Computer Graphics, 2020

Assignment 2: Planet in Space

First, refer to the general instruction for assignment submission, provided separately on the course web. If somethings do not meet the general requirements, you will lose corresponding points or the entire credits for this assignment.

1. Objective

You already learned how to draw a 2D circle made by a set of triangles. The goal of this assignment is to extend the concept of 2D geometric modeling to 3D geometric modeling. In particular, you need to draw a 3D colored sphere, one of the standard 3D primitives, on the screen. This assignment will be used for the next assignments (*solar systems*), which requires to draw more (moving) planets.

2. Triangular Approximation of Sphere

In general, a sphere is represented by its origin and radius. However, such implicit surfaces are not directly supported in GPU, and thus, we need to convert it to a finite set of triangular primitives. A common way of doing this is to approximate the sphere by subdividing it both in the longitude and latitude (see Figure 1). In your implementation, it would be better to subdivide it to finer surfaces for a smoother boundary; e.g., 72 edges in longitude and 36 edges in latitude.

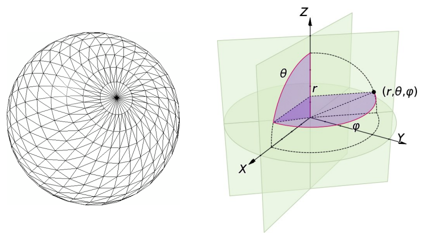


Figure 1: A triangular approximation of a sphere (left) and the definition of spherical coordinates in RHS coordinate system (right).

Finding Cartesian coordinate $P(x, y, z)$ of each vertex can be easily done by converting those in the spherical coordinate system (see Figure 1). Given the origin $O = (0, 0, 0)$ and a radius r, P can be

$$P(x, y, z) = (r \sin\theta \cos\varphi, r \sin\theta \sin\varphi, r \cos\theta), \tag{1}$$

where $\varphi \in [0, 2\pi]$ and $\theta \in [0, \pi]$ represent angles in longitude and latitude. In your shader program, you may need to use 4D (homogeneous) positions for `gl_Position`; in the case, use 1 for the last 4-th component (this will be covered later in the transformation lecture).

After defining the array of vertices, you need to connect them to form a set of triangles, which will be stored as an index buffer. Pay attention to reflect their topology correctly. In particular, the vertices should be connected in the counter-clockwise order.

3. Normal Vectors and Texture Coordinates

The vertex definition also requires to define normal vectors and texture coordinates. Getting the normal vector $N(x, y, z)$ is very simple. Just normalize your position; or just omit the radius term in the position definition as below.

$$N(x, y, z) = (\sin\theta \cos\varphi, \sin\theta \sin\varphi, \cos\theta), \tag{2}$$

Even the texture coordinates T are simple.

$$T(x, y) = (\varphi/2\pi, 1 - \theta/\pi), \tag{3}$$

where $T_x \in [0, 1]$ and $T_y \in [0, 1]$. For the moment, we will not care about these definitions, but they will be explained later and be used for the next assignments. Please verify your implementation with the example results shown in Figure 3.

4. World Positions to Canonical View Volume

Recall that the default viewing volume is provided with $[-1, 1]^3$ for the x, y, and z axes in terms of LHS coordinate system (see Figure 2 for comparison of LHS and RHS conventions). Also, the default OpenGL camera is located at (0,0,0) and is directed towards positive z-axis (0,0,1).

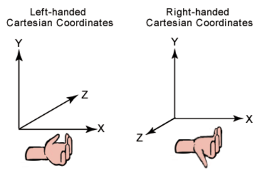

Left-handed Cartesian Coordinates
Right-handed Cartesian Coordinates
Y
Z
X
Y
X
Z

Figure 2: Comparison of LHS and RHS coordinate system. Note that the canonical volume of OpenGL uses the LHS convention; its z axis goes farther from the eye.

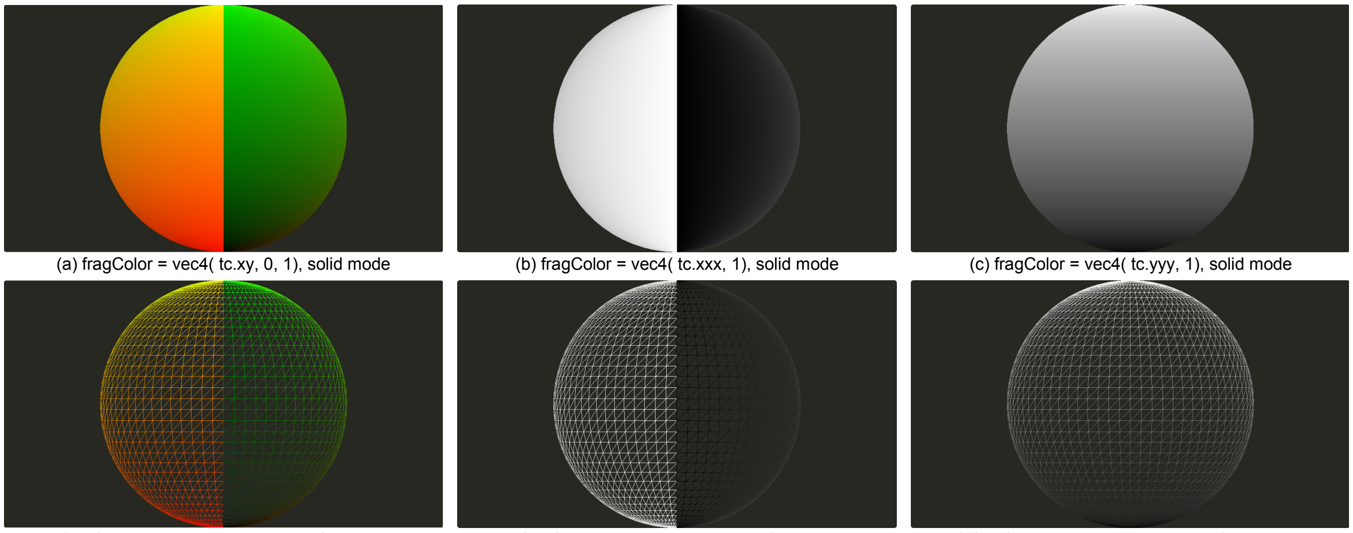
Since Eq. (3) defines the positions in the world coordinate system (with the RHS convention), we need to convert the RHS convention to the LHS convention, and also, need to rotate the view along y-axis.

Such a process can be automatically handled by view and projection matrices, but you did not learn them (they will be convered in viewing and projection lectures). In this assignment, we will not care about them, but sill need to a proper camera configuration. So, all you have to do here is just to use the matrix below to apply view transformation and projection together.

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4}$$

The matrix in Equation (4) will locate the camera at (1,0,0) toward (0,0,0). Also, the vertex definition will be normalized to the canonical view volume. Again, you here do not have to understand how the matrix works. In your C++ program, you can set the matrix and pass it to a uniform variable as below.

```
void update()
{
    mat4 view_projection_matrix =
        {0,1,0,0,0,0,1,0,-1,0,0,1,0,0,0,1};
    uloc=glGetUniformLocation(program,"view_projection_matrix");
    glUniformMatrix4fv(uloc,1,GL_TRUE,view_projection_matrix);
}
```



(a) fragColor = vec4(tc.xy, 0, 1), solid mode
(b) fragColor = vec4(tc.xxx, 1), solid mode
(c) fragColor = vec4(tc.yyy, 1), solid mode
(d) fragColor = vec4(tc.xy, 0, 1), wireframe mode
(e) fragColor = vec4(tc.xxx, 1), wireframe mode
(f) fragColor = vec4(tc.yyy, 1), wireframe mode

Figure 3: Example spheres drawn with three visualization schemes and two rendering modes.

5. Mandatory Requirements

What are listed below are mandatory. So, if anything is missing, you lose 50 pt for each.

- Use the index buffering.
- Enable back-face culling not to show the backfacing faces.
- Your sphere should not be an *ellipsoid*, even when the window is being resized.
- Do not update the vertex buffer (or vertex array object) for every single frame. Initialize the vertex buffer (vertex array object) only once (e.g., in `user_init()`)
- Your program needs to be toggled using 'W' key between wireframe and solid modes to examine the shape of the triangular approximation. This is because the sphere will be look like a circle, but the triangular structure should be different from the circle.

6. Implementation and Requirements

You may start from "cgcirc" or "transform" projects, already distributed on the course web. Precise requirements are as follows.

- Initialize the window size whose aspect ratio (width/height) is $16/9$ (e.g., 1280×720).
- Create vertex and index buffers of a sphere and locate them correctly. Set the origin and radius as (0,0,0) and 1, respectively. If positions, orientation, or texture coordinates are configured wrong, you lose 15 pt for each (45 pt).
- The sphere needs to be colored in terms of texture coordinates; i.e., the color needs to be (tc.xy,0,1) by default (15 pt). The code below is excerpt from the shader code on texture coordinates.
- Pressing 'D' key cyclically toggles among (tc.xy,0,1), (tc.xxx,1), and (tc.yyy,1). Refer to the examples and the sample program binary (10 pt).
- The sphere should be rotated as the sample program does. The sphere rotates at program launch, and pressing 'R' key stops rotation. Then, pressing 'R' again resumes rotation (20 pt).

The following shader code shows how to handle the texture coordinates in your shader program.

```
// vertex shader
in  vec2 texcoord;
out vec2 tc;
void main()
{
    ...
    tc = texcoord;
}

// fragment shader
in  vec2 tc;
out vec4 fragColor;
void main()
{
    ...
    fragColor = vec4(tc.xy,0,1)
}
```

7. Example Results

Your result should be similar to the example shown in Figure 3. Compare with your results, and refine your implementation to match the results.

8. What to Submit

- Source, project (or makefile), and executable files (90 pt)
- A PDF report file: YOURID-YOURNAME-A2.pdf (10 pt)
- Compress all the files into a single archive and rename it as YOURID-YOURNAME-A2.7z.
- Use i-campus to upload the file. You need to submit the file at the latest 23:59, the due date (see the course web page).