CSE 328 PROGRAMMING ASSIGNMENT FOUR

This programming project's goal is to concentrate on applying 3D graphics techniques to edit, manipulate, and display geometric shapes in common use such as cube, tetrahedron, octahedron, sphere, cylinder, cone, ellipsoid, torus, etc.

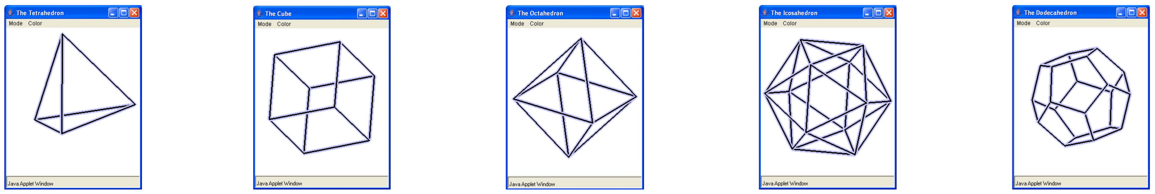
Write a program that implements the drawing of polyhedral objects and quadrics primitives. Your program should be able to: (1) Display wireframe objects; (2) Display flat-shaded solid objects; (3) Display smoothly-shaded solid objects; and (4) Allow users to interactively translate, rotate with respect to any points/axes, scale/zoom, shear, and reflect with respect to any user-specified plane the displayed objects. In particular, the entire project must consist of the following components as far as the system functionalities are concerned.

Please note that you are still required to use OpenGL in the **core-profile** mode (modern OpenGL). Deprecated functions in immediate mode (legacy OpenGL) are **not** permitted.

Before you continue, please review the course materials and the handout documents carefully. If your tessellation shader is not working, you could try to disable the 3D Acceleration feature of your VMWare host.

Please refer to TA Help Page for general requirements. Please note that all deadlines are **hard**. To avoid late submissions due to network issues, do **not** submit at the last minute! Please also note that your credits for this programming assignment will be scaled by 50% (i.e., from $300 + 60$ to $150 + 30$) on Brightspace to match the overall grading schemes.

1 Simple Polyhedral Objects (15 points)



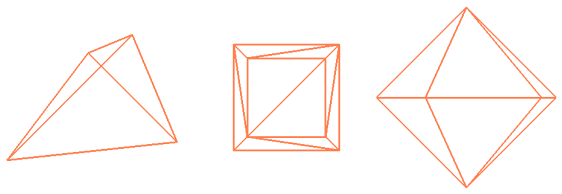
(a) Tetrahedron (b) Cube (c) Octahedron (d) Icosahedron (e) Dodecahedron

Figure 1: Five commonly-used polyhedral objects (also called *platonic solids*).

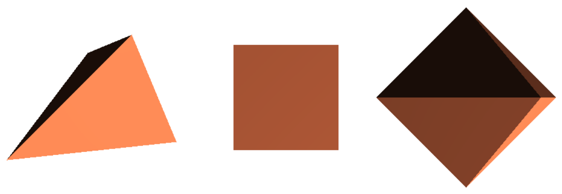
Write your program to display a tetrahedron, a cube, and an octahedron in one scene. Please refer to Figure 1(a-c) and the program template for the details of each object (the coordinates of their vertices, in the model coordinate system.) The tetrahedron should be further translated towards negative x direction by 2 units in the world coordinate system, and the octahedron should be translated towards positive x direction by 2 units. The user will press key `F1` to enter this part.

Your program should offer users the following global functionalities: (1) Display modes; (2) Transformations; (3) Camera functionalities. Please note that these global functionalities should be made available for **the entire program** (i.e., to all parts), and they are detailed as follows:

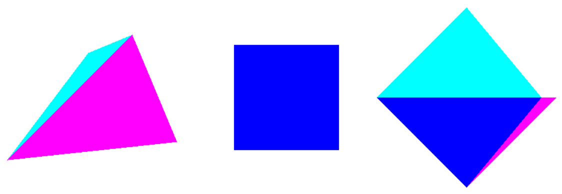
Display Modes. The user should be able to press the following keys to switch to: `WIREFRAME` mode (key `1`), which displays wireframe objects; `FLAT` mode (key `2`), which displays flat-shaded solid objects; `SMOOTH` mode (key `4`), which displays smoothly-shaded solid objects. (Please note that key `3` is reserved for the BONUS part.) Please refer to the lecture slides and the attached materials for the implementation details of each mode as well as the Phong shading model. Please also refer to Figure 2 for the expected visual effect of each mode.



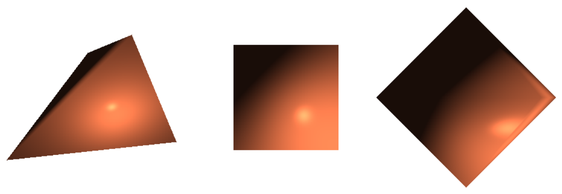
(a) `WIREFRAME` mode: wireframe only



(b) `FLAT` mode: flat-shaded objects



(c) `NORMAL` mode (BONUS)



(d) `SMOOTH` mode: smoothly-shaded objects

Figure 2: Expected visual effect of different display modes.

Transformations. You should also allow users to interactively apply 3D transformations on the displayed objects. The user should be able to left-click the mouse to select a 3D object (upon selection, you should display the wireframe of the axis-aligned bounding box of the object), and right-click the mouse to deselect an object (and clear the bounding box upon deselection). Then, the user should be able to apply any of the following transformations on the selected object:

(1) *Translate* the selected object forward, backward, toward the left, toward the right, upward and downward (with respect to your current viewpoint) with the `W`, `S`, `A`, `D`, `UP` and `DOWN` keys while pressing the `T` key;

(2) *Rotate* the object with respect to a user-specified axis by scrolling the middle button while pressing the `R` key;

(3) *Scale* the object with respect to its geometric center by scrolling the middle button while pressing the `Y` key;

(4) *Shear* the object by dragging the mouse with the specified parameters by pressing the `U` key;

(5) *Reflect* the object with respect to a user-specified plane by clicking the middle button.

The transformation parameters are to be read *dynamically* from the first line of the configuration file `etc/config.txt`. (Please note that the second line is reserved for other parts.) The first line contains 12 space-separated floating point numbers, namely $a_1, a_2, \cdots, a_{12}$. For *rotation*, the rotation axis passes through the geometric center of the object and its direction is is specified by a 3D vector (a_1, a_2, a_3); for *shear*, the shear parameters for the x, y and z axes are specified separately by a_4, a_5, and a_6; for *reflection*, the reflection plane is specified by two 3D vectors (a_7, a_8, a_9) and (a_{10}, a_{11}, a_{12}), with all these values in the world space. Please note that you are **required** to implement these transformations with transformation matrices (OpenGL shader `uniforms`).

Camera Functionalities. You should also offer users with the following camera functionalities: (1) Press key `X` to show or hide the positive x, y, z axes, with the x-axis displayed in red, the y-axis in green, and the z-axis in blue. Each axis starts from the origin and has a length of 10; (2) Press keys

`W`, `S`, `A`, `D`, `UP`, `DOWN`, or drag or scroll the mouse to adjust the camera. Please note that for simplicity, the camera functionalities are *already implemented* in the program template.

2 Subdivision (165 points)

The major concern of this part is the display and (manual) subdivision of polyhedral objects. Please refer to Figure 3 for the visual effect of subdividing an icosahedron. For this part, you are **not** allowed to use tessellation shaders.

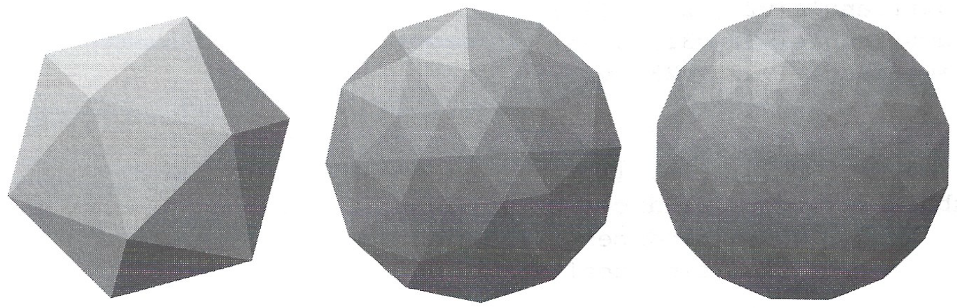

(a) Original icosahedron (b) Subdivided once (c) Subdivided twice

Figure 3: Subdividing an icosahedron to improve the polyhedral approximation to a sphere.

2.1 Icosahedron (35 points)

Write a program to display an icosahedron of the unit sphere (i.e., a 20-triangle representation of the unit sphere) as shown in Figure 3(a). Again, please refer to the program template for the details of an icosahedron. (The program template has hard-coded one unit icosahedron centered at the origin of the world coordinate system.) The user will press key `F2` to enter this part.

Besides the functionalities as specified in part 1, you should also implement the *subdivision* of an icosahedron. The user should be able to press key + to subdivide the icosahedron into an improved 80-triangle approximation of the sphere as shwon in Figure 3(b). The subdivision could be stacked, that is, the user could press key + again to further subdivide the 80-triangle approximation of the sphere into a 320-triangle approximation, as shwon in Figure 3(c). Please refer to the following code for a sample implementation of subdividing a triangular facet on a *unit icosahedron* (an icosahedron with all its vertices sampled uniformly from the *unit sphere* $x^2 + y^2 + z^2 = 1$):

```
// Subdivide an icosahedron to approximate a unit sphere.
// Takes as input a triangular facet on a unit icosahedron
// and subdivide it into four new triangular facets.
void subdivide(glm::vec3 v1, glm::vec3 v2, glm::vec3 v3)
{
    // Sample new vertices so that
    // all vertices lie uniformly on the unit sphere.
    glm::vec3 v12 = glm::normalize((v1 + v2) / 2.0f);
    glm::vec3 v23 = glm::normalize((v2 + v3) / 2.0f);
    glm::vec3 v31 = glm::normalize((v3 + v1) / 2.0f);

    addFacet(v1, v12, v31);
    addFacet(v2, v23, v12);
    addFacet(v3, v31, v23);
    addFacet(v12, v23, v31);
    removeFacet(v1, v2, v3);
}
```

Please note that to press key +, you actually press key = while pressing key `SHIFT`.

2.2 Ellipsoid (30 points)

Write a program to display a 20-triangle approximation of an ellipsoid. The user will press key `F3` to enter this part. In analogy to the icosahedron, you should also support all display modes as well as subdivisions (the 80-triangle and 320-triangle approximation for the same ellipsoid). Again, you could put the ellipsoid at the origin of the world coordinate system.

2.3 Dodecahedron (20 points)

Write a program to display a dodecahedron. Again, please refer to the program template for the details of a dodecahedron. You could put the dodecahedron at the origin of the world coordinate system. The user will press key `F4` to enter this part. You should also support all display modes as well as subdivisions.

2.4 Torus (80 points)

Based on the idea of sampling parametric equations and the strategy of subdivision, assemble and subdivide a torus. The user will press key `F5` to enter this part.

Torus Display (20 points). Display a torus by sampling vertices using its parametric equation, Again, you could put the torus in any size centered at the origin of the world coordinate system. Although you are free to select the number of polygons to be used to approximate the torus, the instructor suggests that you sample a continuous torus using at least $15 \times 15 = 225$ different points in order to better approximate a torus. Note that, the display quality of a torus at this stage should be reasonably good!

Torus Subdivision (60 points). Once again, based on the subdivision strategy detailed above, generalize your program to handle the refinement of a torus, for example, 15×15 now becomes 30×30, at this stage, you could either directly work on the quadrilaterals or split one quadrilateral along one main diagonal into two triangles. Either strategy suffices here! After that, perform one more level of refinement for a torus so that now the resolution becomes 60×60.

3 Tessellation (40 points)

The major topic of this part is to display quadric primitives with OpenGL tessellation shader. You will learn to assemble quadric surface meshes from their parametric equations with the help of OpenGL tessellation shader. For this part, you are **not** allowed to display these shapes by manually subdividing polyhedral objects.

Hint: For quadric surface meshes which are assembled from their parametric equations, you could just derive vertex normal from parametric equations. You are **not** required to handle face normal (which is used in the `FLAT` display mode). That is, for this part, the `FLAT` mode could look the same as the `SMOOTH` mode (without sharp edges.)

3.1 Simple Quadric Primitives (20 points)

Carefully study the sample program `OpenGLTessellationSphere` (available at the bottom of the TA Help Page as part of the Sample Code for the Lecture on PA3) and pay special attention to the drawing instructions of parametric surfaces. Display a sphere, a cylinder, and a cone with OpenGL tessellation shader. You could put these three objects in any size and anywhere in the scene, as long as all of them are completely visible (i.e., **not** occluded by each other) upon switching to this part. The user will press key `F6` to enter this part.

Meanwhile, you will have to be more careful about how you are going to deal with these special cases, especially cylinders and cones, because these shapes should be considered to be *closed* to actually represent solid objects.

3.2 Superquadrics (20 points)

You could also try to generalize the previous parts to display any superquadrics. The user will press key `F7` to enter this part. Again, you could put the superquadric in any size centered at the origin of the world coordinate system. The parameters of the super-quadrics should be read *dynamically* from the second line of the configuration file `etc/config.txt`. You could determine the format of this line by yourself, but please detail it in your `README` file.

4 Flight Simulation (80 points)

Add a small-scale city that comprises all the urban structures supported by your shape repository from the previous parts and then fly over your universe (virtual world) that you have just created (including a few urban structures, e.g., skyscrapers, domes, stadiums, and modern architecture). You are welcome to use NYC as a possible template to arrange their spatial distributions in order to imagine a few interesting landmarks, but you should utilize all the shapes from the previous parts. In the interest of time, 8-12 urban structures with detailed geometry and shapes suffice for this purpose. There are many methods for modeling the above-suggested objects. For example, one suggestion would be: Modeling the stadium using a truncated cone, modeling the dome using a truncated sphere, and modeling the tower and high-rise buildings using a set of blocks (refer to Figure 4.) Please note that this is ONLY an illustrative figure, you are welcome to arrange the spatial distribution of your universe using your own imagination.

As far as the flight simulation is concerned, there is a synthetic camera installed in the cockpit of the plane. The plane is allowed to do either a *vertical loop* or a *horizontal loop* above your universe. You should make use of an object hierarchy for rendering your universe (i.e., make use of the matrix stack).

The user will press key `F8` to enter this part. The user should be able to press key `H` for a horizontal loop, or press key `V` for a vertical loop. When the loop is completed, the camera should go back to its original position (its position, look-at direction, etc. before the loop.) Note that, you should also switch between different *display modes* (in order to speed up your flight simulation).

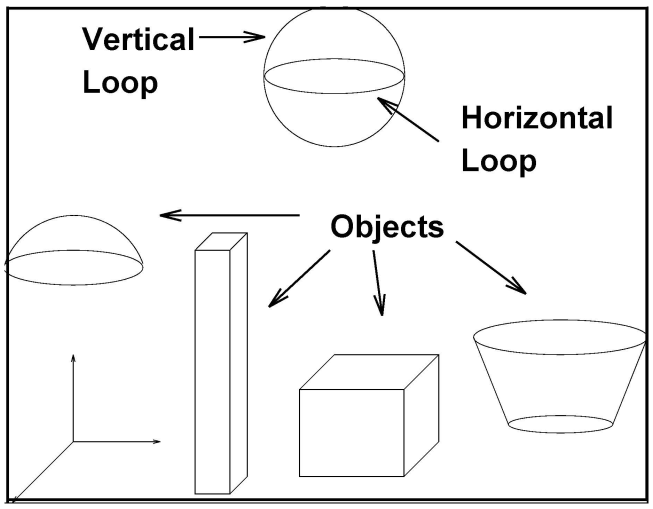


Figure 4: Flight simulation over your universe that comprises a small-scale city with 8-12 urban structures.

5 BONUS (60 points)

For the bonus part, you are welcome to explore different options. Meanwhile, the course instructor would like to suggest many possible extensions and generalizations.

Normal Display Mode. As a concrete example, you could try to display flat-shaded solid objects by mapping the normal $\mathbf{n} = (n_x, n_y, n_z)$ of each face to (R, G, B) components respectively (i.e., n_x to R, n_y to G, and n_z to B), as shown in Figure 2 (c). For consistency, the user would like to switch to this `NORMAL` mode by pressing key `3`, in addition to other display modes.

Other Possibilities. Another constructive suggestion, if you wish, you could explore the Internet to understand the current state-of-the-art methods for different modeling and design methods, and you are welcome to follow either existing ideas or explore your own ideas.

As far as shape modeling is concerned, you might provide global and/or local editing tools to afford the users better control over the modeled urban structures. For example, you could deform them either globally or locally. Alternatively, you could add a terrain (e.g., mountains, lakes, rivers, etc.), a theme park, and/or a road system, in order to enhance the overall cityscape.

As far as urban structures are concerned, you are also welcome to explore different methods from what I had already suggested above. For example, you could: (1) Add more buildings into your universe; (2) Make your objects more photo-realistic, etc.

As far as your enhanced flight experiences, you might consider allowing your self-defined camera to follow an arbitrary spline specified by users, etc.

At this point, since these are the BONUS parts, you are strongly encouraged to explore your own ideas. However, you are welcome to discuss with the course instructor and the TA about your ideas before you go ahead with your implementation for this part. Please note that we are rewarding up to **30 points** for this part, so a reasonable amount of time investment is necessary. HAVE FUN AND ENJOY WORKING ON THIS ASSIGNMENT!