3D Graphics Programming
Lab 6: Textures

Karsten Pedersen
Department of Creative Technology

October 10, 2018

In this lab we are going to add a texture to our triangle. There are a couple of steps we first need to take as explained in the lecture:

1. Load the image
2. Upload to the GPU (as a texture)
3. Add texture coordinate VBO to the VAO.
4. Modify the fragment shader to map the texture
5. Set the uniform to point to the relevant texture **unit**.

The first step is to load the image. We do this using an image loader to avoid having to read and parse the (often complex!) image format ourselves. The following is a list of common loaders and their supported file formats.

- SDL (inbuilt) (BMP)
- SDL_image (PNG, BMP, JPEG, TIFF)
- lodepng (PNG)
- stb_image (PNG, BMP, JPEG)

For this lab, we will be using *stb_image* due to the fact it has no dependence on external libraries (DLLs) and supports a large number of different file formats. An archive containing this library is provided with the lab. Extract it and add it to your C++ project Create a sample image with a size of 256x256 and add it next to your project file. With this in place, you can now load an image with the following code:

```
#include <stb_image/stb_image.h>

...

int w = 0;
int h = 0;
int channels = 0;

unsigned char *data = stbi_load("image.png", &w, &h, &channels, 4);

if(!data)
{
  throw std::exception();
}
```

Now that we have loaded this image into memory as an *unsigned char **, we now need to upload this to the GPU. This can be done with the following:

```
// Create and bind a texture.
GLuint textureId = 0;
glGenTextures(1, &textureId);

if(!textureId)
{
  throw std::exception();
}

glBindTexture(GL_TEXTURE_2D, textureId);

// Upload the image data to the bound texture unit in the GPU
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, w, h, 0, GL_RGBA,
  GL_UNSIGNED_BYTE, data);

// Free the loaded data because we now have a copy on the GPU
free(data);

// Generate Mipmap so the texture can be mapped correctly
glGenerateMipmap(GL_TEXTURE_2D);

// Unbind the texture because we are done operating on it
glBindTexture(GL_TEXTURE_2D, 0);
```

With the texture in place, we now need to modify our Vertex Array to include a texture coordinate buffer (so we know how to map the texture in the fragment shader).

```
  VertexBuffer *positions = new VertexBuffer();
  positions->add(glm::vec3(0.0f, 0.5f, 0.0f));
  positions->add(glm::vec3(-0.5f, -0.5f, 0.0f));
  positions->add(glm::vec3(0.5f, -0.5f, 0.0f));

  VertexBuffer *texCoords = new VertexBuffer();
  texCoords->add(glm::vec2(0.5f, 0.0f));
  texCoords->add(glm::vec2(0.0f, 1.0f));
  texCoords->add(glm::vec2(1.0f, 1.0f));

  VertexArray *shape = new VertexArray();
  shape->setBuffer("in_Position", positions);
  shape->setBuffer("in_TexCoord", texCoords);
```

You no longer need *in_Color* in this example so remove it because we are effectively obtaining the color for each pixel from the texture instead. You may need to alter your *VertexArray* wrapper to handle *in_TexCoord* as well as your *ShaderProgram*.
Now that you have changed the attribute streams that you are passing into the shader, you must modify the vertex shader so that the names match. Because we cannot really do anything useful with the texture coordinates in the vertex shader, just pass the texture coordinates through into the fragment shader where they will be used.
The fragment shader should then use the *texture2D* function to extract a pixel sample from the texture at the given coordinate and simply use that value as the output colour. The following fragment shader demonstrates this:

```
uniform sampler2D in_Texture;

varying vec2 ex_TexCoord;

void main()
{
  vec4 tex = texture2D(in_Texture, ex_TexCoord);
  gl_FragColor = tex;
}
```

With this in place, now in the program loop, each time before we draw the triangle, we want to set the active texture unit to 1 and bind the texture.

```
glActiveTexture(GL_TEXTURE0 + 1);
glBindTexture(GL_TEXTURE_2D, textureId);
```

Finally we then instruct the uniform sampler within the shader to sample data from the active bound texture in texture unit 1

Note:
The *in_Texture* can simply be set using *glUniform1i*. Make sure that you do not use *glUniform1f* because it will not work. Your wrapper class should provide a *setUniform* override for both data types.

```
shader->setUniform("in_Texture", 1);
```

Note:
The uniform value that we pass in here is **NOT** the texture ID. It refers to the active texture unit (1).

By the end of this lab, your project should look similar to that in **Figure 1**.

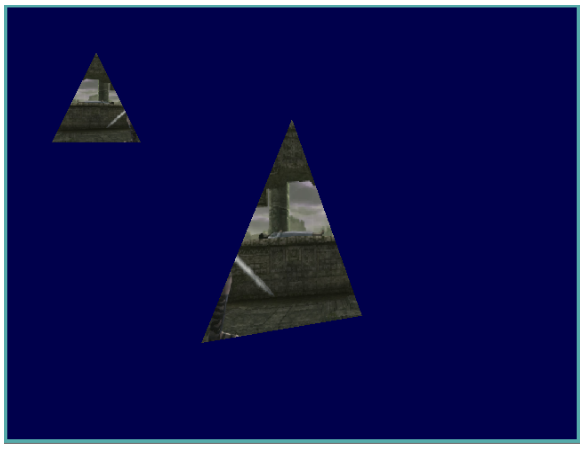

Figure 1: *The final output of the textured triangle*