Assignment 8: Texture Mapping

CS Spring 2019

Due Date: Apr. 30th

Follow the instructions carefully. If you encounter any problems in the setup, please do not hesitate to reach out to TA.

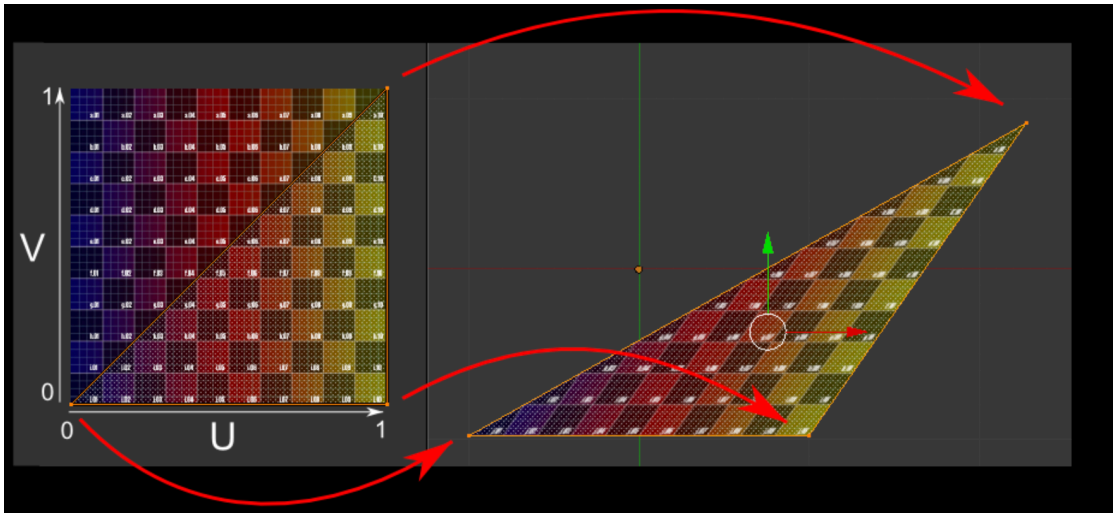
Requirements

In this lab, we will learn about UV coordinate, and use it to load textures in OpenGL.

UV coordinates

When texturing a mesh, you need a way to tell OpenGL which part of the image has to be used for each triangle. This is done with UV coordinates.

Each vertex can have, on top of its position, a couple of floats, U and V. These coordinates are used to access the texture, in the following way:



Using the texture in OpenGL

We arrive now at the real OpenGL part. Creating textures is very similar to creating vertex buffers : Create a texture, bind it, fill it, and configure it.

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

// "Bind" the newly created texture : all future texture functions will modify this texture
glBindTexture(GL_TEXTURE_2D, textureID);

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

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

In glTexImage2D, the GL_RGB indicates that we are talking about a 3-component color, and GL_BGR says how exactly it is represented in RAM. As a matter of fact, BMP does not store Red->Green->Blue but Blue->Green->Red, so we have to tell it to OpenGL.

We'll have a look at the fragment shader first. Most of it is straightforward :

```
#version 330 core

// Interpolated values from the vertex shaders
in vec2 UV;

// Ouput data
out vec3 color;

// Values that stay constant for the whole mesh.
uniform sampler2D myTextureSampler;

void main(){

    // Output color = color of the texture at the specified UV
    color = texture( myTextureSampler, UV ).rgb;
}
```

Three things:

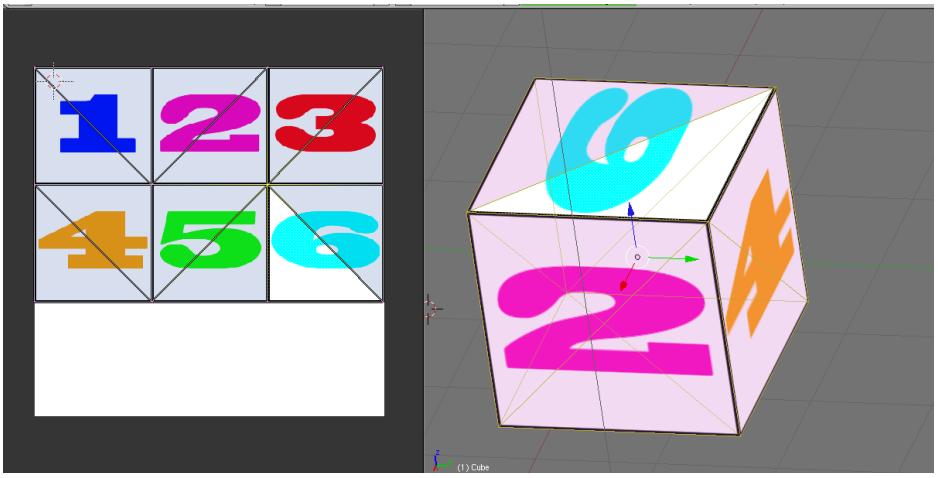
- The fragment shader needs UV coordinates. Seems fair.
- It also needs a "sampler2D" in order to know which texture to access (you can access several texture in the same shader)
- Finally, accessing a texture is done with texture(), which gives back a (R,G,B,A) vec4. We'll see about the A shortly.

The vertex shader is simple too, you just have to pass the UVs to the fragment shader :

```
#version 330 core

// Input vertex data, different for all executions of this shader.
layout(location = 0) in vec3 vertexPosition_modelspace;
layout(location = 1) in vec2 vertexUV;

// Output data ; will be interpolated for each fragment.
out vec2 UV;

// Values that stay constant for the whole mesh.
uniform mat4 MVP;

void main(){

    // Output position of the vertex, in clip space : MVP * position
    gl_Position =  MVP * vec4(vertexPosition_modelspace,1);

    // UV of the vertex. No special space for this one.
    UV = vertexUV;
}
```

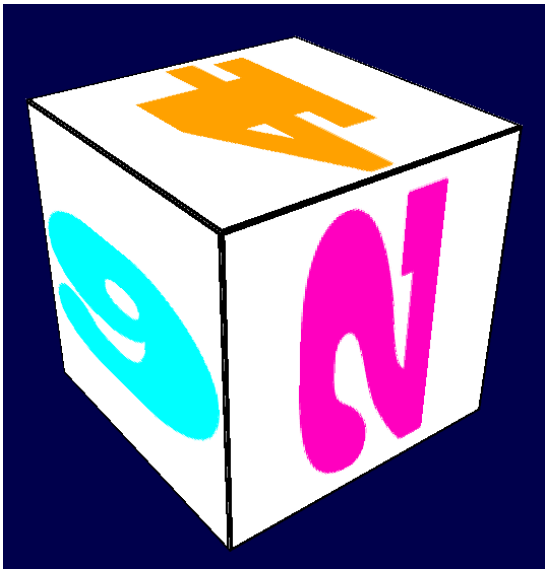
You must be familiar with “layout(location = 1) in vec2 vertexUV”? Well, we’ll have to do the exact same thing here, but instead of giving a buffer (R,G,B) triplets, we’ll give a buffer of (U,V) pairs.

The UV coordinates above correspond to the following model :



The rest is obvious. Generate the buffer, bind it, fill it, configure it, and draw the Vertex Buffer as usual. Just be careful to use 2 as the second parameter (size) of glVertexAttribPointer instead of 3.

This is the result :

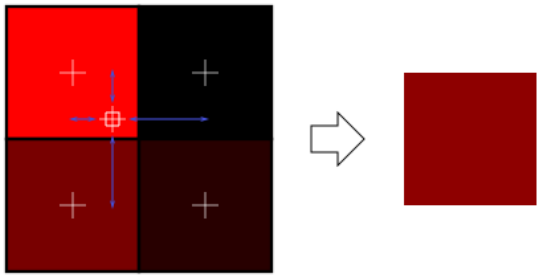


Filtering and mipmapping

There are several things we can do to improve texture quality.

Linear filtering

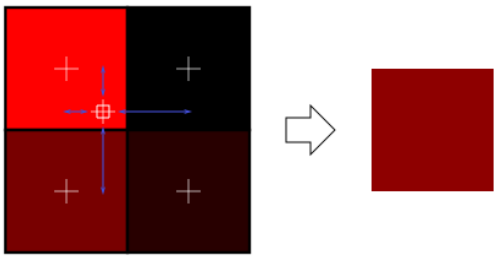
With linear filtering, texture() also looks at the other texels around, and mixes the colours according to the distance to each center. This avoids the hard edges seen above.



This is much better, and this is used a lot, but if you want very high quality you can also use anisotropic filtering, which is a bit slower.

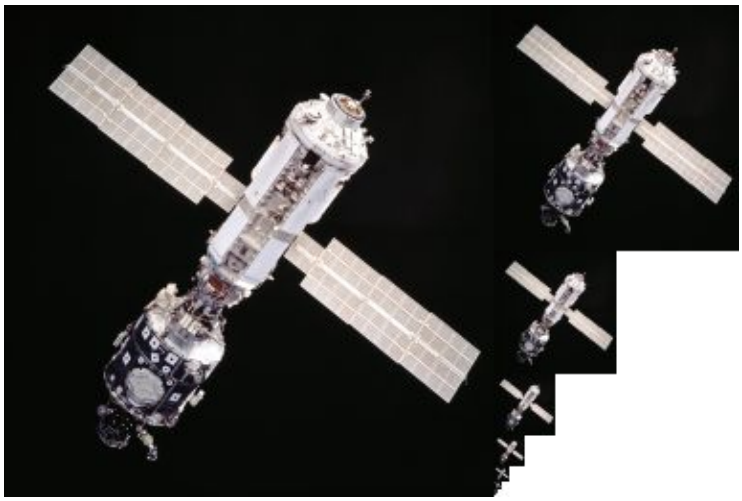
Anisotropic filtering

This one approximates the part of the image that is really seen through the fragment. For instance, if the following texture is seen from the side, and a little bit rotated, anisotropic filtering will compute the colour contained in the blue rectangle by taking a fixed number of samples (the “anisotropic level”) along its main direction.



Mipmaps

Both linear and anisotropic filtering have a problem. If the texture is seen from far away, mixing only 4 texels won’t be enough. Actually, if your 3D model is so far away than it takes only 1 fragment on screen, ALL the texels of the image should be averaged to produce the final color. This is obviously not done for performance reasons. Instead, we introduce MipMaps :



- At initialisation time, you scale down your image by 2, successively, until you only have a 1x1 image (which effectively is the average of all the texels in the image)
- When you draw a mesh, you select which mipmap is the more appropriate to use given how big the texel should be.
- You sample this mipmap with either nearest, linear or anisotropic filtering
- For additional quality, you can also sample two mipmaps and blend the results.

Luckily, all this is very simple to do, OpenGL does everything for us provided that you ask him nicely :

```
// When MAGnifying the image (no bigger mipmap available), use LINEAR filtering
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
// When MINifying the image, use a LINEAR blend of two mipmaps, each filtered LINEARLY too
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
// Generate mipmaps, by the way.
glGenerateMipmap(GL_TEXTURE_2D);
```

Exercises

Try to map your own texture (.bmp) to a cylinder and a cube.