COMP3811 – Synopsis S.0 Textures

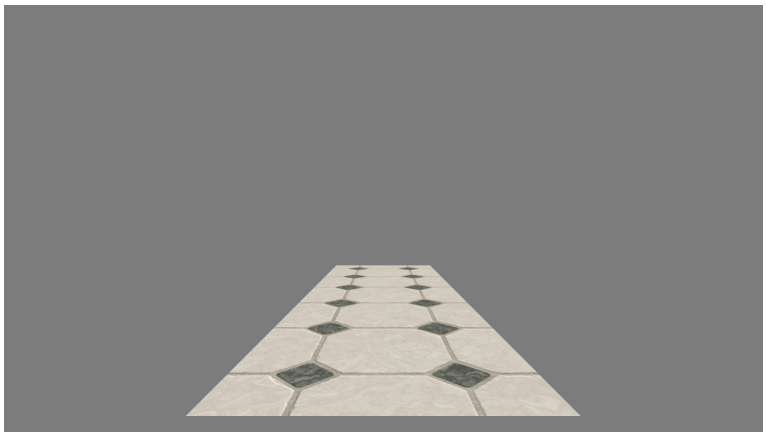
Contents

1 Texture Coordinates 1

2 Defining the texture 2

3 Sampling textures (GLSL) 3

4 Drawing 4



This document provides a brief overview of texturing. It will summarize the steps required to draw a simple textured plane (see teaser image).

The overview assumes that you have a working sample from one of the previous exercises that includes the `SimpleMeshData` structure. It is recommended that you create a fresh copy of this sample, as you will need to modify the `SimpleMeshData` structure and related functions for texturing. A working camera will be helpful as well.

You can get the texture shown in the screenshots by clicking here (some PDF readers might require a right-click and then selecting *Save Attachment As* or an equivalent option).

1 Texture Coordinates

The first step is to add texture coordinates to each vertex. For this example, vertices will carry two attributes: the position (a `Vec3f`) and a texture coordinate (a `Vec2f`). The following assumes that these are the only two per-vertex input attributes.

First, change the `SimpleMeshData` structure to match the above vertex definition:

```
struct SimpleMeshData
{
   std::vector<Vec3f> positions;
   std::vector<Vec2f> texcoords;
};
```

Also update `create_vao()` to create a matching VAO. This example assumes that you will pass in the position as vertex attribute 0 and the texture coordinate as attribute 1:

```
glBindBuffer( GL_ARRAY_BUFFER, texCoordVBO );
glVertexAttribPointer(
   1,   // location = 1 in vertex shader
   2, GL_FLOAT, GL_FALSE,   // 2 floats, not normalized to 0..1 (GL_FALSE)
   0,   // see above
```

```
   nullptr  // see above
);
glEnableVertexAttribArray( 1 );
```

You will need to create the corresponding VBO and upload the texture coordinates to it. Play close attention to the second argument to `glVertexAttribPointer`. The texture coordinates are 2D vectors, not 3D vectors!

We will "hardcode" the plane geometry, so you you will not need functions such as `concatenate()`. You can remove them for now (or update them to work with the new `SimpleMeshData` definition).

The plane is defined to be at $y = -1$ and extends from $x = -1$ to $x = +1$ and from $z = -3$ to $z = 3$. The plane extends three times as far in the z-direction as in the x-direction. We will assign the texture coordinates as follows. The vertex at position $(-1, -1, 3)$ has texture coordinate $(0, 0)$. The vertex at position $(1, -1, -3)$ receives texture coordinate $(1, 3)$. The texture will cover the plane along the x-axis once. We will configure the texture to repeat, meaning it will be repeated three times along the z-axis.

The mesh definition for the plane is thus:

```
SimpleMeshData sd{
   // Positions
   { { -1.f, -1.f, 3.f },
     { +1.f, -1.f, 3.f },
     { +1.f, -1.f, -3.f },
     { -1.f, -1.f, 3.f },
     { +1.f, -1.f, -3.f },
     { -1.f, -1.f, -3.f } },

   // Texture coordinates
   { { 0.f, 0.f },  { 1.f, 0.f },  { 1.f, 3.f },
     { 0.f, 0.f },  { 1.f, 3.f },  { 0.f, 3.f } }
};
```

(You can briefly skip ahead to the first part of Section 3 if you want to debug this part.)

2 Defining the texture

To load the texture, we will use the `stb_image.h` library. There are three steps to loading the texture:

1. Load the image data from the file using `stbi_load`. Before loading, we will ask the `stb_image` library to flip the texture vertically. More often than not, this matches better with how OpenGL stores the images (but some meshes and 3D models come with textures stored "flipped").
2. Create a texture object and upload the image data into the texture object. Texture object names are generated with `glGenTextures`. Before assigning the texture data with `glTexImage2D`, we will bind the texture with `glBindTexture`. After the call to `glTexImage2D`, we no longer need the CPU copy of the image data (OpenGL has created a copy that is likely stored in VRAM), so we can free it (`stbi_image_free`).
3. Finally, we ask OpenGL to create mipmaps for the texture and setup filtering, addressing and other texture parameters.

The code for will be presented in more detail during the lectures:

```
GLuint load_texture_2d( char const* aPath )
{
   assert( aPath );

   // Load image first
   // This may fail (e.g., image does not exist), so there's no point in
   // allocating OpenGL resources ahead of time.
   stbi_set_flip_vertically_on_load( true );

   int w, h, channels;
   stbi_uc* ptr = stbi_load( aPath, &w, &h, &channels, 4 );
   if( !ptr )
      throw Error( "Unable to load image '%s'\n", aPath );

   // Generate texture object and initialize texture with image
```

```
    GLuint tex = 0;
    glGenTextures( 1, &tex );
    glBindTexture( GL_TEXTURE_2D, tex );

    glTexImage2D( GL_TEXTURE_2D, 0, GL_SRGB8_ALPHA8, w, h, 0, GL_RGBA, GL_UNSIGNED_BYTE,
      ▷  ptr );

    stbi_image_free( ptr );

    // Generate mipmap hierarchy
    glGenerateMipmap( GL_TEXTURE_2D );

    // Configure texture
    glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
    glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR );

    glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE );
    glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE );

    glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_MAX_ANISOTROPY, 6.f );

    return tex;
}
```

Refer to the OpenGL and STB library documentation for additional details.

3 Sampling textures (GLSL)

First, update the vertex shader to accept the texture coordinate as input attribute one, and then pass it on to further stages as an output:

```
layout( location = 1 ) in vec2 iTexCoord; // Update location to match your VAO configuration
                                          // if you usa a custom VAO layout.

// ... (later) ...

out vec2 v2fTexCoord;
```

Note that the texture coordinates are of type `vec2` and *not* `vec3`.

The texture coordinate does not need to be transformed. The vertex shader just needs to pass it on to the next stage in the rendering pipeline:

```
    // Somewhere in main():
    v2fTexCoord = iTexCoord;
```

Next, update your fragment shader to receive the interpolated texture coordinate value:

```
in vec2 v2fTexCoord;
```

(Double-check that the spelling of the name the type (`vec2`) matches the output from the vertex shader.)

At this point, it might be a good idea to briefly visualize the texture coordinates from the fragment shader to ensure that they are being passed through correctly. Draw the texture coordinates as the R and G components of the color output (set B = 0). Compare to Figure 1.

If the texture coordinates seem correct, proceed by definition the texture sampler in the shader:

```
layout( binding = 0 ) uniform sampler2D uTexture;
```

Note the `binding` instead of `location` here. 2D textures use a texture sampler type of `sampler2D`.

We can now sample the texture using the built-in `texture` function in GLSL. For now, we will simply sample the texture using the passed-in texture coordinates:

```
    // somewhere in main()
    oColor = texture( uTexture, v2fTexCoord ).rgb;
```

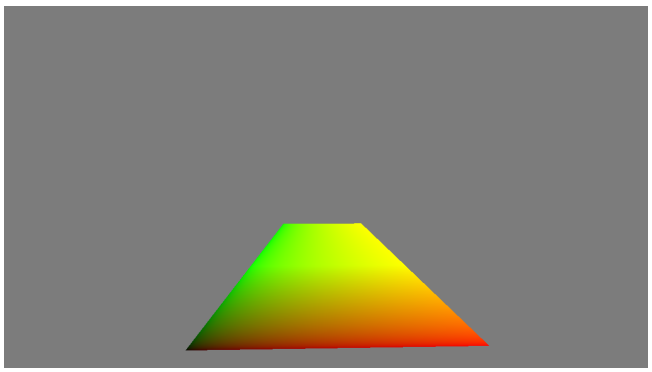


Figure 1: *Visualization of the texture coordinates of the plane mesh.*

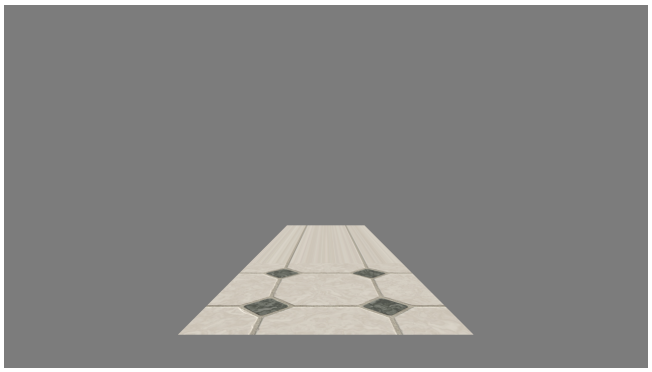


Figure 2: *Initial rendering.*

The `texture()` function returns a `vec4` with RGBA data. In this case, we're only interested in the RGB part, so we can use GLSL's swizzling syntax to extract that.

4 Drawing

Before we can draw the mesh (`glDrawArrays`), we must bind the texture to the specified texture unit (the `binding` number specified in the fragment shader):

```
glActiveTexture( GL_TEXTURE0 );
glBindTexture( GL_TEXTURE_2D, textureObjectId );
```

Here, the `glActiveTexture` selects the texture unit to bind the texture to (unit zero for binding zero), and `glBindTexture` binds the texture to the currently selected unit.

Run the program and verify that the output is as shown in Figure 2.

You may have noticed that the texture is not repeated. To fix this, change the texture addressing mode (`GL_TEXTURE_WRAP_x`) to `GL_REPEAT`. Compare your results to the teaser image.

Optional

- The example uses the `GL_SRGB8_ALPHA8` internal texture type. This ensures that the values sampled from the texture are converted from 8-bit sRGB to linear color values. Temporarily change the internal type to `GL_RGBA`. Can you spot the difference?
- Explore different filtering modes.
- Anisotropic filtering is enabled by setting the maximum anisotropy parameter to 6. This is a somewhat arbitrary choice. Try a few different values and see if you can spot a difference. The minimum value is zero (=disabled) and the maximum value is typically 16.
- Grab the explosion sprite image. This is a PNG image with an alpha channel. Use alpha blending to blend the explosion over the background.