Introduction of Orrery

Word Count: 919

Creating an orrery, or a simulation of a solar system, presents many challenges both mathematically and logically. A vast amount of computations are needed to be able to update the system realistically.

Mathematical Concepts Used

Gravitational Orbits

Problem- Realistically simulating the gravitational orbit of planets

In universal law, gravity is the principle that two particles attract each other with forces directly proportional to the product of their masses divided by the square of the distance between them (Newton, 2010). To correctly simulate gravity, we would have to know all the forces between all the objects at all points.

Explanation

Newton's law of universal gravitation states that every point mass attracts every single other point mass by a force pointing along the line intersecting both points. The force is proportional to the product of the two masses and inversely proportional to the square of the distance between them. Put into an equation this is:

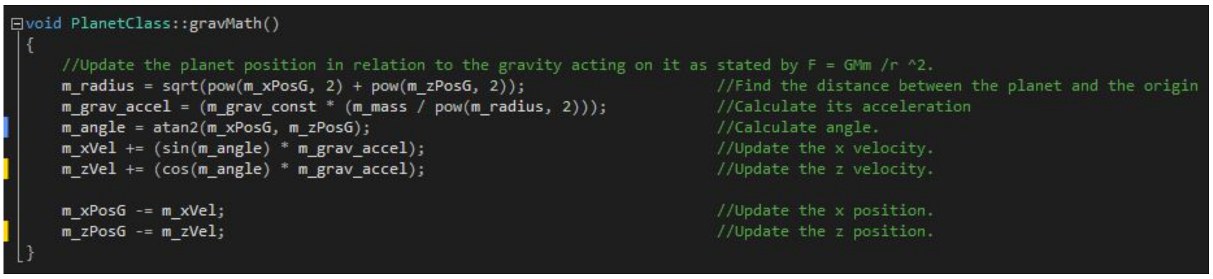
$$F = G\frac{m_1 m_2}{r^2}$$

Where, F is the force between the two masses (m_1 *and* m_2), G is the gravitational constant (defined as $6.673\times10^{-11}\ N\cdot(m/kg)^2$) and r is the distance between the two masses.

Solution

Using Pythagorean Theorem ($a^2 + b^2 = c^2$) (Weisstein, Pythagorean Theorem, n.d.)We can easily calculate the distance between the sun (at the origin) and the planet by finding the square root of the sum of x and the z coordinates squared (as we are rotating around the y axis). We then put our values into the gravity equation to find the force (or the acceleration value) between the sun and the planet. As we have given the planet an initial position, we then calculate the angle between this vector and the x axis using $atan2$ (Pitt-Francis & Whiteley, 2012). Knowing $theta$ we can now update the planets velocity using more trigonometry ($sin\ (theta)\ * F\ and\ cos\ (theta)\ * F\ for\ the\ x\ and\ z\ velocities\ respectively$). We then update our planets position from the velocity.

Source Code



```
void PlanetClass::gravMath()
{
    //Update the planet position in relation to the gravity acting on it as stated by F = GMm /r ^2.
    m_radius = sqrt(pow(m_xPosG, 2) + pow(m_zPosG, 2));                 //Find the distance between the planet and the origin
    m_grav_accel = (m_grav_const * (m_mass / pow(m_radius, 2)));        //Calculate its acceleration
    m_angle = atan2(m_xPosG, m_zPosG);                                  //Calculate angle.
    m_xVel += (sin(m_angle) * m_grav_accel);                            //Update the x velocity.
    m_zVel += (cos(m_angle) * m_grav_accel);                            //Update the z velocity.

    m_xPosG -= m_xVel;                                                  //Update the x position.
    m_zPosG -= m_zVel;                                                  //Update the z position.
}
```

The entire calculation is needed to be performed every frame and is done from one function which is called for each planet. Simulating gravity in this way meant we could achieve an elliptical orbit by setting the initial planet position and velocity tangent to the orbit that we wanted to achieve and allow the function to process it from then on.

Matrix Rotation

Problem- Rotation within a 3D environment

A rotation matrix is a matrix that is used to perform a rotation in Euclidean space (Arvo, 1992). Typically we can use Euler Angles for an object's rotation- However this can lead to problems such as Gimbal Lock (Simeone). You also have to be careful at the order of your matrix multiplication as matrix multiplication is not commutative (Operations with Matrices, n.d.).

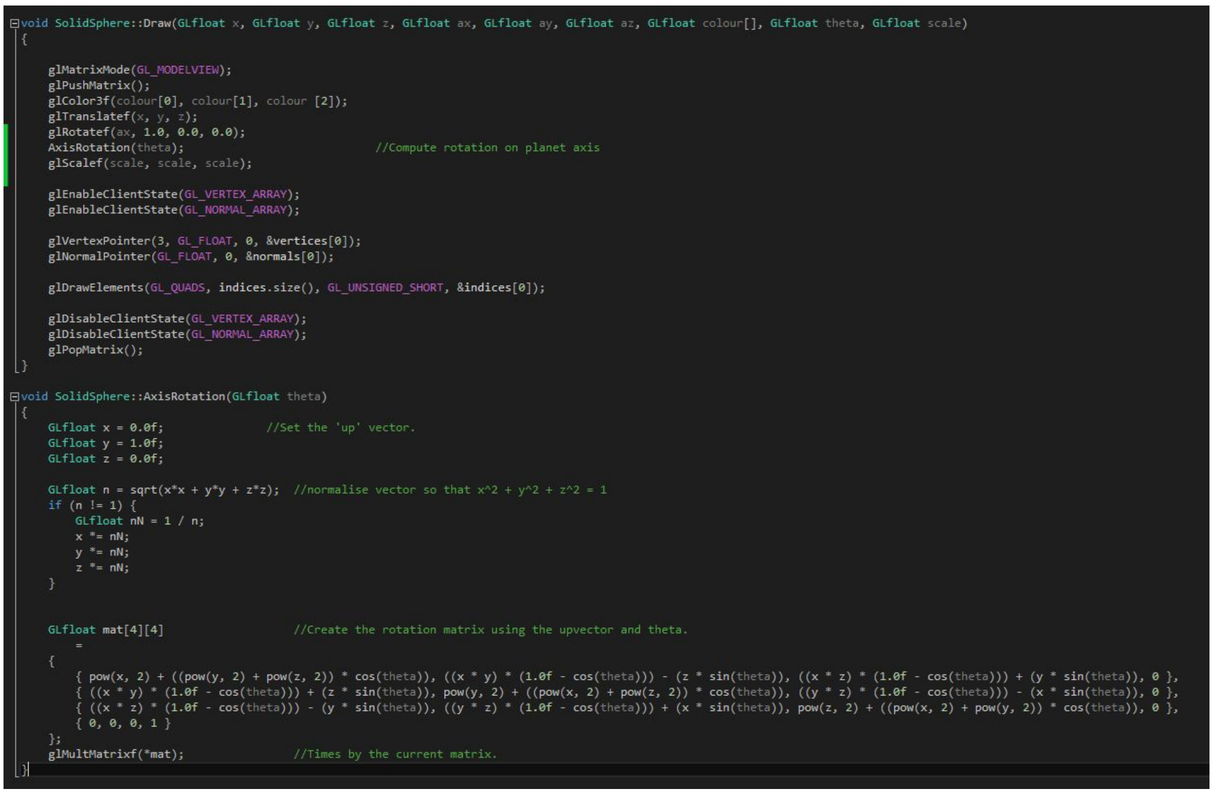
Explanation

Euler angles are a way to represent the 3D orientation of an object using a combination of three rotations around different axes (Weisstein, Euler Angles, n.d.). Rotation matrices are a means representing a rotation about the origin.

Solution

OpenGL’s matrices can be either column-major or row-major and are on a right-handed coordinate space (Martz, n.d.). To correctly rotate our planets on their tilt and translate to their correct position we use a combination of a translation matrix, scaling matrix, and two rotation matrices. First we scale our object down by a desired amount. Then we create a rotation matrix which allows the object to be rotated by $theta$ around an arbitrary axis defined by a vector. This matrix was written when I had originally planned to have the planets orbit around one another rather than just the sun. It was left in the program in case this might be desired in the future, and also as an example of how rotation matrices work. We then use Euler angles with *glRotatef* on the x axis to rotate the planet onto its tilt and multiple with our current matrix. Then our multiplied matrix is translated to its desired position as defined by our gravity function.

Source Code



```cpp
void SolidSphere::Draw(GLfloat x, GLfloat y, GLfloat z, GLfloat ax, GLfloat ay, GLfloat az, GLfloat colour[], GLfloat theta, GLfloat scale)
{

    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
    glColor3f(colour[0], colour[1], colour [2]);
    glTranslatef(x, y, z);
    glRotatef(ax, 1.0, 0.0, 0.0);
    AxisRotation(theta);                        //Compute rotation on planet axis
    glScalef(scale, scale, scale);

    glEnableClientState(GL_VERTEX_ARRAY);
    glEnableClientState(GL_NORMAL_ARRAY);

    glVertexPointer(3, GL_FLOAT, 0, &vertices[0]);
    glNormalPointer(GL_FLOAT, 0, &normals[0]);

    glDrawElements(GL_QUADS, indices.size(), GL_UNSIGNED_SHORT, &indices[0]);

    glDisableClientState(GL_VERTEX_ARRAY);
    glDisableClientState(GL_NORMAL_ARRAY);
    glPopMatrix();
}

void SolidSphere::AxisRotation(GLfloat theta)
{
    GLfloat x = 0.0f;               //Set the 'up' vector.
    GLfloat y = 1.0f;
    GLfloat z = 0.0f;

    GLfloat n = sqrt(x*x + y*y + z*z);  //normalise vector so that x^2 + y^2 + z^2 = 1
    if (n != 1) {
        GLfloat nN = 1 / n;
        x *= nN;
        y *= nN;
        z *= nN;
    }


    GLfloat mat[4][4]                   //Create the rotation matrix using the upvector and theta.
        =
    {
        { pow(x, 2) + ((pow(y, 2) + pow(z, 2)) * cos(theta)), ((x * y) * (1.0f - cos(theta))) - (z * sin(theta)), ((x * z) * (1.0f - cos(theta))) + (y * sin(theta)), 0 },
        { ((x * y) * (1.0f - cos(theta))) + (z * sin(theta)), pow(y, 2) + ((pow(x, 2) + pow(z, 2)) * cos(theta)), ((y * z) * (1.0f - cos(theta))) - (x * sin(theta)), 0 },
        { ((x * z) * (1.0f - cos(theta))) - (y * sin(theta)), ((y * z) * (1.0f - cos(theta))) + (x * sin(theta)), pow(z, 2) + ((pow(x, 2) + pow(y, 2)) * cos(theta)), 0 },
        { 0, 0, 0, 1 }
    };
    glMultMatrixf(*mat);                //Times by the current matrix.
}
```

These have to be post multiplied in OpenGL (Wang).

Discussions and Conclusion

When creating the gravity function I choose only to have the planets be affected by gravity from the sun and not from one another. This was in part to save on frame time, but also to avoid the complications of the *n-body problem* (Aarseth, 2003) which is still a developing and problematic area of physics. If we were to simulate gravity between two objects were one was not at the origin, then we would need to use trigonometry and the dot product of the two vectors given by the objects.

Each planet's mass is its real world mass, but the sizes and initial distance from the sun were given in relation to the earth (which was given a value of 1.0). Originally the positions and sizes were all entered as their real world values and scaled down to a more manageable level. However finding the initial velocities to allow a planet to enter an orbit become difficult. I could have calculated the vector of a tangent to the desired orbit, but the easier solution was to give the positions of the planets as ratios of earth's initial position and slowly increase the positive x velocity until a desired orbit was achieved.

Another way to achieve a rotation in 3D space would be to use quaternions which allow you to rotate an object with a scalar and a vector. However I chose to stick with Euler Angles and matrices as OpenGL does not support quaternions directly and would require extra functions to convert (Bobic, 1998).

References

Aarseth, S. J. (2003). *Gravitational N-body Simulations, Tools and Algorithms.* Cambridge: Cambridge University Press.

Arvo, J. (1992). Fast random rotation matrices. In D. Kirk, *Graphics Gems III* (pp. 117-120). San Diego: Academic Press Professional.

Bobic, N. (1998, July 5). *Rotating Objects Using Quaternions*. Retrieved from Gamasutra: http://www.gamasutra.com/view/feature/131686/rotating_objects_using_quaternions.php

Houghton Mifflin Harcourt. (n.d.). *Operations with Matrices*. Retrieved from Cliffs Notes: http://www.cliffsnotes.com/math/algebra/linear-algebra/matrix-algebra/operations-with-matrices

Martz, P. (n.d.). *9. Transformations*. Retrieved from OpenGL: https://www.opengl.org/archives/resources/faq/technical/transformations.htm

Newton, I. (2010). *Philosophiae Naturalis Principia Mathematica.* Seaside: Watchmaker Publishing.

Pitt-Francis, J., & Whiteley, J. (2012). *Guide to Scientific Computing in C++.* Berlin: Springer Science & Business Media.

Simeone, A. L. (n.d.). *3D Transformations.* Retrieved from Portsmouth University Moodle: http://moodle.port.ac.uk/pluginfile.php/618283/mod_resource/content/0/Lecture%2010%20-%20Transformations.pdf

Wang, H. (n.d.). *Transformation_II.* Retrieved from Ohio State University: http://web.cse.ohio-state.edu/~whmin/courses/cse5542-2013-spring/6-Transformation_II.pdf

Weisstein, E. W. (n.d.). *Euler Angles*. Retrieved from MathWorld: http://mathworld.wolfram.com/EulerAngles.html

Weisstein, E. W. (n.d.). *Pythagorean Theorem*. Retrieved from MathWorld--A Wolfram Web Resource: http://mathworld.wolfram.com/PythagoreanTheorem.html