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. TorusDraw.cpp:

The program `TorusDraw.cpp` provides the code for displaying the mesh model of a torus. The structure of the application is shown below (Fig. (a)).

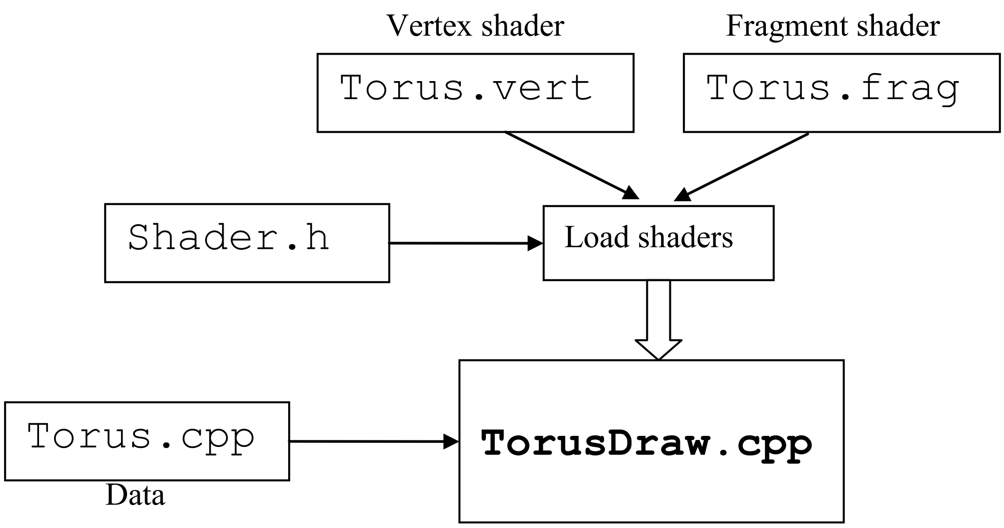


Fig. (a)

- The “main()” function in `TorusDraw.cpp` initializes the rendering context with version OpenGL 4.2, and core profile.

- The “initialize()” function calls the “createShaderProg()” function of `shader.h` to load the shaders `Torus.vert` and `Torus.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 (`TorusDraw.cpp`) to shaders. Inside the shader code `Torus.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, light's parameters) can be defined in the initialize() function.

1. Inside the "initialize()" function, the projection matrix is defined using the GLM's perspective() function which has a parameter list similar to that of gluPerspective() function. Similarly, define the view matrix using GLM's lookAt function which takes three parameters of type glm::vec3. Specify the eye position at (0,0, 25) and the origin as the camera's look point. Use this matrix to transform light's position from world coordinates to eye coordinates ($L_{eye} = V\ L_{world}$). This value is then passed to the shader. The 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, passes the values to the vertex shader, and calls the "render()" function of the Torus class. The angle of rotation of the torus model is continuously incremented using a timer callback. The GLM function rotate() returns a 4x4 matrix corresponding to a rotational transformation.

  Define the model-view matrix as the product of the view matrix and the rotation matrix. The shader uses this matrix to transform all vertices into eye coordinate frame. Similarly, define the model-view-projection matrix as the product of the projection matrix and the model-view matrix.

- The constructor of the "Torus" class (see `Torus.cpp`) generates the vertex buffer objects and the vertex array object for the mesh model. It also includes a `render()` function to generate the display of the model using the reference to the vertex array object.

- The vertex shader (`Torus.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 (`Torus.frag`) outputs a single colour value for every fragment, and causes the torus to appear as shown in Fig. (b).

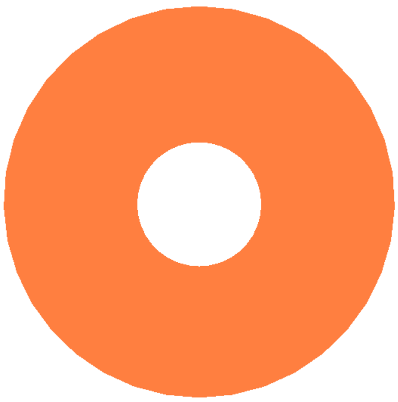


Fig. (b)

2. Inside the vertex shader (`Torus.vert`), the result of the lighting calculations (sum of ambient, diffuse and specular reflections at a vertex) is stored in the

variable "theColour". Please note that this variable is declared with a storage qualifier "out". The fragment shader (`Torus.frag`) has the declaration of the same variable with storage qualifer “in”. This is the mechanism of passing values from the vertex shader to the fragment shader. Assign this value to the output of the fragment shader (gl_FragColor) to get the output shown in Fig.(c).

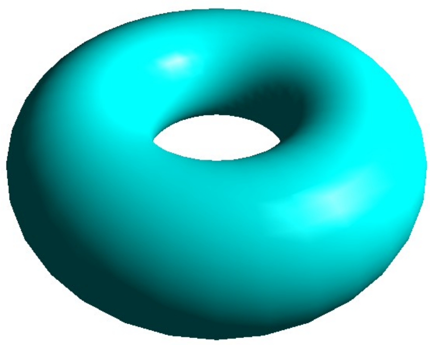


Fig. (c)

3. The fragment shader has a built-in variable `gl_FragCoord`. Its z-component, `gl_FragCoord.z` gives the depth of the fragment in the range [0, 1]. Modify the fragment shader to output a colour value whose r, g, b components all have the depth value of the fragment. You will then get the depth-map of the torus, with regions close to the near-plane having a dark colour and regions farther away from the camera having lighter colours (Fig. (d)).

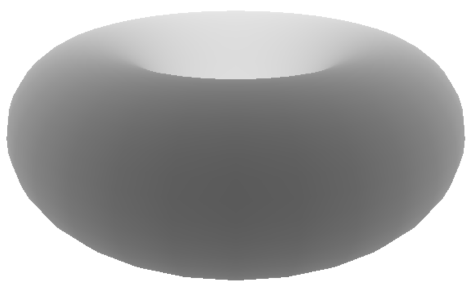
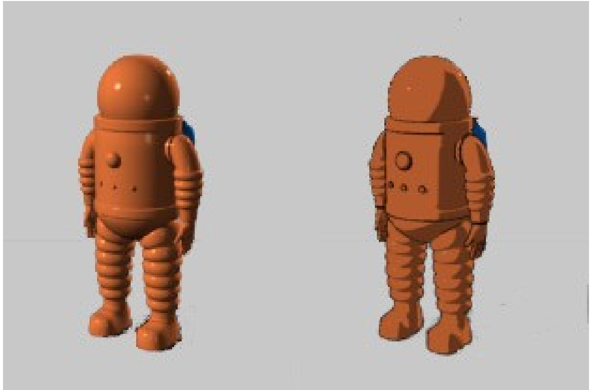


Fig. (d).

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. (e).



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

Source: Wikipedia

Fig. (e).

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 colour values. This is done by using a threshold for the diffuse term ***n•l***. (Note: the brightness is proportional to this term). In the following, we will create a two-tone shading of the torus model.

2. For two-tone shading, we require the value of ***n•l*** for each fragment. This value is computed in the vertex shader. Declare this variable as an "out" variable inside the vertex shader, and also as an "in" variable inside the fragment shader. The interpolated values of ***n•l*** will then become available in the fragment shader. Modify `Torus.frag` to output a colour based on the following rule:
   if ***n•l*** is less than 0.1, output one colour, otherwise output another colour.
   A sample output is shown below in Fig. (f):

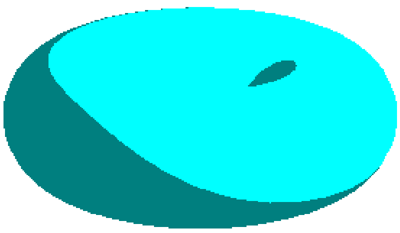


Fig. (f)

3. To further enhance the quality of the 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. (e). A fragment belongs to a silhouette edge if $\boldsymbol{n} \bullet \boldsymbol{v} = 0$, where $\boldsymbol{v}$ is the view vector (Fig. (g)).

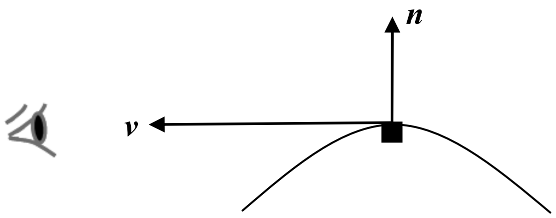


Fig.(g)

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 torus model as shown in Fig. (h).

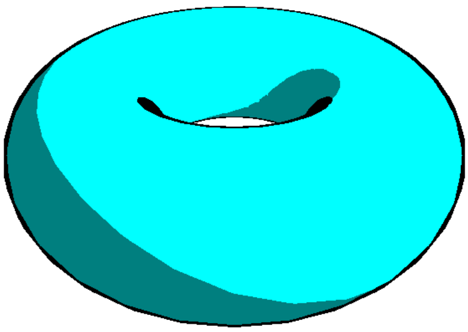


Fig. (h)

III. Quiz-09

The quiz will remain open until **5pm, 24-May-2019**.