# 521140S Computer Graphics Programming Assignment II

## General information

In this assignment, you will explore the OpenGL 3.1 implementation of basic drawing, projection, texturing and external object loading.

**Assignment should be finished alone.** Feel free to discuss about the problems but sharing code is not allowed.

## What to return

Return the finished code: CG_assignment2.py, the utils folder, the resources folder and a pdf report with the requested screenshots before the deadline.

## Tasks

In this assignment, you are expected to render a simple scene, where you implement the code to draw a cube, to render it with texture and projection.

1. Get the scene to work.
    a. Download example code package from Moodle.
    b. Find the *CG_assignment2.py* file. Run the code. If there is any problem, please refer to Tutorial 0. (0 p)

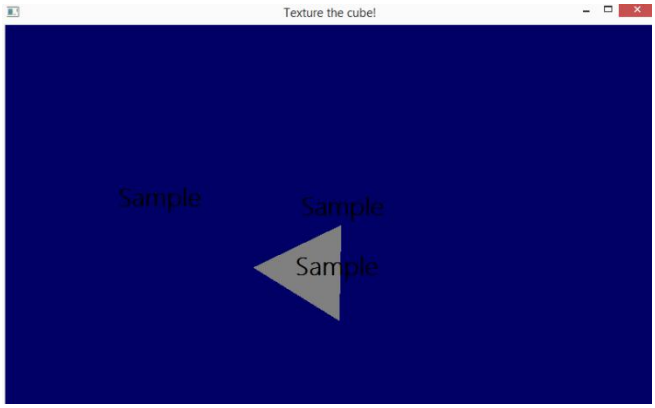


Figure 1

2. Adding camera movement and inputs

    From the previous assignment, fill in the code to the *mvp_controller.py*. This will allow you to move around and debug when defining the texture. (0 p)

3. Draw a cube with basic triangles.
    a. In the OpenGL world, no matter how complex the models are, they are all defined by basic triangles. The most common way to draw a model is to define the vertexes of a branch of triangles to ensemble the model like below.

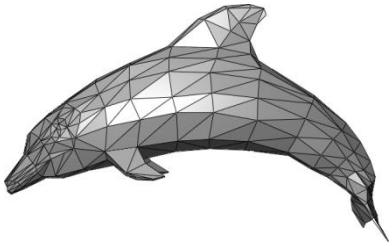

Figure 2

In the next step, we will use triangles to model a 2x2x2 cube centered at origin.

First, move to the beginning of the code *CG_assignment2.py*, find the definition of `vertex_buffer_data`, you can see that 12 pieces of triangles are defined by us, but only the first triangle's vertices are set by us (the vertices of the rest 11 triangles are all 0.0). Fill in the missing vertices.

After correctly setting the vertices, run the code and you will finally get a cube scene below. Take a screenshot in the report. (1 p)

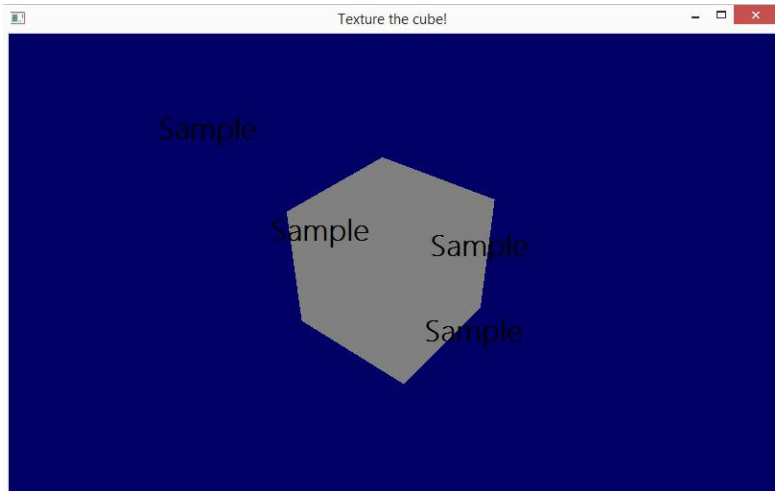


Figure 3

b. However, there is no surface on the cube. What we need to do is to map it with a texture.

The basic idea of mapping texture is to point the certain position of the texture to the corresponding vertex of the model.

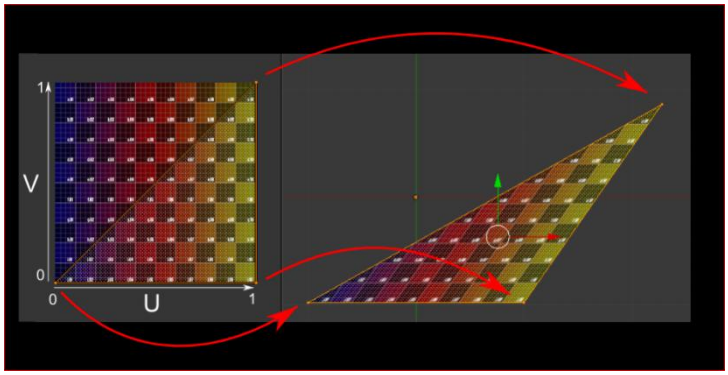


Figure 4

Move to the definition of `uv_buffer_data`, which is right next to the definition of `vertex_buffer_data`, in the comment, you can find that we define the uv coordinate for the first triangle as an example. Implement it according to the mapping rule below:

```
(u,v)->(x,y,z).
```

However, the texture does not show up on the cube as we have not buffered the texture data yet. In the `init_context_raw` function the data buffering has already been done for the **vertex data** as follows:

- A buffer is created for the vertex data using `glGenBuffers`
- The buffer is bound using `glBindBuffer`
- The data is buffered using `glBufferData`

In the def `draw` function the process continues as follows:
- Set the location and format of your data using `glEnableVertexAttribArray`
- Bind the data to the buffer using `glBindBuffer` before drawing
- Instructions of the data layout are given using `glVertexAttribPointer`, i.e., the location and the format of the data

Follow the same procedure for buffering the **texture data**. After this you will get one triangle mapped with a piece of texture. Take a screenshot of this. (2 p)

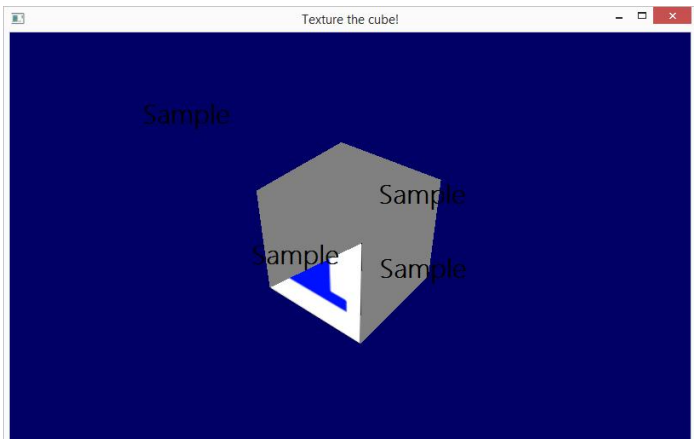


Figure 5

c. Fill in the missing data from `uv_buffer_data` to obtain the final cube: Run the code and take a screenshot of this. (2 p)

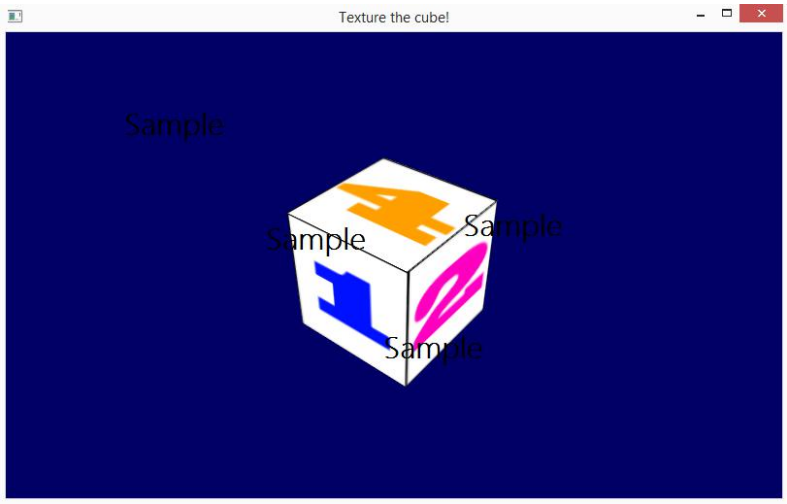


Figure 6

4. Load an external object.

As you can imagine, if we draw the models with raw triangles each time, it will take too much time and defining complex models becomes extremely difficult. So, let's try to load external objects that can be created with software like blender.

Here, we offer the utils called `ObjLoader` to help you to load objects. The methods how to call an external model is offered here:

```
texture = TextureLoader("resources/object/uvmap.png")
model = ObjLoader("resources/object/cube.obj").to_array_style()

vertex_buffer_data = model.vertexs
uv_buffer_data = model.texcoords
```

Now, move to the function `init_context_load`. Refer to the function of `init_context_raw` to complete the function `init_context_load`.

After you complete the function `init_context_load`, don't forget to uncomment the line `win.init_context_raw` into `win.init_context_load` in the `main` function. Run the code and you should have successfully loaded the object. Take a screenshot of this.

(2 p)

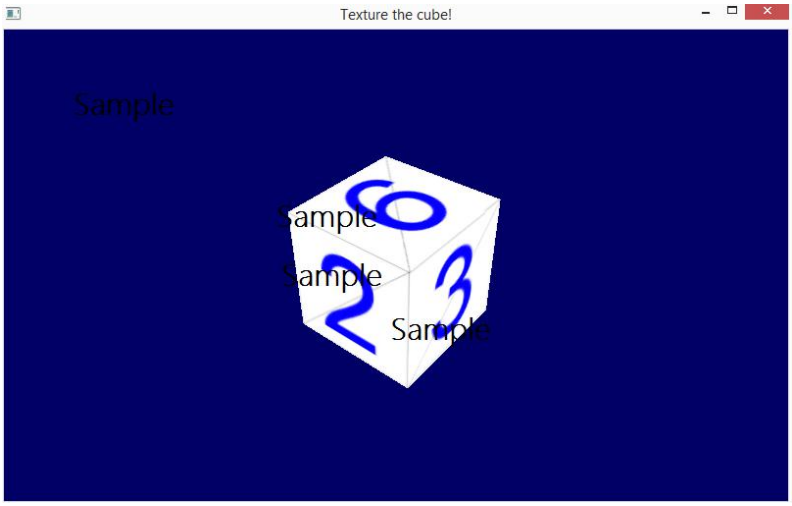


Figure 7

5. Shader programming.

   The source codes of the shader program are defined in two files *resources/vertex.glsl* and *resources/fragment.glsl*

   Note that the shaders use their own OpenGL Shading Language, which is very similar to c.

   We can modify the vertex data of the cube in the vertex file *vertex.glsl.* For example, we can scale the cube by multiply *position_offset* with some scale factor. The image below is the result of the scaling factor 0.5.

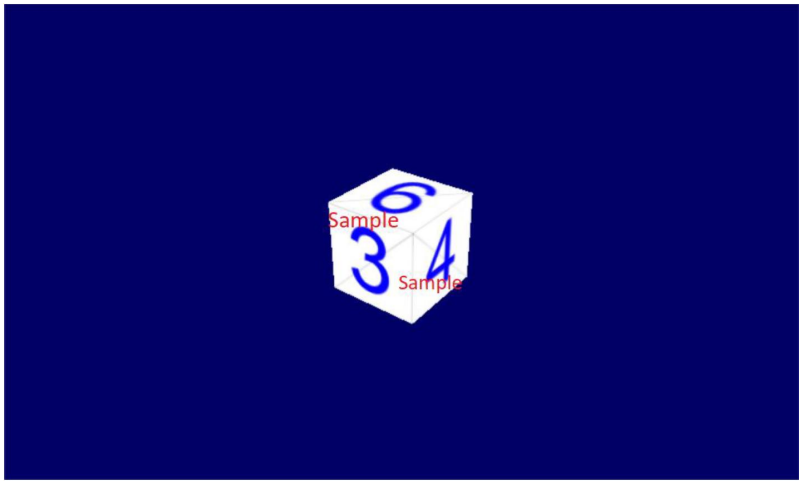


Figure 8

We can also adjust the texture color of the object in the fragment file *fragment.glsl.* The image below is the result of color inversion. Please note that the color value has been normalized to the range [0, 1].


Figure 9

Modify the source code of vertex and fragment shader. Take a screenshot of the change. You can implement other types of transformation for the vertex and texture. It is not restricted to only the scaling and color inversion. (1 p)