COSC363 Computer Graphics
Lab10: More OpenGL-4

This lab provides examples of shader based implementations of texture mapping and tessellation operations.

I. Cube.cpp:

1. The program `Cube.cpp` provides the code for displaying the mesh model of a cube.

   - The file `Cube.h` contains the array definitions specifying vertex coordinates, normal components, texture coordinates and polygon indices.

   - The vertex shader (`Cube.vert`) implements a simple lighting model using only the diffuse component. It also outputs the texture coordinates and the diffuse lighting term (***n•l***) to the fragment shader.

   - The fragment shader (`Cube.frag`) defines a uniform variable "tSampler1" of type `Sampler2D` (a texture type). The function `texture()` returns a colour value obtained by sampling the texture using the input texture coordianates. The fragment shader outputs this colour for each fragment, thus generating the display of a texture mapped primitive.

   - The program `Cube.cpp` loads the texture "Brick.tga" (Fig. 1(a)) and assigns it to texture unit 0. Using the function `glUniformi()` it assigns the same value 0 to the variable "tSampler1" in the fragment shader. The program generates the output of a texture mapped cube (Fig. 1(b)).

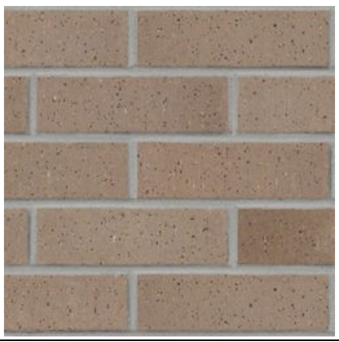	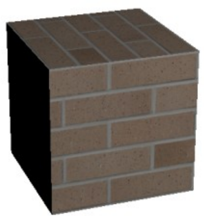	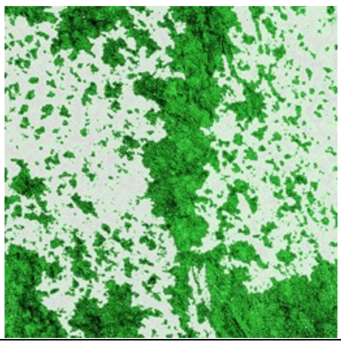

Fig. 1(a) Fig. 1(b) Fig. 1(c)

2. Modify the program `Cube.cpp` to load a second texture "Moss.tga" (Fig. 1(c)). Select texture unit 1 for this texture, and assign the value 1 to a

uniform variable “tSampler2”. Include statements for loading the texture and setting its filter parameters (see slide [9]-40). Add the new uniform variable “tSampler2” in the fragment shader. Modify the shader’s output by combining the colour values obtained from the two textures (see slide [9]-45). The output of multi-texturing should look similar to that given in Fig. 2.

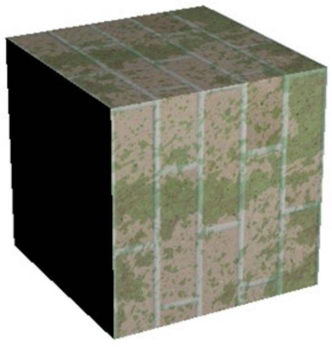

Fig. 2.

II. QuadPatches.cpp:

In this exercise, we will use tessellation shaders to model a curved surface segment in three-dimensional space. The program specifies the coordinates of 9 patch vertices in the array “patchVerts[]”. The relative positions of the patch vertices are shown in Fig. 3.

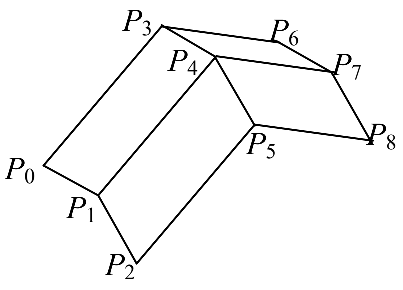

Fig. 3

Please note the following:

- Patches (primitives of type GL_PATCHES) contain only point information. Therefore only one VBO is needed to specify the data for a patch.
- The number of patch vertices must be specified using `glPatchParameteri()` function. The program includes this statement.
- When the tessellation shader stage is active, the `glDrawArrays()` function must specify the primitive type as GL_PATCHES (see display function).
- Patch vertices cannot be directly rendered. A patch is not a renderable primitive. Therefore the vertex shader (QuadPatches.vert) simply copies them to the output variable without transforming them to clip coordinates.

The vertex shader `QuadPatches.vert` has its simplest structure in the form of a pass-through shader. It directly outputs the patch vertices without any modification, to the next shader, the tessellation control shader.

The control shader `QuadPatches.cont` sets the outer tessellation levels to 6 and inner levels to 4. It also transfers the patch vertices received from the vertex shader directly to the evaluation shader.

The evaluation shader `QuadPatches.eval` processes the vertices of triangles generated by the primitive generator. The vertices are defined using tessellation coordinates (`gl_TessCoord.x, gl_TessCoord.y`). The evaluation shader repositions these vertices in three dimensional space by combining the patch vertices using blending functions. In the current form, this equation uses only the four corner vertices of the patch P_0, P_2, P_6, P_8 and bi-linear blending functions (see slide [10]-14), and produces a tessellated quad that passes through the four points (Fig. 4a). Please note that the evaluation shader outputs the repositioned vertices ("posn") in clip coordinates.

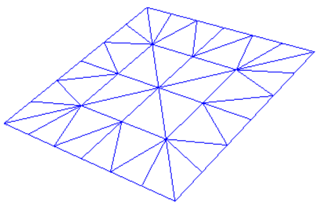
Fig. 4a

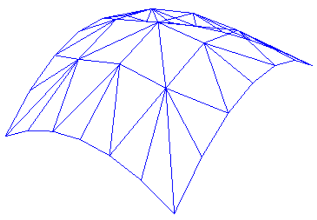
Fig. 4b

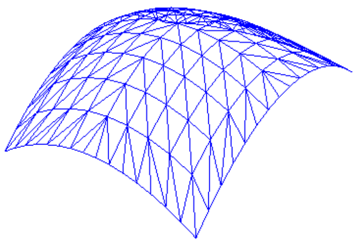
Fig. 4c

Modify the equation in the evaluation shader by using bi-quadratic blending functions and all 9 patch vertices as given on Slide [10]-17. The program should produce an output shown in Fig. 4b. The surface model can be rotated about the y-axis using the left and right arrow keys. Increase the outer tessellation level to 15 and the inner level to 10 to get an output shown in Fig. 4c. The models generated are bi-quadratic Bezier surfaces. The patch vertices act as control points for the surface. Experiment with variations in the positions of patch vertices and changes in tessellation levels.

Ref:

[9] COSC363 Lecture slides: Lec09_OpenGL4.pdf

[10] COSC363 Lecture slides: Lec10_Tessellation.pdf

III. Quiz-10

This is the last quiz of the course!

The quiz will remain open until **5pm, 4-June-2021**.