Tutorial 06

Aditya Prakash

October 5, 2015

1 About

The code is very similar to most of the tutorials before, you can run the code simply by running *./06_texturing*.

2 Understanding the code

For texturing the cube, we firstly need to load the image, that we want to texture the faces with. We will then follow it up creating textures, this part will be very similar to how we create vertex buffers, there will be some parameters, which we will come to later. Once we create the texture, all we need to do is to bind the texture, pass texture coordinates(this will be similar to what you've learnt in class) and then tell fragment shader how to sample color.

2.1 Loading the image

This piece of code deals with loading a bmp image in our C++ code. Note that, this piece of code is exclusive to bmp images only, so our code can only deal with bmp images, for images other than bmp, you can make a few modifications to the code to make it work. The piece of code for loading the image is in "texture.cpp".

```
    file = fopen( filename , "rb" );
    if ( file == NULL ) return 0;  // if file is empty
    fread(header , 1, 54, file);
    // Read  MetaData
    size = *(int*)&(header[0x22]);
    w = *(int*)&(header[0x12]);
    h = *(int*)&(header[0x16]);

    //Just in case metadata is missing
    if(size == 0)
      size = w*h*3;
```

The first line is simple opening of the file that we want to open. The header of a BMP file is 54 bytes, knowing that we read the first 54 bytes, into the header. In the following lines we extract the image size, width and height from the header. Since sometimes images over web may have loss of meta data, we make sure that the image size is not affected. We reassign it the correct value in case size is 0. One example is the convert command in Linux. If you convert an image to bmp using convert command you'll find that the header is not in the format of BMP files. If you want to know more about headers of BMP files, I'll link some references in the end.

Once we do the above stuff, we can easily read data using

```
  data = new unsigned char [size];
```

2.2 Creating textures

Now once we have the image data, we can create our textures. I have included this piece of code also in "texture.cpp", because, if we are reading the data, why not create texture at the same time, instead of keeping the data, and then sometime in future creating textures. The code for generating textures is similar to how we deal with vertex buffers, i.e. GenBuffers followed by BindBuffer, except for, instead of Buffer we use textures.

```
    glGenTextures( 1, &texture ); //texture is GLuint
    glBindTexture( GL_TEXTURE_2D, texture );
```

Nextly, we define some parameters for the texture.

The parameter GL_TEXTURE_MIN_FILTER and

GL_TEXTURE_MAG_FILTER specify how to deal with minification or magnification respectively. As the image size and object size most probably won't be the same. So, these parameters tell OpenGL to how to interpolate if the image is larger/smaller than the object. In our case we are using the basic Linear interpolation (GL_LINEAR). It can also be specified the value GL_NEAREST, which will do the nearest neighbour interpolation, which will the pixelated look to the image. As, it won't do any interpolation, it will assign the color of the nearest pixel. The parameter GL_TEXTURE_WRAP_S, GL_TEXTURE_WRAP_T define, how texture will behave for some value outside the texture(eg in 256x256 image refering to pixel 260), in this case, we are just repeating the same texture(GL_REPEAT). The S and T are for width or height. You can also set it to GL_CLAMP which will set is to Black. This can be different for width and height.

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

Once we have created the buffer, we need to give the image to OpenGL using the currently bound texture, we do this by glTexImage2D.

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0,
   GL_BGR, GL_UNSIGNED_BYTE, data);
```

The parameters for glTexImage2D are, firstly how to interpret the data i.e. 2D textures (GL_TEXTURE_2D). The next parameter is level, this is for mipmaps, if you are using mipmaps. Since, we are not using mipmaps, we set it to 0. The following parameter is how OpenGL is going to store the data, we set it to GL_RGB. Try changing it to GL_BGR and see what happens(Not on the default texure, as it is greyscale so no matter what colors you swap it won't affect output, replace the texture with a different texture, then replace GL_RGB with GL_BGR). The next three parameters are width, height and border to the image. We are not interested in border for this tutorial, so I have set it to 0, you can try and play around with this parameter too(you might need to change width height being passed to glTexImage2D if you set the border). The next parameter is how data is stored in the existing image. In BMP, data is stored in BGR instead of RGB. Next two parameters are about data, what type of data it is, unsigned char, and pointer to the data.

Once we pass texture to OpenGL, we can delete the data. By using free(). This is one of the most important reason why I created texture at the same time as loading image, we don't need to store data for a long time.

By doing this much, we can access the texture in fragment shader. But we still need to pass the coordinate mapping to the vertex shader.

2.3 Passing Texture Coordinates

Since we are dealing with cubes, our job is very easy, we can simply assign texture coordinates as

```
glm::vec2 t_coords[4] = {
  glm::vec2( 0.0, 0.0),
  glm::vec2( 0.0, 1.0),
  glm::vec2( 1.0, 0.0),
  glm::vec2( 1.0, 1.0)
};
```

and while creating our quads, we modify our code simply to

```
void quad(int a, int b, int c, int d, glm::vec4 color)
{
  v_colors[tri_idx] = color; v_positions[tri_idx] = positions
     [a];
  v_normals[tri_idx] = normals[a];
  tex_coords[tri_idx] = t_coords[1];
  tri_idx++;
  v_colors[tri_idx] = color; v_positions[tri_idx] = positions
     [b];
  v_normals[tri_idx] = normals[b];
  tex_coords[tri_idx] = t_coords[0];
  tri_idx++;
  v_colors[tri_idx] = color; v_positions[tri_idx] = positions
     [c];
  v_normals[tri_idx] = normals[c];
  tex_coords[tri_idx] = t_coords[2];
  tri_idx++;
  v_colors[tri_idx] = color; v_positions[tri_idx] = positions
     [a];
  v_normals[tri_idx] = normals[a];
  tex_coords[tri_idx] = t_coords[1];
  tri_idx++;
  v_colors[tri_idx] = color; v_positions[tri_idx] = positions
     [c];
  v_normals[tri_idx] = normals[c];
  tex_coords[tri_idx] = t_coords[2];
  tri_idx++;
  v_colors[tri_idx] = color; v_positions[tri_idx] = positions
     [d];
  v_normals[tri_idx] = normals[d];
  tex_coords[tri_idx] = t_coords[3];
  tri_idx++;
 }
```

We use the following code to load the textures from our main code. Since we can have multiple textures in OpenGL(upto 32 textures). We specify which texture to bind.

```
 GLuint tex=LoadTexture("images/all1.bmp",256,256);
  glBindTexture(GL_TEXTURE_2D, tex);
```

Now that we have created an array with texture coordinates, we need to pass it to our shaders, using the similar practice as positions, colors and normals.

```
GLuint texCoord = glGetAttribLocation( shaderProgram, "
   texCoord" );
..
 glBufferSubData( GL_ARRAY_BUFFER, sizeof(v_positions),
    sizeof(tex_coords), tex_coords);
...
 glEnableVertexAttribArray( texCoord );
  glVertexAttribPointer( texCoord, 2, GL_FLOAT, GL_FALSE, 0,
     BUFFER_OFFSET(sizeof(v_positions)) );
```

By doing this much our work is very much done in the C++ part.

3 Shaders

3.1 Vertex Shader

In vertex shader we read the texCoord and specify the interpolated value of pixel using a varying attribute and pass it to fragment shader. Since the value of varying attribute will anyways be interpolated when it is passed from vertex shader to fragment shader.

```
in vec2 texCoord;
varying vec2 tex;
...
tex = texCoord;
```

3.2 Fragment Shader

In fragment shader we sample the output using a uniform sampler2D.And use an OpenGL function texture2D() to find the color of pixel tex from the sampler. texture2D() reads a pixel from a 2D texture using two parameters. First parameter is a sampler, which texture are we reading from and the second parameter will be our coordinates from the varying variable(You can replace tex, with a 2 vector like vec2(0.5,0.5), what this will do is, sample a pixel as 0.5,0.5 in texture and assign the color of that pixel to entire cube).

```
uniform sampler2D texture;
varying vec2 tex;
....
frag_color = texture2D(texture, tex);
```

4 Output and Multiple textures

If you compile the code successfully, you will see something like the Figure 1.

If you want to use multiple textures instead of the single one, one simple way is to load every single image in a different texture. Alternatively you can create a texture like, Figure 2 and treat the different textures as instead of being from (0,0) to (1,1), we can refer to four textures from (0,0) to (0.5,0.5), (0.5,0.5) to (1.0,1.0), (0.5,0),(1.0,0.5) and (0,0.5)to(1,0.5). I have included the texture of figure 2 in the code as well. In order to use it you might need to make some changes in the C++ code, shaders will remain same.

```
// Modify t_coords to handle 4 textures
glm::vec2 t_coords[4][6] = {
  {glm::vec2( 0.0, 0.0),
   glm::vec2( 0.0, 0.5),
   glm::vec2( 0.5, 0.0),
   glm::vec2( 0.5, 0.5)},
  {glm::vec2( 0.5, 0.0),
   glm::vec2( 0.5, 0.5),
   glm::vec2( 1.0, 0.0),
   glm::vec2( 1.0, 0.5)},
  {glm::vec2( 0.0, 0.5),
```

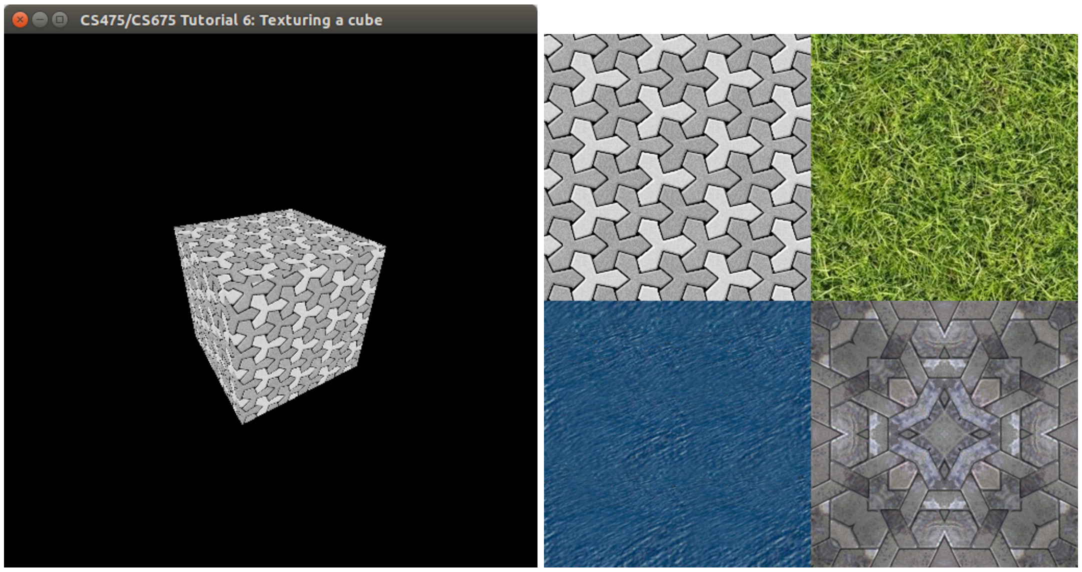

Figure 1: (left) The output of general code (right) Compiling Four different Textures(All images Downloaded from google and compiled into one 512x512 image)

```
   glm::vec2( 0.0, 1.0),
   glm::vec2( 0.5, 0.5),
   glm::vec2( 0.5, 1.0)},
  {glm::vec2( 0.5, 0.5),
   glm::vec2( 0.5, 1.0),
   glm::vec2( 1.0, 0.5),
   glm::vec2( 1.0, 1.0)},

};
....
.....

//Modify the function quad
void quad(int a, int b, int c, int d, int face)
{
  face = face%4;

  v_colors[tri_idx] = color; v_positions[tri_idx] = positions
     [a];
  v_normals[tri_idx] = normals[a];
  tex_coords[tri_idx] = t_coords[face][1];
  tri_idx++;
  v_colors[tri_idx] = color; v_positions[tri_idx] = positions
     [b];
  v_normals[tri_idx] = normals[b];
  tex_coords[tri_idx] = t_coords[face][0];
  tri_idx++;
  v_colors[tri_idx] = color; v_positions[tri_idx] = positions
     [c];
  v_normals[tri_idx] = normals[c];
  tex_coords[tri_idx] = t_coords[face][2];
  tri_idx++;
  v_colors[tri_idx] = color; v_positions[tri_idx] = positions
     [a];
  v_normals[tri_idx] = normals[a];
  tex_coords[tri_idx] = t_coords[face][1];
  tri_idx++;
  v_colors[tri_idx] = color; v_positions[tri_idx] = positions
     [c];
  v_normals[tri_idx] = normals[c];
  tex_coords[tri_idx] = t_coords[face][2];
  tri_idx++;
  v_colors[tri_idx] = color; v_positions[tri_idx] = positions
     [d];
  v_normals[tri_idx] = normals[d];
  tex_coords[tri_idx] = t_coords[face][3];
  tri_idx++;
 }
.......
......

// The calls to quad() also need to be changed
....
// To load texture the call becomes
GLuint tex=LoadTexture("images/all.bmp",256,256);
```

5 References

1. BMP File Format: https://en.wikipedia.org/wiki/BMP_file_format
2. Texture Parameters:
   https://www.khronos.org/opengles/sdk/docs/man/xhtml/glTexParameter.xml