COSC363 Computer Graphics
Lab09: OpenGL-4 Basics

Aim:

In this lab, you will familiarize yourself with the structure of OpenGL-4 programs that use buffer objects and shaders (vertex and fragment shaders) for developing applications for the programmable pipeline.

I. Seashell.cpp:

The program contains the code for displaying the mesh model of a seashell. The structure of the application is shown below (Fig. 1).

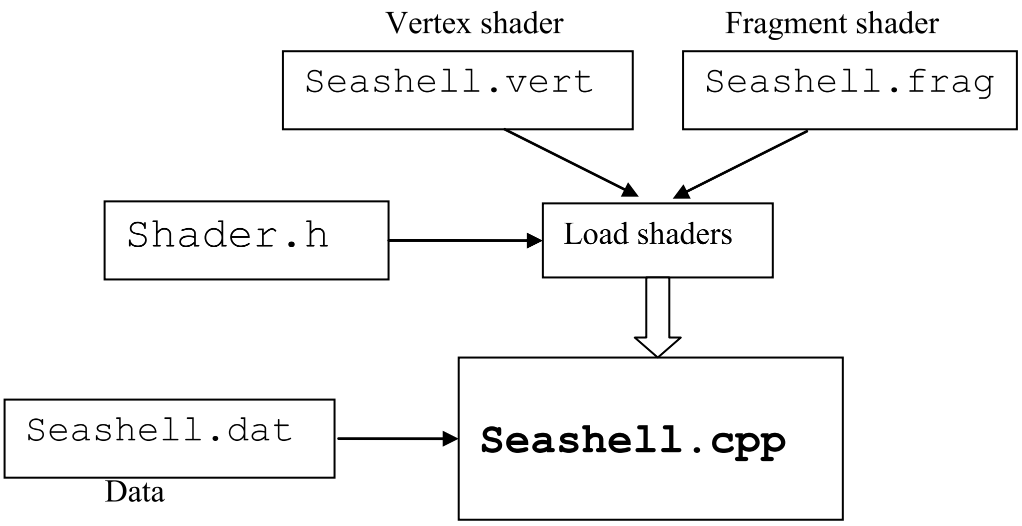


Fig. 1

- The "main()" function in `Seashell.cpp` initializes the rendering context with version OpenGL 4.1, and core profile.

- The "initialize()" function calls the "createShaderProg()" function of `shader.h` to load the shaders `Seashell.vert` and `Seashell.frag`. It also gets the locations of uniform variables defined in the vertex shader by calling "glGetUniformLocation()" function. "Uniform" variables are used to pass values of matrices, light's parameters etc. from the application (`Seashell.cpp`) to the shaders. Inside the shader code `Seashell.vert` , you will find a set of variables "mvMatrix", "mvpMatrix", "norMatrix" and "lightPos" declared with storage qualifier "uniform". These matrices are used within the shader for lighting calculations and transformations. Please note that any parameter that has a fixed value for the given scene (eg. camera parameters, view matirx, light's parameters) can be defined in the initialize() function. The vertex shader performs all lighting calculations in eye coordinates, and hence requires light's position in that frame.

- The "display()" function creates the matrices required for model, view and projection transformations, and passes the values to the vertex shader. The angle of rotation of the model is continuously incremented using a timer callback. The GLM function rotate() returns a 4x4 matrix corresponding to the rotational transformation. Note that this function requires the angle of rotation specified in radians. The model-view matrix is the product of the view matrix and the rotation matrix. The shader uses this matrix to transform all vertices into eye coordinate frame (for lighting calculations). Similarly, the model-view-projection matrix is the product of the projection matrix and the model-view matrix. This matrix is used by the shader to transform vertices into the clip coordinate space. The display of the model is generated by binding the VAO containing the models vertex data, and calling the "glDrawElements()" function.

- The vertex shader (`Seashell.vert`) contains code for lighting calculations and the transformation of vertex coordinates to the clip coordinate space (using the model-view-projection matrix).

- The fragment shader (`Seashell.frag`), in this example, is a pass-thru shader which receives the interpolated colour values in the variable "oColor" and directly outputs those values. The generated output is shown in Fig. 2.

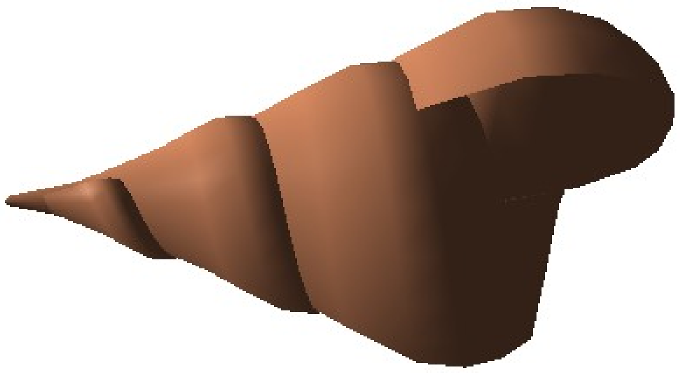
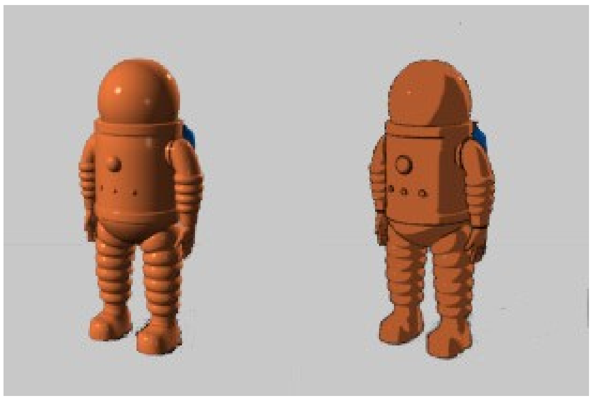


Fig. 2.

II. Non-Photorealistic Rendering:

Non-photorealistic rendering refers to the process of generating displays with expressive or artistic styles. This rendering paradigm is also known by other names such as toon-shading, cel-shading, sketch-based rendering etc. An example from Wikipedia (http://en.wikipedia.org/wiki/Non-photorealistic_rendering) is shown in Fig. 3.



A robot model rendered using Gouraud shading and 2-tone shading.

Source: Wikipedia

Fig. 3.

1. A two-tone shading of a model as seen in the above figure is generated by replacing the continuous variation of shades on a surface with just two completely flat colours. This is done by using a threshold for the diffuse term ***n•l***. (the dot product between the light source vector and the normal vector). If we ignore specular reflections, the brightness at a vertex is proportional to this term. In the following, we will create a two-tone shading of the seashell model.

2. For two-tone shading, we require the value of ***n•l*** for each fragment. This value is computed in the vertex shader and stored in the variable `diffTerm`. Declare this variable as an "out" variable inside the vertex shader, and also as an "in" variable inside the fragment shader. The variable declarations must be placed outside the main() function (a local variable cannot be declared with a storage qualifier "out") The interpolated values of ***n•l*** will then become available in the fragment shader. Modify `Seashell.frag` to produce a two-tone shading of the model based on the following rule:
   if ***n•l*** is less than 0.2, output a dark shade of a colour, otherwise output a lighter shade, as shown in Fig. 4.

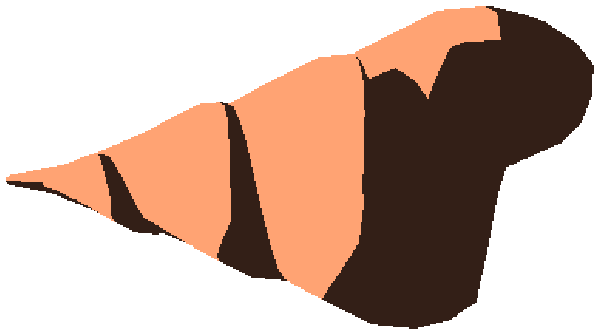


Fig. 4.

3. To further enhance the quality of two-tone shading, it is necessary to highlight the silhouette edges of the model. Such edges can be clearly seen in the robot model in Fig. 3. Silhouette edges correspond to boundary regions between a

front facing and a back facing polygon on a model. A fragment belongs to a silhouette edge if $\boldsymbol{n} \bullet \boldsymbol{v} = 0$, where $\boldsymbol{v}$ is the view vector (Fig. 5).

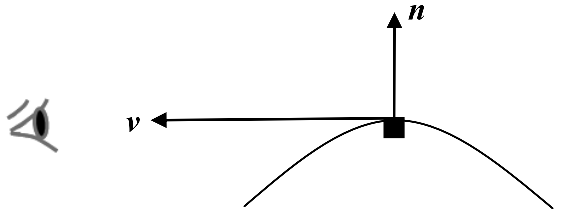


Fig. 5.

4. Compute the value of $\boldsymbol{n} \bullet \boldsymbol{v}$ inside the vertex shader and pass the value to the fragment shader. Modify the output of the fragment shader as follows:

   If $|\boldsymbol{n} \bullet \boldsymbol{v}| < 0.2$, output black colour.

   The above condition, if properly implemented, will cause some of the silhouette edges to become visible on the model as shown in Fig. 6.

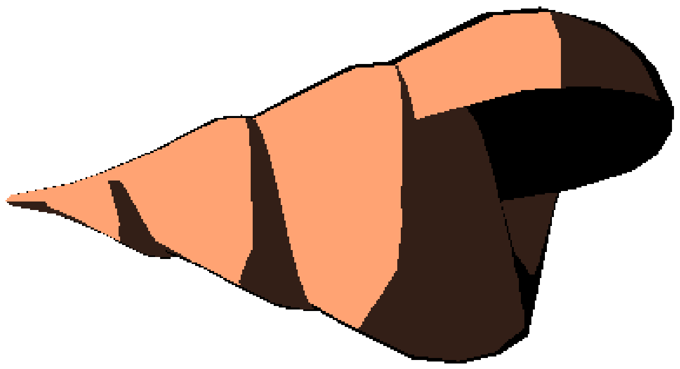


Fig. 6.

III. 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 ($\boldsymbol{n} \bullet \boldsymbol{l}$) 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. 7(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. 7(c)).

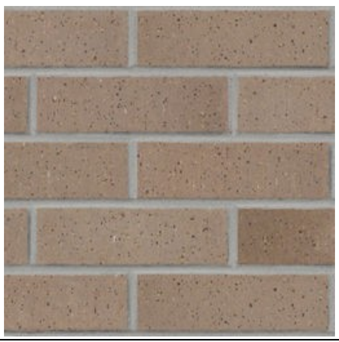
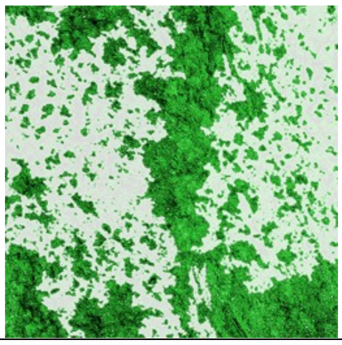
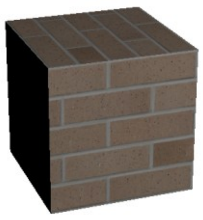
		

Fig. 7(a) Fig. 7(b) Fig. 7(c)

2. Modify the program `Cube.cpp` to load a second texture "Moss.tga" (Fig. 7(b)). Select texture unit 1 for this texture, and assign the value 1 to a uniform variable "tSampler2" . Add this new uniform variable "tSampler2" in the fragment shader. Modify the shader's output by combining the colour values obtained from the two textures (see Lec09-Slide 50). The output of multi-texturing should look similar to that given in Fig. 8.

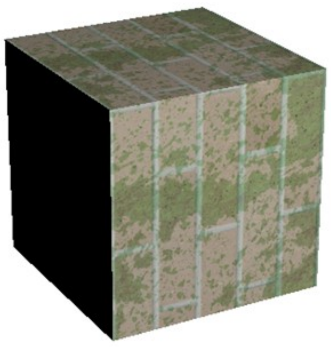


Fig. 8.

IV. Quiz-09

The quiz will remain open until **5pm, 22-May-2020**.