texture is transparent. In order to see the cube with the text, you need to turn on alpha blending and setup a blending function.

Add the following code at the top of the **render** function after the call to **glClear**

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

The above code will switch on alpha blending(glEnable) and then the second line sets up the initial blending function, this will blend the new pixel with what is on the frambuffer. See the following URL for more details - https://www.opengl.org/sdk/docs/man2/xhtml/glBlendFunc.xml

3. Moving to 2D

Typically when we render elements like text we want to draw it in 2D Space rather than 3D. This requires us to have two view matrices, one which defines a perspective camera(look at, position, centre and up) and another which defines a orthographic project(dimensions of the screen).

3.1 Orthographic Projection

Orthographic Projection is a means of representing a 3D object in 2D space. To calculate this we will use the GLM library. Navigate to the update function and comment out the line that calculates the projection matrix. Now add the following line to replace to calculate the view matrix using orthographic projection.

```
projMatrix = glm::ortho(0.0f, (float)WINDOW_WIDTH, (float)WINDOW_HEIGHT,0.0f,0.1f, 100.0f);
```

The **ortho** function takes in the following parameters

1. Left coordinate of our projection(0.0f in this case, which means the left side of the screen)
2. Right coordinate of our projection(Window width in this case, which means the right of the screen)
3. Top coordinate of our projection(Window height, which means the bottom of the screen)
4. Bottom coordinate of our projection(0.0f, which means the top of the screen)
5. Near Clip(any vertex less than this amount will be clipped)
6. Far Clip(any vertex more than this amount will be clipped)

Run the application and you will see that you cube will be very small, there is a couple of ways of fixing this. We could add a call to **glm::scale** and multiply the call to **glm::translate** to scale our cube or we could define a new set of vertices defined in screen space.

3.2 Building a 2D Sprite

We are going to use our **fontTexture** to generate a sprite which has the same dimensions as the texture.

Add the following function to **main.cpp** before the **main** function

```
void initGeometryFromTexture(GLuint textureID)
{

}
```

We now are going to fill out this function, first thing we will do is retrieve the size of the passed in texture

```
int width, height;

glBindTexture(GL_TEXTURE_2D, textureID);
glGetTexLevelParameteriv(GL_TEXTURE_2D, 0, GL_TEXTURE_WIDTH, &width);
glGetTexLevelParameteriv(GL_TEXTURE_2D, 0, GL_TEXTURE_HEIGHT, &height);
```

The above snippet binds the texture into the pipeline and then uses the glGetTexLevel* functions to retrieve the width and height of the texture.

We can now use these values to build up a sprite

```
Vertex spriteData[] = {
        { vec3(0.0, 0.0f, 0.0f), vec2(0.0f, 0.0f), vec4(1.0f, 1.0f, 1.0f, 1.0f) },// Top Left

        { vec3(0.0f, height, 0.0f), vec2(0.0f, 1.0f), vec4(1.0f, 1.0f, 1.0f, 1.0f) },// Bottom Left

        { vec3(width, height, 0.0f), vec2(1.0f, 1.0f), vec4(1.0f, 1.0f, 1.0f, 1.0f) }, //Bottom Right

        { vec3(width, 0.0f, 0.0f), vec2(1.0f, 0.0f), vec4(1.0f, 1.0f, 1.0f, 1.0f) }// Top Right

        };

GLuint spriteIndices[]={
  0,1,2,
  0,3,2,
};
```

We are going to generate our vertex array

```
glGenVertexArrays(1, &VAO);
glBindVertexArray(VAO);
```

Vertex Buffer

```
glGenBuffers(1, &triangleVBO);
glBindBuffer(GL_ARRAY_BUFFER, triangleVBO);
glBufferData(GL_ARRAY_BUFFER, 4 * sizeof(Vertex), spriteData, GL_STATIC_DRAW);
```

Element Buffer

```
glGenBuffers(1, &triangleEBO);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, triangleEBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, 6 * sizeof(int), spriteIndices, GL_STATIC_DRAW);
```

And finally the vertex attributes

```
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), NULL);
glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void**)sizeof(vec3));
glEnableVertexAttribArray(2);
glVertexAttribPointer(2, 4, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void**)(sizeof(vec3) +
                                    sizeof(vec2)));
```

You will now need to call this function, add it to the **main** function, but ensure that you call it after the **fontTexture** has been created. You should also pass in the **fontTexture** as the first parameter.

If you now run this application you should see the text now displayed in 2D space.

3.3 Mixing 3D and 2D

We are now able to mix 3D and 2D drawing, we need to refactor code a little. First off we need to create functions that will be used to carry out the following functionality

- Create 2D Scene
- Create 3D Scene
- Update 2D Scene
- Update 3D Scene
- Render 2D Scene
- Render 3D Scene
- Cleanup 2D Scene
- Cleanup 3D Scene

You will also need to have 2D and 3D versions of following

- Vertex Array Object
- Vertex Buffer Object
- Element Buffer Object
- Textures
- Shaders

You should now have an attempt at implementing the above, if you have any issues you should contact the lab demonstrator or have a look at the solution in the following GItHub repo branch

https://github.com/BigBearGCU/GP2BaseCode/tree/Ex5-Mixing-2D-3D

NB. The above solution is not the most optimal, we are going to move to a component based system next week.