Putting the Eye Position on an Orbiting Body

Program Setup

#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 <GL/gl.h>
#include <GL/glu.h>

const float ZNEAR = 1.0;
const float ZFAR = 1000000.0;

enum views
{
  OUTSIDEVIEW, EARTHVIEW, MOONVIEW, CORVALLISVIEW
};

float Time;
enum views WhichView;

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

Timing Setup

At the top of the program:

const int MAXIMUM_TIME_SECONDS = 10*60; // I decided to use 10 minutes
const float ONE_FULL_TURN = 2. * M_PI; // this is 2π instead of 360° because glm uses radians

In the Animate() function:

int ms = glutGet( GLUT_ELAPSED_TIME ); // milliseconds
ms %= MAXIMUM_TIME_MILLISECONDS; // [0, MAXIMUM_TIME_MILLISECONDS-1]
Time = (float)ms / 1000.f; // seconds
// note that Time goes from 0. to however many seconds you asked for, not 0 – 1.!
// force a call to Display():
glutSetWindow(MainWindow);
glutPostRedisplay();

In the InitGraphics() function:

Among other things...

Timing Setup

At the top of the program:

const int MAXIMUM_TIME_MILLISECONDS = 1000 * MAXIMUM_TIME_SECONDS;
const float ONE_FULL_TURN = 2. * M_PI; // this is 2π instead of 360° because glm uses radians

In the Animate() function:

int ms = glutGet( GLUT_ELAPSED_TIME ); // milliseconds
ms %= MAXIMUM_TIME_MILLISECONDS; // [0, MAXIMUM_TIME_MILLISECONDS-1]
Time = (float)ms / 1000.f; // seconds
// note that Time goes from 0. to however many seconds you asked for, not 0 – 1.!
// force a call to Display():
glutSetWindow(MainWindow);
glutPostRedisplay();

In the InitGraphics() function:

Among other things...

For a lot of our projects, a ZFAR of 1000. works well. But, for something bigger, like the solar system, you will need a ZFAR that will contain the whole depth of the scene.

We are defining them here because we will also need to know ZNEAR to help set the on-the-planet views.

Convert a latitude and longitude position and a planet radius to an eye position and a look-at position. A line from the eye position to the look-at position will end up being tangent to the globe.

Gram-Schmidt orthogonalization: forces the eyeToLook vector to be perpendicular to the up vector by subtracting the part that is not perpendicular.

These are the near and far clipping plane distances in front of the eye. The ZNEAR is typically pretty small, but the ZFAR depends on the scene.

For a lot of our projects, a ZFAR of 1000. works well. But, for something bigger, like the solar system, you will need a ZFAR that will contain the whole depth of the scene.

We are defining them here because we will also need to know ZNEAR to help set the on-the-planet views.

The different eye positions to use

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

Convert a latitude and longitude (in degrees) and a planet radius to an (x,y,z)

Convert a latitude and longitude eye position and a latitude/longitude look-at position (in degrees) and a planet radius to an eye position and a look-at position. A line from the eye position to the look-at position will end up being tangent to the globe.

Gram-Schmidt orthogonalization: forces the eyeToLook vector to be perpendicular to the up vector by subtracting the part that is not perpendicular.

These are the near and far clipping plane distances in front of the eye. The ZNEAR is typically pretty small, but the ZFAR depends on the scene.

For a lot of our projects, a ZFAR of 1000. works well. But, for something bigger, like the solar system, you will need a ZFAR that will contain the whole depth of the scene.

We are defining them here because we will also need to know ZNEAR to help set the on-the-planet views.

The different eye positions to use

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

At the top of the program:

```cpp
const float SUN_RADIUS_MILES = 432690.;
const float SUN_SPIN_TIME_DAYS_EQUATOR = 25.;
const float SUN_SPIN_TIME_HOURS_EQUATOR = SUN_SPIN_TIME_DAYS_EQUATOR * 24.;
const float SUN_SPIN_TIME_SECONDS_EQUATOR = SUN_SPIN_TIME_HOURS_EQUATOR * 60. * 60.;
const float SUN_SPIN_TIME_DAYS_POLES = 35.;
const float SUN_SPIN_TIME_HOURS_POLES = SUN_SPIN_TIME_DAYS_POLES * 24.;
const float SUN_SPIN_TIME_SECONDS_POLES = SUN_SPIN_TIME_HOURS_POLES * 60. * 60.;
const float EARTH_RADIUS_MILES = 3964.19;
const float EARTH_ORBITAL_RADIUS_MILES = 92900000.;
const float EARTH_ORBIT_TIME_DAYS = 365.3;
const float EARTH_ORBIT_TIME_HOURS = EARTH_ORBIT_TIME_DAYS * 24.;
const float EARTH_ORBIT_TIME_SECONDS = EARTH_ORBIT_TIME_HOURS * 60. * 60.;
const float EARTH_SPIN_TIME_DAYS = 0.9971;
const float EARTH_SPIN_TIME_HOURS = EARTH_SPIN_TIME_DAYS * 24.;
const float EARTH_SPIN_TIME_SECONDS = EARTH_SPIN_TIME_HOURS * 60. * 60.;
const float MOON_RADIUS_MILES = 1079.6;
const float MOON_ORBITAL_RADIUS_MILES = 238900.;
const float MOON_ORBIT_TIME_DAYS = 27.3;
const float MOON_ORBIT_TIME_HOURS = MOON_ORBIT_TIME_DAYS * 24.;
const float MOON_ORBIT_TIME_SECONDS = MOON_ORBIT_TIME_HOURS * 60. * 60.;
const float MOON_SPIN_TIME_DAYS = MOON_ORBIT_TIME_DAYS;
const float MOON_SPIN_TIME_HOURS = MOON_SPIN_TIME_DAYS * 24.;
const float MOON_SPIN_TIME_SECONDS = MOON_SPIN_TIME_HOURS * 60. * 60.;
```

Warning: 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.

### Possibly Adjusting the Viewing Volume

Remember these lines from our sample code?

```cpp
if( WhichProjection == ORTHO )
glOrtho( -3., 3.,     -3., 3.,     ZNEAR, ZFAR );
else
  gluPerspective( 90., 1.,           ZNEAR, 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 "0.1, 1000." worked. However, now we are doing solar systems, which could, potentially, have much larger coordinates. So, depending on how you construct your scene, you might have to adjust ZNEAR and (especially) ZFAR.

### Earth Transformations

Let's make this fairly straightforward. In model coordinates:

1. Put the eye-position at the corner of its Equator and Prime Meridian (xe = EARTH_RADIUS_MILES, ye = 0., ze = 0.)
2. Set the look-at position to be on a straight-line east of the eye-position: (xl = EARTH_RADIUS_MILES, yl = 0., zl = -100.)
3. Set the up-vector to be: (xu = 100., yu = 0., zu = 0.)

This makes our view-vector (from the eye-position to the look-at position) tangent to the Earth's surface, which is a good way to start.

Now, all we have to do is transform those 3 locations/vector into Solar System Coordinates. (I hate to call them "World Coordinates" here...)
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. Translate the Earth by EARTH_ORBITAL_RADIUS_MILES in its X direction
3. Revolve the Earth by EarthOrbitAngle about the Sun's Y axis

```cpp
glm::mat4 MakeEarthMatrix() {
    float earthSpinAngle = Time * EARTH_SPIN_TIME_SECONDS * ONE_FULL_TURN;
    float earthOrbitAngle = Time * EARTH_ORBIT_TIME_SECONDS * ONE_FULL_TURN;
    glm::mat4 identity = glm::mat4(1.);
    glm::vec3 yaxis = glm::vec3(0., 1., 0.);
    /* 3. */ glm::mat4 erorbity = glm::rotate(identity, earthOrbitAngle, yaxis);
    /* 2. */ glm::mat4 etransx = glm::translate(identity, glm::vec3(EARTH_ORBITAL_RADIUS_MILES, 0., 0.));
    /* 1. */ glm::mat4 erspiny = glm::rotate(identity, earthSpinAngle, yaxis);
    return erorbity * etransx * erspiny; // 3 * 2 * 1
}
```

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. Translate the Earth by EARTH_ORBITAL_RADIUS_MILES in its X direction
5. Revolve the Earth by EarthOrbitAngle about the Sun's Y axis

```cpp
glm::mat4 MakeMoonMatrix() {
    float moonSpinAngle = Time * MOON_SPIN_TIME_SECONDS * ONE_FULL_TURN;
    float moonOrbitAngle = Time * MOON_ORBIT_TIME_SECONDS * ONE_FULL_TURN;
    float earthOrbitAngle = Time * EARTH_ORBIT_TIME_SECONDS * ONE_FULL_TURN;
    glm::mat4 identity = glm::mat4(1.);
    glm::vec3 yaxis = glm::vec3(0., 1., 0.);
    /* 5. */ glm::mat4 erorbity = glm::rotate(identity, earthOrbitAngle, yaxis);
    /* 4. */ glm::mat4 etransx = glm::translate(identity, glm::vec3(EARTH_ORBITAL_RADIUS_MILES, 0., 0.));
    /* 3. */ glm::mat4 mrorbity = glm::rotate(identity, moonOrbitAngle, yaxis);
    /* 2. */ glm::mat4 mtransx = glm::translate(identity, glm::vec3(MOON_ORBITAL_RADIUS_MILES, 0., 0.));
    /* 1. */ glm::mat4 mrspiny = glm::rotate(identity, moonSpinAngle, yaxis);
    return erorbity * etransx * mrorbity * mtransx * mrspiny; // 5 * 4 * 3 * 2 * 1
}
```

Note: EarthSpinAngle has no effect on the Moon's matrix.
Put this in the Display() function:

```cpp
void Display()
{
    glm::mat4 e, m;
    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.); // vectors don't get translations
    glm::mat4 glModelView = GL_MODELVIEW ? glm::mat4() : glm::mat4(1.);
    switch (WhichView)
    {
    case MOONVIEW:
        // 3rd way to set gluLookAt()
        m = MakeMoonMatrix();
        SetViewingFromLatLng(0., 0., 0., -10., MOON_RADIUS_MILES, &eyePos, &lookPos);
        m = MakeMoonMatrix();
        upVec = glm::normalize(glm::vec3(eyePos));
        eyePos = m * eyePos;
        lookPos = m * lookPos;
        upVec = glm::vec3(m * glm::vec4(upVec, 0.));
        glTranslatef(0., 0., -4.f * ZNEAR);
        gluLookAt(eyePos.x, eyePos.y, eyePos.z, lookPos.x, lookPos.y, lookPos.z, upVec.x, upVec.y, upVec.z);
        break;
    . . .
    }
}
```

### Earth/Corvallis Transformations

```cpp
float y = EARTH_RADIUS_MILES * sin(44.57°); // 0.702
float xz = cos(44.57°); // 0.712
float x = xz * cos(123.27°); // -0.391
float z = xz * sin(123.27°); // 0.596
```

### What if You Want the Eye at Corvallis (or some other arbitrary location)?

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 = sin(44.57°);` // 0.702
- float `x = xz * cos(123.27°);` // -0.391
- float `z = xz * sin(123.27°);` // 0.596

Then multiply x, y, and z by EARTH_RADIUS_MILES
glm::mat4 e = MakeEarthMatrix();
glm::mat4 m = MakeMoonMatrix();

gEnable(GL_TEXTURE_2D);

GLfloat EarthTex = GetTextureID(EarthImage);
GLfloat MoonTex = GetTextureID(MoonImage);

gPushMatrix();

gBindTexture(GL_TEXTURE_2D, EarthTex);

gTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);

gMultMatrixf(glm::value_ptr(e));
gCallList(EarthList);

gPopMatrix();

// Moon

GLfloat MoonTex = GetTextureID(MoonImage);

gPushMatrix();

gBindTexture(GL_TEXTURE_2D, MoonTex);

gTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);

gMultMatrixf(glm::value_ptr(m));
gCallList(MoonList);

gPopMatrix();

Note: You Can Also Use these Matrices to Draw the Objects in the Proper Locations instead of using glRotatef() and glTranslatef()