Texturing with GLSL

Notes for a Msc Course in Computer Graphics

University of Minho
António Ramires

Introduction

Texturing is a feature that opens many doors when considering computer graphics. Textures are not necessarily images. Although that representation is the most easily observable, they can also be seen as an array of data.

In this document we will cover the basics of texturing in GLSL.

It is assumed that the reader is familiar with the concept of texturing, and understands the process of defining texture coordinates for a 3D model. Most 3D models are created with texture coordinates, so when importing a model, chances are that the texture coordinates are already defined.

Texture Coordinates

The first step is making the textures coordinates available to our shaders. Texture coordinates are just another vertex attribute, much like normals. Hence in the application we need to treat them as such, i.e. we need to add a buffer with the texture coordinates to the vertex array object that contains all the other models attributes.

In the vertex shader we shall receive the texture coordinates as inputs, and commonly we simply copy them to an output variable. this variable will be received as input in the fragment shader, where we can use it for the purpose of our application.

The code below simply illustrates the process described above:

Vertex Shader

```
#version 330

uniform mat4 pvm;
```

```
in vec4 position;
in vec2 texCoord;

out vec2 texCoordV;

void main() {

	texCoordV = texCoord;
	gl_Position = pvm * position;
}
```

Fragment Shader

```
#version 330

in vec2 texCoordV;

out vec4 colorOut;

void main() {

	colorOut = vec4(texCoord, 0.0, 0.0);
}
```

The output of these shaders displays texture coordinates as colors, showing how the texture mapping is defined in a model. For instance, the figure below shows a plane, an elephant, and the teapot, with their texture coordinates. Red will be used for the s coordinate, and green for the t coordinate.

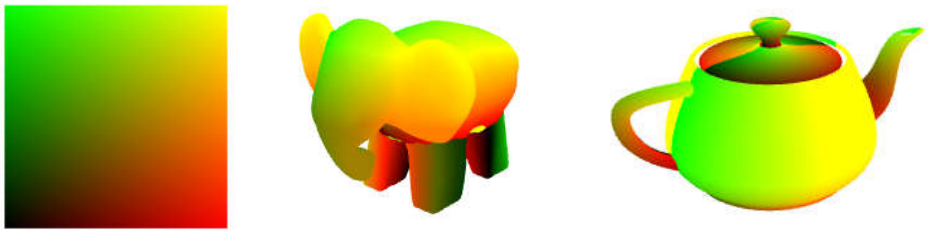

Figure. Drawing texture coordinates as colors

Playing with texture coordinates

In here it will be shown how texture coordinates per se can be used to color a model. For instance, assume that we want to obtain a grid effect as shown in the image below:

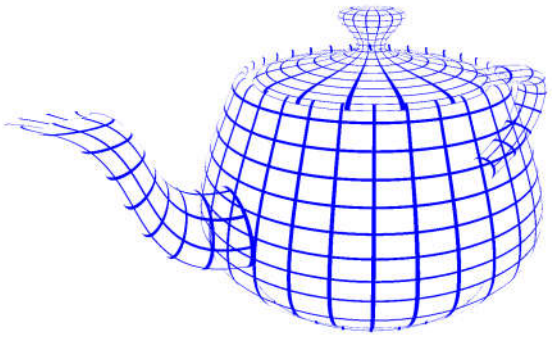

Figure. Grid teapot

To achieve the above we are only painting certain pixels (in blue) and discarding the remaining pixels with the GLSL keyword `discard`. The density of the grid is defined by a multiplication factor applied to the texture coordinates. The fragment shader then selects only those pixels that have one of the texture coordinates with a fractional value below 0.1, colouring these with blue. This last value, 0.1, controls the width of the lines drawn.

The vertex shader is the same as above, only the fragment shader needs to be rewritten as follows:

```
#version 330

in vec2 texCoordV;
out vec4 colorOut;

void main() {
	// different constants will yield different grid densities
	vec2 t = texCoordV * 8.0;

	// the constant (0.1) defines the with of the lines
	if (fract(t.s) < 0.1 || fract(t.t) < 0.1)
		colorOut = vec4(0.0, 0.0, 1.0, 1.0);
	else
		discard;
}
```

A teapot in stripes can be obtained with a similar approach. In this case we can attribute a colour to pixels with the fractional part of a texture coordinate, for instance *t*, lower than 0.5, and another color to pixels with the same fractional coordinate larger than 0.5.

The shader could be as follows:

```
#version 330
```

```
uniform vec4 color1 = vec4(0.0, 0.0, 1.0, 1.0);
uniform vec4 color2 = vec4(1.0, 1.0, 0.5, 1.0);

in vec2 texCoordV;
out vec4 colorOut;

void main() {

      vec2 t = texCoordV * 8.0;
      if (fract(t.s) < 0.5)
            colorOut = color1;
      else
            colorOut = color2;
}
```

The result, when applied to the teapot, will be:

Figure. Stripes in a teapot

As can be seen, the result is severely aliased. This is because the coloring function we are using is a step function. To fix this we can use a function with smoother transitions as depicted in the figure below:

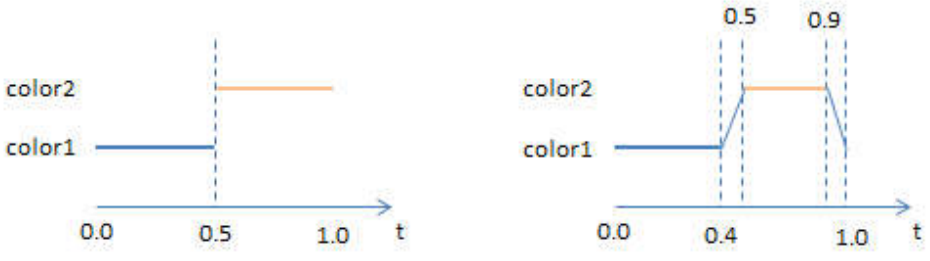

Figure. Left: step function. Right: smoother function

The fragment shader to implement this function is fairly simple. We are going to use GLSL `mix` function to do a linear interpolation on the colors as seen below. The first two parameters of the

`mix` function are the colours to be mixed, and the last parameter specifies how the colours are going to be mixed, according to the following expression:

$$mix(color1,\ color2,\ f) = color2 \times f + color1 \times (1 - f)$$

The fragment shader could be written as follows:

```
#version 330

uniform vec4 color1 = vec4(0.0, 0.0, 1.0, 1.0);
uniform vec4 color2 = vec4(1.0, 1.0, 0.5, 1.0);
in vec2 texCoordV;

out vec4 colorOut;

void main() {

	vec2 t = texCoordV * 8.0;

	float f = fract(t.s);

	if (f < 0.4)
		colorOut = color1;
	else if (f < 0.5)
		colorOut = mix(color1, color2, (f - 0.4)* 10.0) ;
	else if (f < 0.9)
		colorOut = color2;
	else
		colorOut = mix(color2, color1, (f - 0.9) * 10.0);
}
```

The result, presented in the image below (right), shows a considerable improvement.

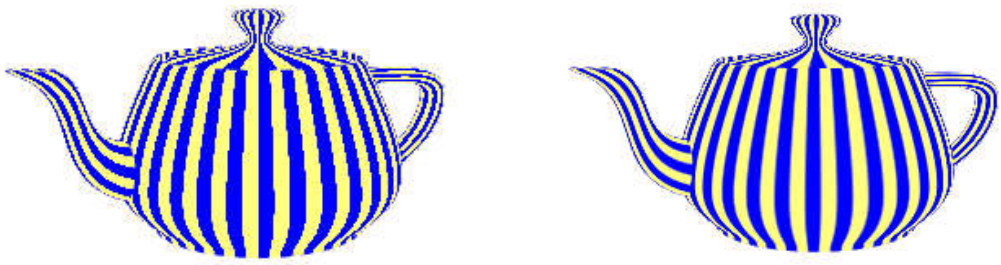

Figure. Stripes computed with step and smooth function

Texturing with an image

First things first, i.e. lets see how to load an image, and create a texture with its data. To load the image we are going to use the DevIL library. DevIL, formerly know as OpenIL, stands for **Dev**elopers **I**mage **L**ibrary. DevIL and OpenGL work in a similar fashion. First we need to generate an image, then bind it, load it, and finally access its data. For the OpenGL side we have to generate a texture, bind it, specify its parameters and send the image data to the driver.

A simple snippet to load a 2D image and create a texture is now presented:

```
ILboolean success;
unsigned int imageID;
GLuint textureID = 0;

// Load Texture Map
ilGenImages(1, &imageID);

ilBindImage(imageID);

// match image origin to OpenGL's
ilEnable(IL_ORIGIN_SET);
ilOriginFunc(IL_ORIGIN_LOWER_LEFT);

success = ilLoadImage((ILstring)filename.c_str());

if (!success) {
      VSLOG(mLogError, "Couldn't load texture: %s",
                                    filename.c_str());
      // The operation was not successful
      // hence free image and texture
      ilDeleteImages(1, &imageID);
      return 0;
}

/* Convert image to RGBA, just to be on the safe side */
ilConvertImage(IL_RGBA, IL_UNSIGNED_BYTE);


/* Create and load textures to OpenGL */
glGenTextures(1, &textureID); /* Texture name generation */
glBindTexture(GL_TEXTURE_2D, textureID);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA,
                        ilGetInteger(IL_IMAGE_WIDTH),
                        ilGetInteger(IL_IMAGE_HEIGHT),
                        0, GL_RGBA, GL_UNSIGNED_BYTE,
                        ilGetData());
```

```
ilDeleteImages(1, &imageID);
```

The variable `textureID` will be used later on the OpenGL setup phase.

Texture IDs are assigned to texture units. The shader will receive, in a uniform variable, the texture unit number, and it will access the texture unit assigned previously to the referred unit.

In OpenGL we can proceed as follows to assign a texture ID to a texture unit:

```
glActiveTexture(textureUnit);
glBindTexture(GL_TEXTURE_2D, textureID);
```

The variable `textureUnit` should then be sent to the shader, as a uniform, where it will be declared as a sampler.

So, let's consider an image that represents the above defined smoother function:

Figure. image enconding smooth color transition

In the shader we will receive the texture unit, and use it to access the texture. In GLSL we will use function `texture` which takes as arguments the texture unit, and the texture coordinates.

Now, we present a fragment shader that gets the same effect as before, using the image above, instead of a coded function inside the shader itself.

```
#version 330

uniform sampler2D texUnit;

in vec2 texCoordV;

out vec4 colorOut;

void main() {

      vec2 t = texCoordV * 8.0;
      colorOut = texture(texUnit, t);
}
```

Another option would be to encode the result of the mix function in the texture, but only for the intensity, as shown in the image below.

Figure. Image encoding intensities

In this case, the shader would use the intensity to mix the colors, and it could be written as follows:

```
#version 330

uniform vec4 color1 = vec4(0.0, 0.0, 1.0, 1.0);
uniform vec4 color2 = vec4(1.0, 1.0, 0.5, 1.0);

uniform sampler2D texUnit;

in vec2 texCoordV;

out vec4 colorOut;

void main() {

      vec2 t = texCoordV * 8.0;

      float f = texture(texUnit, t).r;
      colorOut = mix(color1, color2, f) ;
}
```

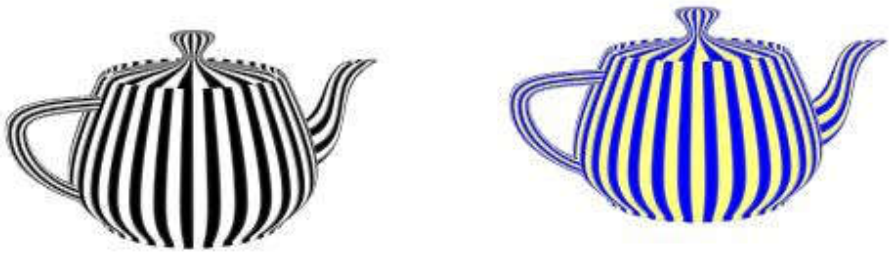

Figure. Using image intensity. Left: alone. Right: with colors