# Computer Graphics

Instructor: Sungkil Lee
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

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

A sphere is represented by its origin and radius, but we again need to convert such implicit surfaces to a finite set of triangular primitives. A common approximation is its subdivision both in the longitude and latitude (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.

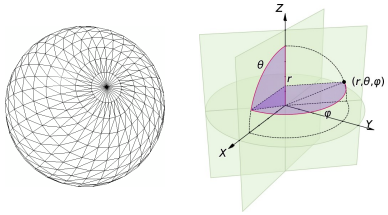


**Figure 1:** A triangular approximation of a sphere (left) and the definition of spherical coordinates in RHS coordinate system (right).

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

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

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 (covered 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 normal vectors and texture coordinates. Getting the normal vector $N(x, y, z)$ is simple. Just normalize your position or omit the radius term in the position.

$$N(x, y, z) = (\sin\theta \cos\varphi, \sin\theta \sin\varphi, \cos\theta), \qquad (2)$$

Even the texture coordinates $T$ are simple.

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

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 in the canonical view volume (CVV) of the LHS convention is located at $(0, 0, 0)$ and is directed towards $(0, 0, 1)$.

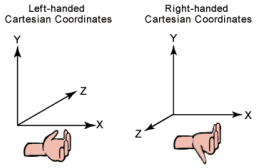


**Figure 2:** Comparison of LHS and RHS coordinate systems. The canonical view volume of OpenGL uses the LHS convention.

Since Eq. (1) defines the positions in the world coordinate system (with the RHS convention), we need to convert the RHS convention to the LHS convention; i.e., $x$, $y$, $z$ axes in the world need to be converted to $-z'$, $x'$, $y'$ axes in the CVV.

Such a process can be automatically handled by view and projection matrices, but you did not learn them (they will be covered later). In this assignment, we will not care about them, but sill need to properly configure the camera. All you have to do here is to use the matrix below to apply view transformation and projection together, using the matrix in Eq. (4).

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad (4)$$

More precisely, let $eye$ and $at$ be $(1, 0, 0)$ and $(0, 0, 0)$ in the world. Then, the matrix multiplies a change-of-frame matrix and translation matrix $T(-1, 0, 0)$; moving the scene to $(-1, 0, 0)$ is equivalent to moving the camera to $(1, 0, 0)$. In the end, you also need to apply aspect correction (used in "cgcirc" sample). In your program, you can pass the matrix to a uniform variable as follows.

```
void update()
{
    float aspect = window_size.x/float(window_size.y);
    mat4 aspect_matrix = mat4::scale(
        std::min(1/aspect,1.0f),std::min(aspect,1.0f),1.0f);
    mat4 view_projection_matrix = aspect_matrix*
        mat4 { 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);
}
```

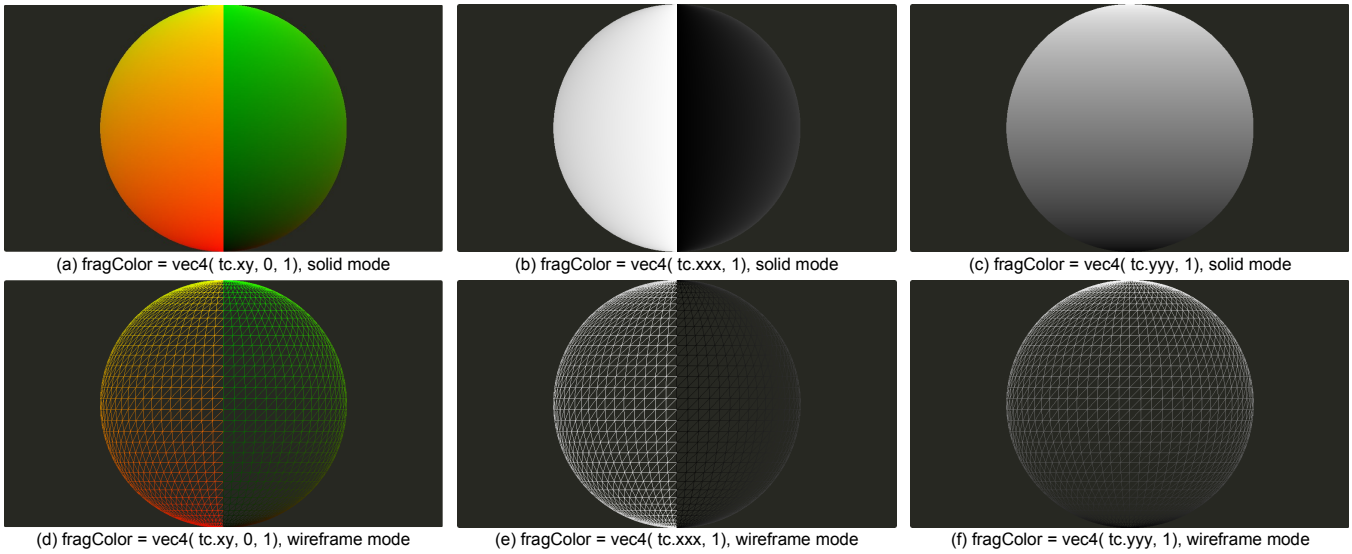


**Figure 3:** Example spheres drawn with three visualization schemes and two rendering modes.

## 5. Implementation and Requirements

You may start from "cgcirc" (or "cgtransform") project, already distributed on the course web. Precise requirements are as follows.

- 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, normal vectors, 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 seamlessly (at the stopped location) (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); }
```

## 6. 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.

## 7. 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 back-facing faces.
- Initialize the window size whose aspect ratio (width/height) is $16/9$ (e.g., $1280 \times 720$). In other words, 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 see the triangular approximation. This is because the sphere will be look like a circle, but the triangular structure should be different from the circle.

## 8. What to Submit

- Source, project (or makefile), and executable files (90 pt). Executable files (e.g., *.exe) should also include shader files (and asset files, if necessary) in /bin/shaders/ directory. Also, please make sure to conform to the directory structure convention of distributed samples (e.g., /src/, /bin/, and /bin/shaders/).
- 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. (or yourid-yourname-a2.zip)
- 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).