Placing the Eye Position on an Orbiting Body

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License

```
#include <stdio.h>
#include <string>
#define _USE_MATH_DEFINES
#include <cmath>
#define GLM_FORCE_RADIANS
#include "glm/vec2.hpp"
#include "glm/vec3.hpp"
#include "glm/mat4x4.hpp"
#include "glm/gtc/matrix_transform.hpp"
#include "glm/gtc/matrix_inverse.hpp"
#include "glm/gtc/type_ptr.hpp"
#include <GL/gl.h>
#include <GL/glu.h>
```

```
const float EYEDISTFACTOR = 0.75f;
const float ZFAR = 1000000.0f;
const float FOVDEG = 90.0f;
enum views {
    OUTSIDEVIEW, EARTHVIEW, MOONVIEW, CORVALLISVIEW
};
enum views NowView;
float Time;
```

Make Time go from 0. to however many seconds you want in your total animation, not 0. to 1.

Program Setup

The different eye views you can use

```
void glOrtho( float left, float right, float bottom, float top, float near, float far );
```

```
void gluPerspective( float fovy, float aspect, float zNear, float zFar );
```

Be careful because objects can disappear due to clipping:

- Items in your scene closer to you than znear in front of your eye will be clipped away.
- Items in your scene farther from you than zfar in front of your eye will be clipped away.

This makes them hard to debug.

When we started doing computer graphics, the objects were fairly small, so ’0.1f, 1000.0f’ worked. However, now we are doing solar systems, which could, potentially, have much larger coordinates. Our special eye-position math helps set znear. But, depending on how you construct your scene, you might have to adjust zfar.

Physical Parameter Setup

These are the actual numbers for our solar system. You would need to change them to your exaggerated numbers! Also, you might need to change the znear and zfar values in your call to gluPerspective() to work with whatever scale you choose.

> Warning: these are the actual numbers for our solar system. You would need to change them to your exaggerated numbers!
Here's How I Tried Scaling Things

### Display List Setup

```c
// 1. SpheDL = glGenLists(1);
//    gluSphere(SpheDL, 0.5f, 128, 128);
// 2. SunDL = glGenLists(1);
//    gluSphere(SunDL, 1.0f, 512, 512);
// 3. MoonDL = glGenLists(1);
//    gluSphere(MoonDL, 0.1f, 512, 512);
```

A display list can call a previously-created display list

### Timing Setup

```
const int MAXIMUM_TIME_SECONDS = 10*60; // if decided to use 10 minutes
const int MAXIMUM_TIME_MILLISECONDS = MAXIMUM_TIME_SECONDS * 1000;
```

In the `Animate( )` function:

```c
int ms = glutGet(GLUT_ELAPSED_TIME); // milliseconds
```

### Earth Transformations

Steps to transform the Earth-eye-viewing system into Solar System Coordinates:

1. Spin the Earth by `EarthSpinAngle` about its Y axis
2. Tilt the Earth by 23.5° about its X axis
3. Translate the Earth to where it belongs in its orbit

\[
\begin{bmatrix}
N_x \\
N_y \\
N_z \\
1
\end{bmatrix}
\]

\[
M_{WS}
\]
Moon Transformations

Steps to transform the Moon-eye-viewing system:
1. Spin the Moon by MoonSpinAngle about its Y axis
2. Translate the Moon by MOON_ORBITAL_RADIUS_MILES in its X direction
3. Revolve the Moon by MoonOrbitAngle about the Earth's Y axis
4. Incline the Moon by MOON_ORBITAL_INCLINATION_DEG about the Z axis
5. Translate the Earth by EARTH_ORBITAL_RADIUS_MILES in its X direction
6. Revolve the Earth by EarthOrbitAngle about the Sun's Y axis

```
glm::mat4 makeMoonMatrix() {
    float moonSpinAngle = ONE_HALF_TURN * Time * TimeScale / MOON_SPIN_TIME_SECONDS;
    float moonOrbitAngle = ONE_HALF_TURN * Time * TimeScale / MOON_ORBIT_TIME_SECONDS;
    float earthOrbitAngle = ONE_HALF_TURN * Time * TimeScale / EARTH_ORBIT_TIME_SECONDS;

    glm::mat4 identity = glm::mat4( 1.);
    glm::vec3 yaxis = glm::vec3(0., 1., 0.);
    glm::vec3 zaxis = glm::vec3(0., 0., 1.);

    // 6. */ glm::mat4 erorbity = glm::rotate( identity, earthOrbitAngle, yaxis );
    // 5. */ glm::mat4 etransx = glm::translate( identity, glm::vec3(EARTH_ORBITAL_RADIUS_MILES, 0., 0.));
    // 4. */ glm::mat4 mrorbity = glm::rotate( identity, moonOrbitAngle, yaxis );
    // 3. */ glm::mat4 mrinclinez = glm::rotate( identity, glm::radians(MOON_ORBITAL_INCLINATION_DEG), zaxis );
    // 2. */ glm::mat4 mtransx = glm::translate( identity, glm::vec3(MOON_ORBITAL_RADIUS_MILES, 0., 0.));
    // 1. */ glm::mat4 mrspiny = glm::rotate( identity, moonSpinAngle - moonOrbitAngle, yaxis );

    return erorbity * etransx * mrinclinez * mrorbity * mtransx * mrspiny;  // 6 * 5 * 4 * 3 * 2 * 1}
```

Note: EarthSpinAngle and EarthTiltAngle have no effect on the Moon's matrix.

Transformations In Action!

Earth Viewing

In model coordinates:
1. Put the look-at-position at an arbitrary latitude and longitude
2. Set the eye-position at an arbitrary latitude and longitude
3. Set the up-vector to be perpendicular to the surface at the look-at-position

Now, all we have to do is transform those two locations and one vector into Solar System Coordinates (I hate to call them "World Coordinates" here...).
Converting Latitude-Longitude to XYZ

```cpp
glm::vec3 LatLngToXYZ(float lat, float lng, float rad)
{
    lat = glm::radians(lat);
    lng = glm::radians(lng);
    glm::vec3 xyz;
    float xz = cosf(lat);
    xyz.x = -rad * xz * cosf(lng);
    xyz.y =  rad * sinf(lat);
    xyz.z =  rad * xz * sinf(lng);
    return xyz;
}
```

Convert a latitude and longitude (in degrees) and a planet radius to an (x,y,z). This assumes that (0,0,0) is at the center of the planet.

Converting Eye+Look Latitude-Longitude to Eye+Look XYZ:

```cpp
void SetViewingFromLatLng(float eyeLat, float eyeLng, float lookLat, float lookLng, float rad, glm::vec4 * eyep, glm::vec4 * lookp, glm::vec4 * upp, float *znearp)
{
    glm::vec3 center = glm::vec3(0., 0., 0.); // center of planet
    glm::vec3 eye  = LatLngToXYZ(eyeLat, eyeLng, rad);
    glm::vec3 look = LatLngToXYZ(lookLat, lookLng, rad);
    glm::vec3 upVec = glm::normalize(look - center); // perpendicular to the globe at the look position
    *eyep = glm::vec4(eye, 1.);
    *lookp = glm::vec4(look, 1.);
    *upp = glm::vec4(upVec, 0.);
    *znearp = 0.1f;
}
```

Convert a latitude and longitude eye position and look-at position (in degrees) and a planet radius to an eye position and a look-at position in XYZ coordinates.

Now, all we have to do is transform those two locations and one vector into Solar System Coordinates (I hate to call them “World Coordinates” here...).

Here’s the Viewing Strategy

1. Pick a (latitude,longitude) for the eye position
2. Pick a (latitude,longitude) for the look-at position
3. Use the surface normal at the look-at position for the up-vector

The up-vector is a unit vector perpendicular to the planet at the look-at position.
But There is a Problem -- We Cannot Leave the Eye There!

With this eye and look combination, we will be looking up through the planet.

We are about to use some vector math!
Vector math is a big deal in computer graphics. Your games use it all the time.
You don’t have to completely understand vector math to appreciate what we are about to use it for.
But, if you would like a review of vector math, go to:
http://cs.oregonstate.edu/~mjb/cs491/Handouts/vectors.1pp.pdf

Sidebar: Vectors have Direction and Magnitude

Magnitude: \[ \|V\| = \sqrt{V_x^2 + V_y^2 + V_z^2} \]

Sidebar: Unit Vectors have a Magnitude = 1.0

Unit Vector: \[ \hat{V} = \frac{V}{\|V\|} \]  The circumflex (^) tells us this is a unit vector
So, how much of \( \mathbf{A} \) lives in the \( \mathbf{B} \) direction is that magnitude times the \( \mathbf{B} \) unit vector:

\[
(\mathbf{A} \cdot \mathbf{B}) \mathbf{B}
\]

The viewing strategy is to make the eye-to-look tangent to the planet.

But there is another problem – we have to set the near clipping plane carefully.

Seeing under the top surface of the planet shows the continents reversed!
Here's the Viewing Strategy: Use All These Parameters to Compute \( N \)

1. Field-of-View Angle
2. Eye Position
3. Near Clip Distance, \( N \)
4. Y distance from Eye to Planet Surface, \( d \)
5. Look-at Position
6. Distance from Look-at to Eye, \( E \)
7. Y distance from Eye to Planet Surface, \( d \)
8. Center of Planet

At the point \( S \) where the near clipping plane touches the planet surface:

\[
\Delta x = E - N \\
\Delta y = R - d \\
\]

By experimenting, I found decent values for \( f \) to be between 0.75 – 1.25. In these notes (slide #2), \( f \) is called EYEDISTFACTOR.
Put this in the `Display()` function:

```c
void Display() {
    ... ...
    glm::mat4 e = MakeEarthMatrix();
    glm::mat4 m = MakeMoonMatrix();
    glm::vec4 eyePos = glm::vec4(0., 0., 0., 1.); // a position
    glm::vec4 lookPos = glm::vec4(0., 0., 0., 1.); // a position
    glm::vec4 upVec = glm::vec4(0., 0., 0., 0.); // a vector, so doesn't get translations applied
    float zNear = 0.1f;
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    switch (NowView) {
        ... ...
        case OUTSIDEVIEW: // 1st way to set gluLookAt() ...
            gluLookAt(0., 0., 30000.f, 0., 0., 0., 0., 1., 0.);
            glRotatef((GLfloat)Yrot, 0., 1., 0.);
            glRotatef((GLfloat)Xrot, 1., 0., 0.);
            if (Scale < MINSCALE)
                Scale = MINSCALE;
            glScalef(Scale, Scale, Scale);
            break;
        ... ...
    }
    Put this in the Display() function:

    Standard Outside Viewing
    void Display() {
        ... ...
        glm::mat4 e = MakeEarthMatrix();
        glm::mat4 m = MakeMoonMatrix();
        glm::vec4 eyePos = glm::vec4(0., 0., 0., 1.);
        glm::vec4 lookPos = glm::vec4(0., 0., 0., 1.);
        glm::vec4 upVec = glm::vec4(0., 0., 0., 0.); // a vector, so doesn't get translations applied
        float zNear = 0.1f;
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
        switch (NowView) {
            ... ...
            case OUTSIDEVIEW: // 1st way to set gluLookAt() ...
                gluLookAt(0., 0., 30000.f, 0., 0., 0., 0., 1., 0.);
                glRotatef((GLfloat)Yrot, 0., 1., 0.);
                glRotatef((GLfloat)Xrot, 1., 0., 0.);
                if (Scale < MINSCALE)
                    Scale = MINSCALE;
                glScalef(Scale, Scale, Scale);
                break;
            ... ...
        }
    }

    Earth Viewing
    void Display() {
        ... ...
        glm::mat4 e = MakeEarthMatrix();
        glm::mat4 m = MakeMoonMatrix();
        glm::vec4 eyePos = glm::vec4(0., 0., 0., 1.);
        glm::vec4 lookPos = glm::vec4(0., 0., 0., 1.);
        glm::vec4 upVec = glm::vec4(0., 0., 0., 0.); // a vector, so doesn't get translations applied
        float zNear = 0.1f;
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
        switch (NowView) {
            ... ...
            case EARTHVIEW: // 2nd way to set gluLookAt() ...
                SetViewingFromLatLng(0.f, 70.f, 0.f, 60.f, EARTH_RADIUS_MILES, &eyePos, &lookPos, &upVec, &zNear);
                eyePos = e * eyePos;
                lookPos = e * lookPos;
                upVec = e * upVec;
                gluLookAt(eyePos.x, eyePos.y, eyePos.z, lookPos.x, lookPos.y, lookPos.z, upVec.x, upVec.y, upVec.z);
                break;
            ... ...
        }
    }

    Moon Viewing
    void Display() {
        ... ...
        glm::mat4 e = MakeEarthMatrix();
        glm::mat4 m = MakeMoonMatrix();
        glm::vec4 eyePos = glm::vec4(0., 0., 0., 1.);
        glm::vec4 lookPos = glm::vec4(0., 0., 0., 1.);
        glm::vec4 upVec = glm::vec4(0., 0., 0., 0.); // a vector, so doesn't get translations applied
        float zNear = 0.1f;
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
        switch (NowView) {
            ... ...
            case MOONVIEW: // 3rd way to set gluLookAt() ...
                SetViewingFromLatLng(1.175f, 1.97f, 0.1, 0.1, MOON_RADIUS_MILES, &eyePos, &lookPos, &upVec, &zNear);
                eyePos = m * eyePos;
                lookPos = m * lookPos;
                upVec = m * upVec;
                gluLookAt(eyePos.x, eyePos.y, eyePos.z, lookPos.x, lookPos.y, lookPos.z, upVec.x, upVec.y, upVec.z);
                break;
            ... ...
        }
    }

    Transformations in Action!
    Put this in the Display() function:
Corvallis sits at Latitude 44.57° N x Longitude 123.27° W

Treating lat-long as spherical coordinates and solve for x, y, and z:

float y = sinf(44.57°); // 0.702
float xz = cosf(44.57°); // 0.712
float x = -xz * sinf(123.27°); // -0.391
float z = xz * cosf(123.27°); // 0.596

Then multiply x, y, and z by EARTH_RADIUS_MILES

Because of the way the coordinates work, Corvallis's west longitude needs to positive, even though on maps, west longitude is negative.

Put this in the Display() function:

```cpp
float y = EARTH_RADIUS_MILES * sinf(44.57°);
float x = EARTH_RADIUS_MILES * x * cosf(123.27°);
float z = EARTH_RADIUS_MILES * xz * sinf(123.27°);
```

Transformations In Action!