Textures 5

Many objects of the real world do not have uniform colours or colour gradients. Frequently they have patterns that are repeated in a more or less uniform fashion along the object surface. For 3D models we use the term "colour textures" or "textures" for short. The best way to think about textures is to imagine some image that will be (eventually stretched and) wrapped around an object, or part of it.

5.1 Texture coordinates

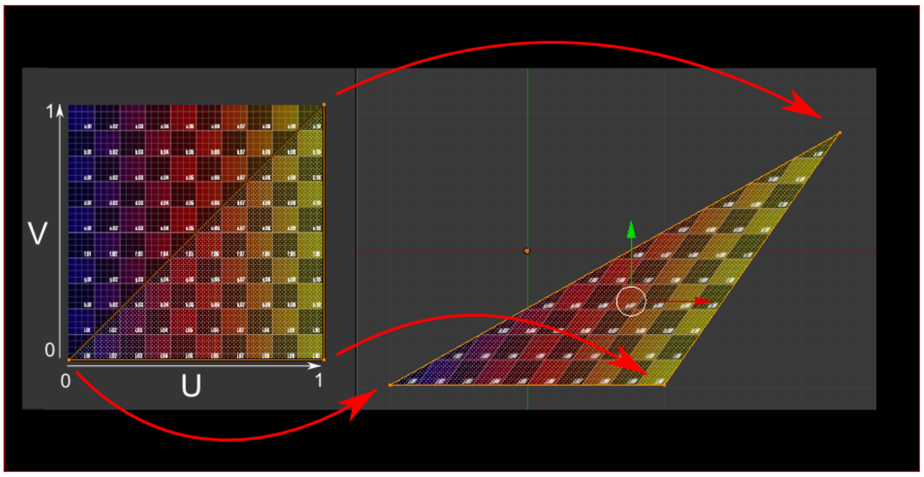


Figure 5.1: Texture coordinate mapping. Image from opengl-tutorial.org

For applying a texture to an object we must first load (or create) an image (or bitmap) in memory and then assign specific points of the image to vertices of the object (the stretching and wrapping operation).

As 3D objects are typically modelled as a mesh of triangles, and each triangle has 3 vertices, then each vertex, besides its 3D coordinates, will also receive some more information about the corresponding location of the triangle texture. This location is specified in the so called UV or texture coordinates.

Considering an image previously loaded in memory, independently of the number of pixels in the horizontal and vertical directions, the texture coordinates will always range from 0 to 1, both in the vertical (V) and horizontal (U) directions. The lower left corner of the image is treated as the origin (0.0) for those coordinates. Figure 5.1 shows the application of a texture to a triangle using the adequate coordinates mapping.

5.2 Creating textures

After having created or loaded an image it must be copied to an OpenGL texture object using the following code.

```
// Create one OpenGL texture
GLuint textureID;
glGenTextures(1, &textureID);

// All future texture functions will apply to this texture
glBindTexture(GL_TEXTURE_2D, textureID);

// Pass the image (data) to OpenGL
glTexImage2D(GL_TEXTURE_2D, 0,GL_RGB, width, height,
                            0, GL_BGR, GL_FLOAT, data);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glGenerateMipmap(GL_TEXTURE_2D);
```

5.3 Applying textures

Texture coordinates and colours may be included together in the vertices information, but each of them specified by the starting offset and stride parameters of glVertexAttribPointer function, as the following example.

```
float vertices[] = {
    // positions          // colors           // texture coords
     0.5f,  0.5f, 0.0f,   1.0f, 0.0f, 0.0f,   1.0f, 1.0f,   // top right
     0.5f, -0.5f, 0.0f,   0.0f, 1.0f, 0.0f,   1.0f, 0.0f,   // bottom right
    -0.5f, -0.5f, 0.0f,   0.0f, 0.0f, 1.0f,   0.0f, 0.0f,   // bottom left
    -0.5f,  0.5f, 0.0f,   1.0f, 1.0f, 0.0f,   0.0f, 1.0f    // top left
};
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 8 * sizeof(float),
                                (void*)(6 * sizeof(float)));
glEnableVertexAttribArray(2);
```

void glVertexAttribPointer(GLuint index, GLint size, GLenum type, GLboolean normalized, GLsizei stride, const void * pointer);

Or as in many other cases, we may just want to pass in a different structure the texture coordinates as

```
float vertices[] = {
        // texture coords
       1.0f, 1.0f,    // top right
       1.0f, 0.0f,    // bottom right
       0.0f, 0.0f,    // bottom left
       0.0f, 1.0f     // top left
};
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 0, 0);
glEnableVertexAttribArray(2);
```

5.4 Shaders

Now these will be processed at the shaders level. Starting with a vertex shader that will receive vertex coordinates, colours and texture coordinates

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aColor;
layout (location = 2) in vec2 aTexCoord;

out vec3 ourColor;
out vec2 TexCoord;

void main()
{
    gl_Position = vec4(aPos, 1.0);
    ourColor = aColor;
    TexCoord = aTexCoord;
}
```

The fragment shader then will access the texture through the sample object,

```
#version 330 core
out vec4 FragColor;

in vec3 ourColor;
in vec2 TexCoord;

uniform sampler2D ourTexture;

void main()
{
    FragColor = texture(ourTexture, TexCoord);
}
```

If we want to mix the texture with the vertices colours we may do this at the fragment shader as

```
#version 330 core
out vec4 FragColor;

in vec3 ourColor;
in vec2 TexCoord;

uniform sampler2D ourTexture;

void main()
{
    FragColor = texture(ourTexture, TexCoord) * vec4(ourColor, 1.0);
}
```

5.5 Exercises

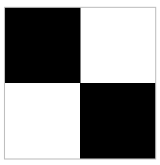
Based on the previous classes code do the following.

Exercise 5.1
Define 2 triangles in 3D coordinates so that they form a square. Translate them to z=-10, define a projection matrix and a view matrix so that the camera has a opening of 45 degrees and is located at $(0, 0, 0)$ looking at point $(0, 0, -1)$. Choose the upvector at will. Correct and adapt the values till you will see the triangles in red colour, and moving along z axis according to $z = 5 + 3 * sin(t)$

Exercise 5.2
Consider the following array of values

```
unsigned char mytexT[]= {
0,0,0,
255,255,255,
255,255,255,
0,0,0
};
```

If viewed as 8 bit values representing RGB triplets, we may think of and image composed only of 2 black and 2 white pixels, as follows



Now you may convert this simple image into a texture and apply to the triangles.

Note that an RGB triplet occupies 3 bytes and GPUs prefer to access elements aligned on 4 byte tuples. As such either you pad your pixels or you change the way they are stored with glPixelStorei(GL_UNPACK_ALIGNMENT,1). In either way you should understand and explain why that works.

5.6 Loading textures

As (2D) textures are commonly stored in some image format, we must be able to load these images into memory. As a result we will have some array of RGB values that later will be used to create a texture object. You must be able to use the following C++ class to load images in BMP and PPM formats and then apply them as textures to objects.

```
class CGRAimage{
    int widthStep;
    static bool pminitialized;
public:
    int width; // columns
    int height; // rows
    unsigned char * data; // RGBRGBRGB...
    CGRAimage();
    CGRAimage(CGRAimage &);
    ~CGRAimage();

    bool loadPPM(const char * fname);
    void savePBM(const char * fname);
    bool loadBMP(const char * fname);
};
```

Exercise 5.3

Load some image from a file of the appropriate type and use it as a texture for the rectangle.

Exercise 5.4

Redraw the cube of the last lesson, but mixing texture and colours, as in the example pictures below.