Cristian Milatinov

May 14th, 2018

Three-Dimensional Simulation Software of the Solar System

Comprehensive Assessment

Table of Contents

**Abstract** 2

**Introduction** 3

**Simulation Engine** 3

Setting Up 3

Main Loop 3

Vertex Array Objects and Vertex Buffer Objects 5

Three-Dimensional Models 5

Entities and Vertex Manipulation Using Matrices 6

Shader Programs 6

**Gravitational Calculations** 9

Initial Planetary State Vectors 9

Position and Velocity Updates Based on Newton’s Gravitational Law 12

Obtaining Orbital Trajectories from State Vectors 13

Rotational Axes of Planets 16

**The Application** 17

User Interface 17

Keyboard Controls 18

Configuration File 18

List of Key Bindings 19

Software Prerequisites 20

**Conclusion** 21

**Bibliography** 22

Abstract

In this project, I will be programming a 3D simulation of our solar system in which the user will be able to see the motion of planets and the different factors that affect this motion. The coding will be done in C++, a fairly modern barebone language in which most Windows applications are coded. The main tool I’ll be using is Microsoft Visual Studio which lets me write and debug C++ code. Since the main purpose of my program is to render a visible scene replicating our solar system on screen, manipulating a computer’s Graphics Processing Unit (GPU) is essential. For that purpose, I’ve chosen to use an existing library called OpenGL (Open Game Library). This library, allows me to draw shapes on screen and easily perform any kind of graphical calculations. The software I am aiming to make is an accurate representation of the real world in terms of the orbital motion of planets and the interactions between them. The application will allow the user to not only view the motion of the planets through time but also interact with them. The user will be able to place asteroids of set mass to disturb and disrupt the planets’ motion. This project will serve as a visual representation of gravitational motion in action and as a comprehensive aid in its understanding. The application will run on Windows operating systems starting from Windows 7. This report will take you through the software’s creation process including part of its coding and the relevant calculations.

Dans ce projet, je vais programmer une simulation 3D de notre système solaire dans laquelle l'utilisateur pourra voir le mouvement des planètes et les différents facteurs qui affectent ce mouvement. Le codage sera fait en C++, un langage assez moderne avec lequel la plupart des applications Windows sont codées. L'outil principal que je vais utiliser est Microsoft Visual Studio qui me permet d'écrire et de déboguer du code C++. Puisque le but principal de mon programme est de rendre une scène visible reproduisant notre système solaire à l'écran, la manipulation de l'unité de traitement graphique (GPU) d'un ordinateur est essentielle. Pour ce faire, j'ai choisi d'utiliser une bibliothèque existante appelée OpenGL (Open Game Library). Cette bibliothèque, me permet de dessiner des formes géométriques sur l'écran et d'effectuer facilement n'importe quel type de calculs graphiques. Le logiciel que je vise à faire est une représentation précise du monde réel en termes de mouvement orbital des planètes et des interactions entre elles. L'application permettra à l'utilisateur non seulement de voir le mouvement des planètes au fil du temps, mais aussi d'interagir avec eux. L'utilisateur pourra placer des astéroïdes de masse définie pour perturber le mouvement des planètes. Ce projet servira de représentation visuelle du mouvement gravitationnel en action et de guide de compréhension. L'application fonctionnera sur les systèmes d'exploitation Windows à partir de Windows 7. Ce rapport vous guidera tout au long du processus de création du logiciel, y compris une partie de son codage et des calculs pertinents.

Introduction

As mentioned above, I will be using OpenGL to render a scene on screen. OpenGL is library that allows me to send draw calls to the GPU. It is important to note that code is executed sequentially and thus, the order in which objects are drawn matters. For optimal performance, one would draw objects in scene from front to back as OpenGL has what we call a depth test. In short, the depth test allows OpenGL to discard pixels that aren’t in the camera direct line of sight. In other words, drawing objects from front to back improves performance as partially hidden pixels in objects that are obstructed by others are simply discarded, making it faster to render each frame. Note that this technique does not work in cases of transparency as objects behind others won’t be rendered. In those cases, the opposite must be done, that is, render the scene back the front. OpenGL also allows me to group up draw calls with a concept called instancing. Instancing lets me render the same object in different places using only one draw call and increases performance is the GPU does not need to reload the object’s data again. These are some of the techniques I will be taking advantage of when coding the application. The later sections will explain the general structure of an application and the rendering process.

Simulation Engine

**Setting Up**

The first thing to do when using OpenGL is to initialize the context and the display. In other words, we must tell OpenGL which version we will be using and create a window of a set size in which the application will be rendered. In my case, the version in use will be OpenGL version 4.0 with a default window size of 1600 by 900 pixels.

**Main Loop**

Every computer application in existence has what we call a main loop. A main loop is the repeating bit of code that is the framework of any program. In it, we do things such as process mouse or keyboard input, update variables in our code, or render a scene on screen. The rendering process, along with the calculation of position and velocity will be the main tasks that will be executed in my main loop. Below, I’ve included the commented main loop of the application.

//=======================================================================================

//We run this code in an endless loop until the user closes the application

//Get the current time in integer multiple of clocks   
//The length of a “clock” is determined by the system

long currentTime = clock();

//This bit gets executed roughly 60 times every second (not every time we loop)

if (currentTime - lastTime >= timePerTick \* CLOCKS\_PER\_SEC) {

//Process any user input

input->processInput(display->getWidth(), display->getHeight(), engine);

//Increments the tick counter

ticksPerSec++;

lastTime = clock();

}

//This line is to handle some other specific input when user moves asteroids  
//It is still called a “tick” even though it runs every time we loop because  
//functionality is the same but it needs to be updated every frame

input->tick();

//Print out some information to the console and update the fps counter (not important)

long currentUpdate = clock();

if (currentUpdate - lastUpdate >= CLOCKS\_PER\_SEC) {

gotoxy(0, 0);

printf("FPS : %d\n", fps);

printf("TICKS : %d\n", ticksPerSec);

if (fps = stoi(fpsCounter->textString)) {

fpsCounter->setText(" " + std::to\_string(fps) + " ", engine->fLoader);

fpsCounter->setScreenPosition(Vector2f());

}

ticksPerSec = 0;

fps = 0;

lastUpdate = clock();

}

//Calculate the time, in seconds, that has passed since we rendered the last frame

auto currentFrame = std::chrono::high\_resolution\_clock::now();

float deltaSec = std::chrono::duration<double>(currentFrame - lastFrame).count();

//Move the camera according to user input

if (display->getIsInFocus())

camera->moveCamera(deltaSec);

//Render the scene

engine->RenderScene(camera, deltaSec);

//Update the display (show the frame we just rendered)

engine->UpdateDisplay();

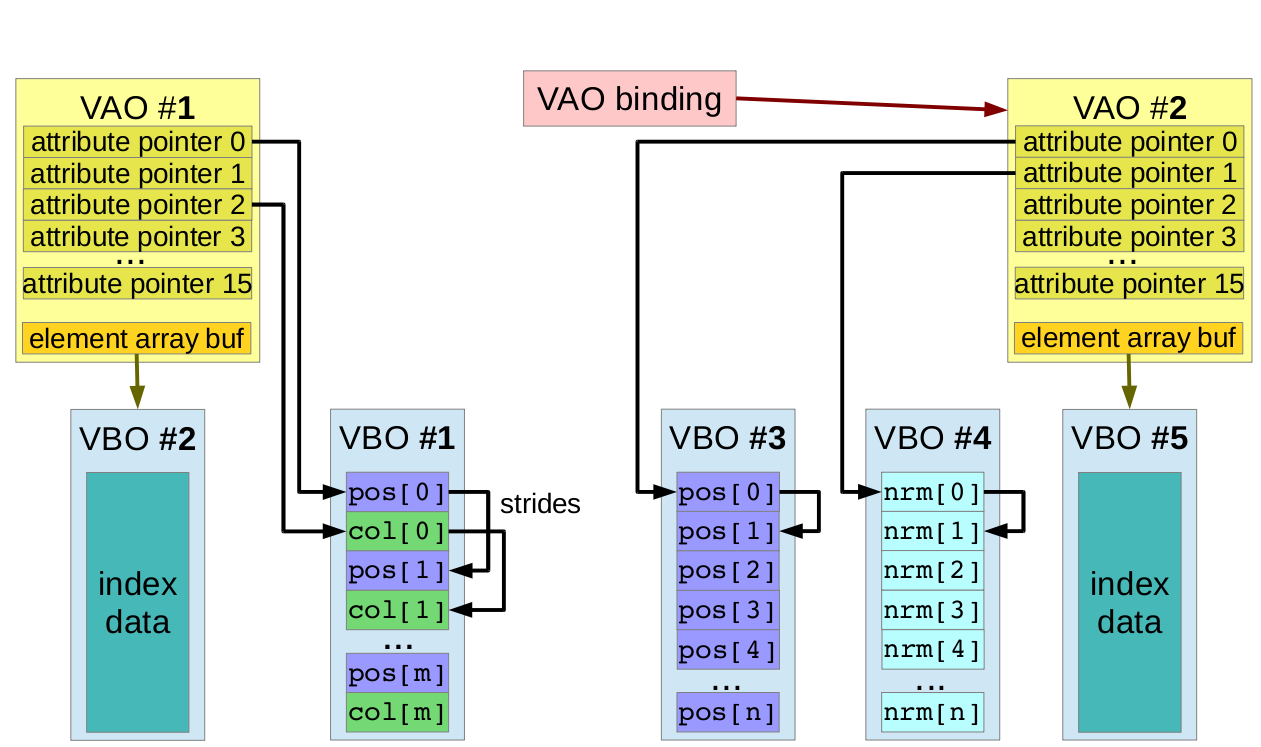
//Store the current frame timestamp so we can take the difference next time and increment //the fps counter

lastFrame = currentFrame;

fps++;  
//=======================================================================================

**Vertex Array Objects and Vertex Buffer Objects**

In order to render a scene, OpenGL requires us to use Vertex Array Objects (VAOs) and Vertex Buffer Objects (VBOs) to display 3D models on screen. Each VAO stores the data for one model and contain 16 attribute lists (labelled 0 to 15) in which we can put information such as the position of a vertex, the color of a vertex or the normal vector of a vertex (used for lighting calculations). VBOs represent the lists of data that we will assign to each attribute list of the VAO. Pointers to these VBOs tell OpenGL how to interpret the data. Every VAO, much like every VBO, has an assigned identifier which we can use to write and manipulate data in it. Here is a typical VAO and VBO structure:

****

**Three-Dimensional Models**

Data for a three-dimensional (3D) model is stored in “.obj” (OBJ) files which represent the object in its entirety. OBJ files contain the position of each vertex in the model in terms of vectors of three components. They also contain the normal vectors associated to each vertex. A normal vector describes the direction a specific face is facing and is always perpendicular to the face’s surface in terms of direction. OBJ files also house a series of texture coordinates as two-component vectors describing which part of the model’s texture a certain face should use. The components of these vectors range from 0 to 1 where [0, 0] is the top left corner of the texture and [1, 1] is the bottom right corner of the texture. Lastly, OBJ files also contain the information for each triangle in the model taking the form of indices. The indices usually come in blocks of three and tell you which vertices form a triangle (or face).

To turn the model data into VAOs and VBOs, I used a simple bit of code that lets me read data out of an OBJ file and turns it into a list of vertex positions, texture coordinates, and vertex normals. I then load each list into a VBO and assign this VBO to the attribute lists of the VAO.

**Entities and Vertex Manipulation Using Matrices**

An entity consists of a 3D model (or mesh) rendered at a specific position in the world. In this case, the world’s origin [0, 0, 0] will be the position of the sun in space. Note that to render 100 entities on screen with the same mesh, we only need to create one VAO with all the necessary attributes. This is because each VAO contains data that describes solely one mesh. But how would we move, rotate, and scale each entity then? The answer is: using matrices. A 4 by 4 matrix can be used to transform any 3D point. By transforming each vertex of the 3D model 100 times with different translations and rotations, we have rendered 100 different entities at different positions using the same vertex data. This type of matrix is called a model matrix. However, we still cannot move our camera to see all of these entities. In OpenGL, the concept of a camera does not exist. So, similarly to the way we used matrices to move our entities, we will use another 4 by 4 matrix to represent the camera. In other words, if we want to move the camera to the left, we will instead move the entire world to the right by using what we call a view matrix. That way, we’ve managed to simulate the effect of a camera. Now that we are able to move around in the world, there is one problem left to solve. The entities we’ve rendered would seem to appear of the same size as one another regardless of their distance from the camera. This is because OpenGL does not have a built-in perspective matrix. A perspective matrix is a matrix that defines the way we flatten 3D objects onto a 2D surface (in this case, the screen) in order for us to see them the same way we would in real life. To create a perspective matrix, all we need is the aspect ratio (1600 / 900 = 16:9) of the window and the horizontal field of view which is usually set to 70 degrees.

**Shader Programs**

Now that we have all three types of necessary matrices, we can tell OpenGL to render all the necessary entities, in our case, the planets and the sun. However, before rendering anything we must create a shader program to tell OpenGL how to render the models. A shader program usually consists of two parts, a vertex shader, and a fragment shader. The vertex shader is run once for each vertex we send it, whereas the fragment shader is run once for each fragment (or pixel) rendered on screen. Note that the output of the vertex shader is always the input of the fragment shader. The fragment shader’s job is to tell OpenGL the final color of each rendered pixel. This shader program will be run on the Graphics Processing Unit (GPU) of the computer, as it is much faster at arithmetic calculations than the Central Processing Unit (CPU). Here is a simple shader program along with comments to help explain how shaders work:

//==================================VERTEX SHADER CODE===================================  
//We tell OpenGL the shader version we’re using

#version 400 core

in vec3 pos; //We take in a three-dimensional vector that represents the position of the

//vertex from attribute list 0 of the VAO

in vec3 color; //We take in a three-dimensional vector that represents the color of the

//vertex from attribute list 1 of the VAO

out vec3 pass\_Color; //We tell OpenGL that we will be sending a vec3 to the fragment

//shader

//These are the 4x4 matrices that will be loaded from the main game code

uniform mat4 projectionMatrix;

uniform mat4 viewMatrix;

uniform mat4 modelMatrix;

//This is the method that is run for each vertex

void main(void){

//We pass the vertex color to the fragment shader

pass\_Color = color;

//We modify the vertex position using matrix multiplication

//Note that we have to make the vertex position a 4-dimensional vector in order to //do so (the multiplication order is also important to get the desired result)

gl\_Position = projectionMatrix \* viewMatrix \* modelMatrix \* vec4(pos, 1.0);

}  
//=======================================================================================

//=================================FRAGMENT SHADER CODE==================================

//We tell OpenGL the shader version we’re using

#version 400 core

//We take in the interpolated color from the vertex shader

in vec3 pass\_Color;

//This is the method that is run for each fragment (pixel)

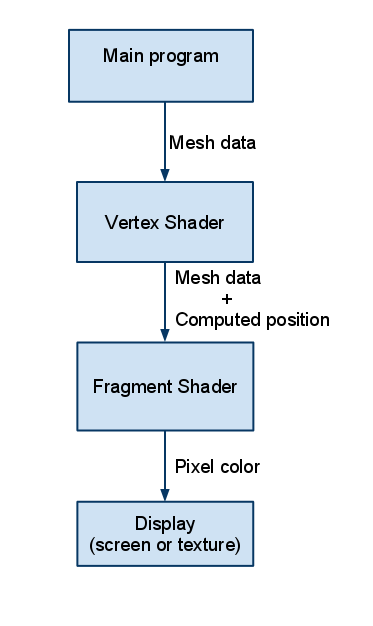
void main(void){

//We set the final color of the fragment we’re rendering

gl\_FragColor = vec4(pass\_Color, 1.0);

}  
//=======================================================================================

Now that we have a working shader program all we have to do to render an entity is to tell OpenGL which VAO to use, bind the correct model, view and project matrices to the shader, and finally, tell OpenGL to render the model data using our shader. Most of the shaders I use in my application are much more complex than this one as they require lighting calculations. There are additional effects such as bloom, antialiasing and others. However, these are the basic concepts when rendering models with OpenGL. Below is a diagram to help illustrate the flow of shaders.



Gravitational Calculations

**Initial Planetary State Vectors**

Now that we can represent a planet as an entity with a specific mesh, we can also make each planet move according to the laws of physics. Each unit in the simulation corresponds to one tenth (1/10) of an astronomical unit (AU). In other words, a hypothetical distance of 10 units represents the distance from the sun to the earth. Before the simulation begins running, we must first set the initial state vectors for each planet. These are an object’s position and velocity vectors in relation to the sun. To do so, we need to take into account both the shape of the orbit, the orbital period, and the time at which each planet passes on their respective orbit’s periapsis. Using data from NASA and the University of Texas about each planet’s orbital elements, we are able to show the ellipse corresponding to the approximate orbit of each planet. The orbital elements of each planet are as follows:

1. Semi-major axis (a):

The semi-major axis of a planet is the average distance at which the planet orbits the sun.

1. Eccentricity (e):

Eccentricity defines the shape of an orbit. An eccentricity of 0 would mean the orbit is perfectly circular. On the other hand, an eccentricity close to 1 would mean the orbit is an infinitely long ellipse almost resembling a line. An eccentricity greater than 1 signifies an open hyperbolic orbit.

1. Inclination (i):

Inclination is the angle between the orbit and the reference plane. In my case, the reference plane would be the plane on which lies the orbit of the earth around the sun. An inclination greater than 90 describes a retrograde orbit.

1. Longitude of ascending node (Ω):

The ascending node is the point at which the orbit crosses the reference plane as the object orbiting goes from under the plane to above the plane. The angle between the reference direction and the ascending node is what we call the longitude of the ascending node.

1. Argument of periapsis (ω):

The argument of periapsis is the angle between the ascending node and the point of the orbit which is closest to the orbit’s center (the sun in our case).

1. Mean anomaly (M):

An imaginary angle the represents the current position of the orbiting object with respect to the periapsis. This angle has values between 0 and 360 degrees and increases by 360 degrees every orbit. A mean anomaly of 0 would mean the orbiting object is currently at its periapsis.

1. Mean motion (n):

The mean motion is the rate at which the mean anomaly changes over time. It is equal to 2π radians divided by the length of a period.

1. Time of periapsis:

This is simply a reference to help us calculate the mean anomaly of a planet at a specific time using its orbital period.

Using these elements, we are able to determine the orbit of each planet with respect to the ecliptic plane. From the mean anomaly of the object, the eccentric anomaly (E) can be calculated using Kepler’s equation:

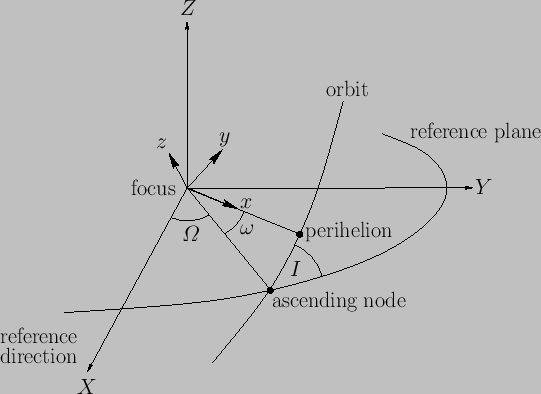
Note that this equation cannot be solved analytically, so we have to rely on the recursive Newtonian approximation of the function:

Where:

We begin with the estimation E0 = M and calculate Ei+1 until has zeros in the first 4 decimal places. We can now calculate the orbital position as a two-dimensional vector where the positive x direction points towards the periapsis using the following formulas:

Where Px and Pz are the respective components of the position vector. Lastly, we create a rotation matrix in OpenGL by starting with a 4 by 4 identity matrix, rotating it by ω about the y-axis, by i about the x-axis, and again by Ω about the y-axis. By multiplying this matrix on the left-hand side of the two-dimensional position vector, we obtain the actual three-dimensional positional vector. The same applies to velocity. A two-dimensional velocity vector where the positive x direction points towards periapsis is obtained with the following formulas:

As mentioned previously, this vector is then rotated to match the orbit’s inclination and other rotations. The following image helps explain the rotations made to convert these vectors to ecliptic coordinates:



You’ll note that, in the illustration, the z and y axes have been interchanged. OpenGL uses, by convention, the y-axis as the up and down direction, therefore making this a necessary conversion. It is much simpler for us to interchange these axes rather than to rotate the entire coordinate system in OpenGL.

**Position and Velocity Updates Based on Newton’s Gravitational Law**

Newton’s laws of motion state that the total force exerted on an object is equal to the product of its mass and acceleration like so:

Note that both force and acceleration carry directions, and thus, inversely, we can also determine an object’s acceleration from the force applied to it divided by its mass:

Assuming the only force acting on each object is the gravitational force, we can determine an object’s acceleration using the following formula:

Where G is the gravitational constant equal to 6.67 x 10-11 N m2 / kg2, M is the mass of the object pulling on the original one (in our case, the sun), and d is the distance between both objects in meters. We will use M = 2 x 1030 kg as that is the mass of our sun. The result is in m/s2 which will be converted into (1/10) AU / year2, as these are the units I am using for the simulation. Similarly, for each planet, we will calculate, using the same formula, the acceleration caused on it by the closest object and add it to the one from the sun. Lastly, every time we loop to update the positions of each of the planets (called Entities in the code), we use the time passed since the last calculation to scale and add the result to the entity’s current velocity. We do the same for position except we add to it the scaled velocity instead of acceleration. Here is a snippet of the code:

//=======================================================================================  
//Magnitude of acceleration caused by the sun in (1/10) AU/year^2

double force = ((6.67e-11 \* 1.99e+30) / pow(ent->getPosition().length() \* 1.5e+10, 2)) \* pow(86400 \* 365.25 \* Engine::getYearsPerSecond(), 2) / 1.5e+10;

//Direction of acceleration (the sun is the origin)

ent->acceleration = (-1.0 \* force) \* Vector3f(pos.x, pos.y, pos.z).Normalize();

//Find the closest entity

Entity \* closest = getClosestEntity(ent);

//This line is to prevent crashes

if (closest != nullptr) {

//Magnitude of acceleration caused by the closest entity in (1/10) AU/year^2

double entForce = ((6.67e-11 \* closest->mass) / pow((closest->getPosition() -   
ent->getPosition()).length() \* 1.5e+10, 2) \* pow(86400 \* 365.25 Engine::getYearsPerSecond(), 2) / 1.5e+10);

//Direction of acceleration (the sun is the origin) and adding it to the one from //the sun

ent->acceleration = ent->acceleration + entForce \* (closest->getPosition() - ent->getPosition()).Normalize();

}

//Add acceleration to velocity ((1/10) AU/year)

ent->velocity = ent->velocity + deltaSec \* ent->acceleration;

//Add velocity to position ((1/10) AU)

ent->setPosition(ent->getPosition() + (deltaSec \* ent->velocity));  
//=======================================================================================

**Obtaining Orbital Trajectories from State Vectors**

In order to able to calculate an object’s predicted orbit, one must first know its state vectors. From those vectors, we begin by calculating the object’s angular momentum vector (h) which is the cross product of both state vectors:

And then the node vector (n):

As well as the eccentricity vector (e):

And, lastly, the specific mechanical energy (E):

Where μ is the sum of the standard gravitational parameters of both bodies (μ = G(M1 + Msun)) in N m2 / kg, r is the positional vector in meters, and v is the velocity vector in meters per second. From there, the eccentricity of the orbit is simply the magnitude of the eccentricity vector and the rest of the orbital elements can be found using the following formulas:

(Note that a must be in AU)

All formulas are to be used with standard SI units with the exception of the formula for the period (in years) which only works because we are calculating orbits around the sun and the semi-major axis must be astronomical units. From this information, drawing orbits of added asteroids or planets is made possible. It is also worth mentioning that eccentricities greater than 1 produce open hyperbolic orbits and therefore significantly impact the calculation accuracy since their semi-major axis has a value of infinity. I do not consider these orbits in my application as tracing them is a nightmare and is not in the scope of my project. As the simulation stumbles upon one of these, it will likely crash. Following, there is once again a code snippet.

//=======================================================================================

float dist = pos.length();

float velocity = vel.length();

float mu = 6.67e-11 \* (mass + 2e+30);

Vector3f hVec = Vector3f::Cross(pos, vel);

Vector3f node = Vector3f::Cross(Vector3f(0, 1, 0), hVec);

Vector3f eVec = (1.0f / mu) \* ( (pow(velocity, 2) - (mu / dist)) \* pos - (Vector3f::Dot(pos, vel) \* vel));

float e = eVec.length();

float mechanicalEnergy = (pow(velocity, 2) / 2.0f) - (mu / dist);

float a = -mu / (2.0f \* mechanicalEnergy);

if (e < 1) {

float i = acos(hVec.y / hVec.length()) \* 180.0 / PI;

float ascendingNode = acos(node.x / node.length()) \* 180.0 / PI;

float periapsisArg = acos(Vector3f::Dot(node, eVec) / (node.length() \* e)) \* 180.0 / PI;

float meanAnomaly = acos(Vector3f::Dot(eVec, pos) / (e \* dist)) \* 180.0f / PI;

//Prevent crashes

if (meanAnomaly != meanAnomaly)

meanAnomaly = 180.0f;

float period = sqrt((4 \* pow(PI, 2) \* pow(a, 3)) / mu) / (24 \* 3600);

if (eVec.y < 0)

periapsisArg = 360.0f - periapsisArg;

if (Vector3f::Dot(pos, vel) < 0)

meanAnomaly = 360.0f - meanAnomaly;

if (hVec.x < 0)

ascendingNode = -ascendingNode;

//Print result to console

std::cout << "Semi-Major Axis : " << a / 1.5e+11 << " AU" << std::endl;

std::cout << "Eccentricity : " << e << std::endl;

std::cout << "Inclination : " << i << std::endl;

std::cout << "Mean Anomaly : " << meanAnomaly << std::endl;

std::cout << "Ascending Node : " << ascendingNode << std::endl;

std::cout << "Argument of Periapsis : " << periapsisArg << std::endl;

std::cout << "Orbital Period : " << period << " days" << std::endl;

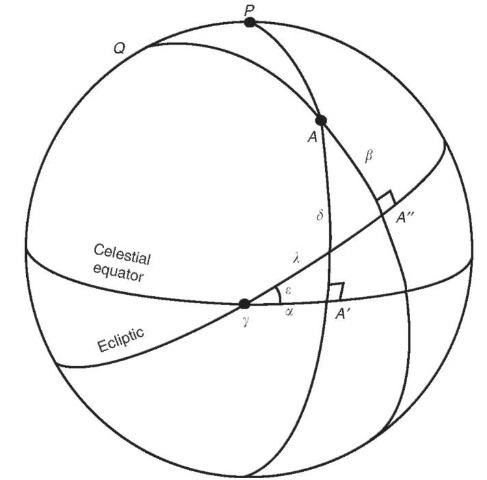
tempOrbit = engine->CreateOrbit(Vector3f(1, 1, 1), engine->origin, a / 1.5e+10, e, i, periapsisArg, ascendingNode, period, new Time(1, 1, 2000, 12, 0));

tempMeanAnomaly = meanAnomaly;

}  
//=======================================================================================

**Rotational Axes of Planets**

Planets rotate along an axis whose north pole is determined by the right-hand rule. Each planet has a different rotational axis, some almost vertical with respect to the ecliptic plane. To find the direction of a planet’s north pole, I used data from a scientific report which gives values of the right ascension (α) and declination (δ) of the different planetary north poles. In order to properly simulate planetary rotation, we must calculate a planet’s north pole vector. The report gives coordinates in equatorial form. Since the simulation coordinates are based on the ecliptic plane, we must convert the equatorial coordinates into ecliptic coordinates. To do so, we rotate a unit vector pointing upward [0, 1, 0] by α about the y-axis, by 90° - δ about the z-axis, and by the planet’s obliquity to its orbit (ε) about the x-axis. Then, to simulate rotation, we simply rotate the planet about the north pole vector at the appropriate speed every frame. Following is an illustration to help explain the conversion.

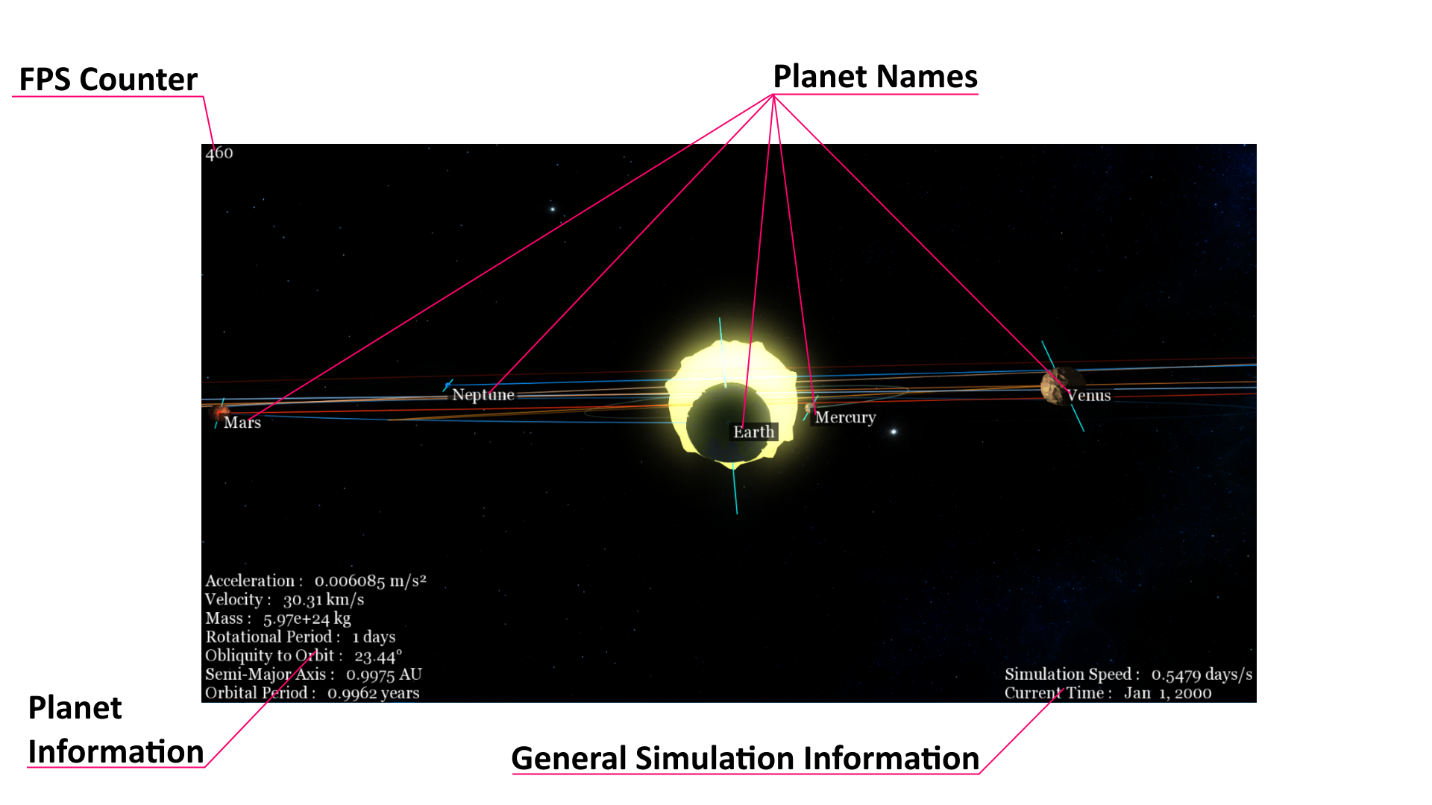


On the picture, λ is the ecliptic longitude and β is the ecliptic latitude which I haven’t covered because we use rectangular (in terms of vector components) ecliptic coordinates instead.

The Application

**User Interface**

To view the simulation, simply start the program. You are greeted with a top-down view of the solar system. In the top left corner of the screen is displayed the Frames Per Second (FPS) counter of the simulation. That number simply indicates how many times the simulation is being refreshed and rendered each second. The higher the FPS, the more fluid the simulation is. Higher refresh rates produce more accurate simulations as position and velocity are being recalculated faster. This does not however have an impact on the simulation speed. It simply serves as a relative measure of accuracy. The bottom right corner of the screen contains the general information about the simulation. It consists of the simulation speed in days per second, and of the current hypothetical time of year. Note that the simulation is set to start on the J2000 epoch date which is January 1st, 2000 at noon. The keys F1 through F8 snap the camera to the different planets of the solar system. When switching the camera to a planet, information concerning its orbit is displayed in the bottom-left corner of the screen. These statistics include velocity, acceleration, and the current computed semi-major axis and period. Below is an image showing the different parts of the user interface.



**Keyboard Controls**

Upon launching the program, WASD keys may be used to move the camera along with Shift and Space to control altitude. Keys F1 through F8 snap the camera to their respective planet (Mercury to Neptune). While the camera is snapped to a planet, you may use the scroll wheel to scroll away or toward the planet. F9 allows you go back into free roam mode where you can move around freely. Q, T, G, M and H allow you toggle different elements of the user interface. C puts you in edit mode where you can place asteroids and give them an initial velocity with the UIOJKL keys. Z pauses and resumes the simulation. F lets you view the planets to scale. I’ve appended, near the end of this section, a table with the full list of controls and what they do.

**Configuration File**

The application contains a configuration file named “config.ini”. It contains important values used to initialize the application and looks something like this:

;========================================================================================  
[Frame]

width=1600

height=900

[Settings]

msaa=8

fov=70.0

asteroidMass=11e+27  
;========================================================================================

Currently, these are the only settings the user can change. To change a setting, simply open the configuration file with a text editor, change the corresponding value, save the file, and make sure to format it the same way as above. Restart the application after you’ve made changes to the file. The width and height settings refer to the size of the application window in pixels. MSAA (Multi Sampled Anti-Aliasing) refers to the number of samples per pixel the application should test when rendering things to the screen. This can be set to 0, 2, 4, or 8 samples. Intermediate values will cause a crash (1 is interpreted as 0 and not all computers handle MSAA). The FOV (Field of View) setting changes the size of the camera’s field of view in degrees. Lastly, the asteroid mass is the mass in kilograms to use for the asteroids’ gravitational calculations.

**List of Key Bindings**

|  |  |
| --- | --- |
| **Key(s)** | **Effect** |
| W, A, S, D, Shift, Space | Move around in space when in free camera mode: -W lets you move forward -A lets you move left -S lets you move backward -D lets you move right -Shift lets you move down -Space lets you move up |
| F1-F8 | Snap the camera to the corresponding planet  in order of distance from the sun: -F1 lets you snap to Mercury -F8 lets you snap to Neptune |
| F9 | Return to free camera mode |
| Q | Toggle showing planetary rotational axes |
| Z | Pause and Resume the simulation |
| X | Delete all objects added through edit mode |
| C | Place an asteroid: -Begin placing an asteroid by pressing C (edit mode) (circular orbit by default) -Move the camera to place the asteroid in the desired position -Press Enter to place the asteroid -Use UJIKOL keys to give the asteroid an initial velocity -Press Enter to add the asteroid to the simulation |
| F | Toggle planet scale |
| G | Toggle showing main elements of the user interface: -FPS Counter -General Simulation Information -Planet Information |
| H | Toggle showing planet names |
| T | Toggle rendering all text |
| R | Reset the simulation to J2000 |
| U, J, I, K, O, L | Use to modify an asteroid's initial velocity (see use of key binding C): -U lets you increase the x component -J lets you decrease the x component -I lets you increase the y component -K lets you decrease the y component -O lets you increase the z component -L lets you decrease the z component |
| M | Toggle showing original planetary orbits |
| +, - | Increase and decrease simulation speed (shown at the bottom right) |
| Escape | Free mouse cursor (click to control camera again) |

**Software Prerequisites**

The application runs on Windows operating systems of version 7 and later. In order to run the software, one must first ensure that they have a GPU that supports OpenGL version 4.0 at the very least. Furthermore, the Visual C++ Redistributable 2015 Package is a required installation before running the simulation. It can be found on the official Microsoft website. Lastly, make sure that the “res”, “shaders”, and “fonts” folders are all in the same parent folder as the executable (.exe).

Conclusion

In conclusion, the software runs relatively smoothly on most computers. When using speeds over 1000 days/s Mercury tends to immediately fly off into the distance. I believe this is due to the way I’ve designed my position and velocity calculations. Because, at high speeds, Mercury’s velocity is very fast, and the simulation does not work continuously (we update positions and velocities a certain number of times per second), the application will not register the change in acceleration properly over the distance it has traveled since the last frame. That causes the planet to fly off into space. The number of FPS also influences these calculations as ideally, they should be done infinitely many times per second. Thus, making the software was certainly a challenge as it required some clever problem-solving techniques.

Bibliography

1. ARCHINAL, B. A. et al. “Report of the IAU Working Group on Cartographic Coordinates and Rotational Elements: 2009”. Celestial Mechanics and Dynamical Astronomy. Volume 109, Issue 2, pages 101-135. February 2011.
2. DE VRIES, Joey. “Learn OpenGL”. Accessed on May 14th, 2018. <https://learnopengl.com>
3. FORD, Dominic. “3D Diagram of the Solar System”. Last updated on May 14th, 2018. Accessed on  
   May 14th, 2018. <https://in-the-sky.org/solarsystem.php>
4. “Orbital elements”. University of Texas. Accessed on May 14th, 2018. <http://farside.ph.utexas.edu/teaching/celestial/Celestialhtml/node34.html>
5. “Planetary Fact Sheet - Metric”. National Aeronautics and Space Administration. Accessed on May 14th, 2018. <https://nssdc.gsfc.nasa.gov/planetary/factsheet/>
6. “Sol Planetary System Data”. Princeton University. Accessed on May 14th, 2018. <https://www.princeton.edu/~willman/planetary_systems/Sol/>
7. “Determining orbital position at a future point in time”. Stack Exchange, Space Exploration. Accessed on May 14th, 2018. <https://space.stackexchange.com/questions/8911/determining-orbital-position-at-a-future-point-in-time>
8. “How to programmatically calculate orbital elements using position/velocity vectors?”. Stack Exchange, Space Exploration. Accessed on May 14th, 2018. <https://space.stackexchange.com/questions/1904/how-to-programmatically-calculate-orbital-elements-using-position-velocity-vecto>
9. “How to get the axial tilt vector(x,y,z) relative to ecliptic”. Stack Exchange, Astronomy. Accessed on May 14th, 2018. <https://astronomy.stackexchange.com/questions/18176/how-to-get-the-axial-tilt-vectorx-y-z-relative-to-ecliptic>