MSIM 742

Synthetic Environments

Assignment Three

Mark Brosche

MSVE Master of Engineering Student

mbros006@odu.edu

(407)-325-2818

Distance Graduate - Outside Virginia

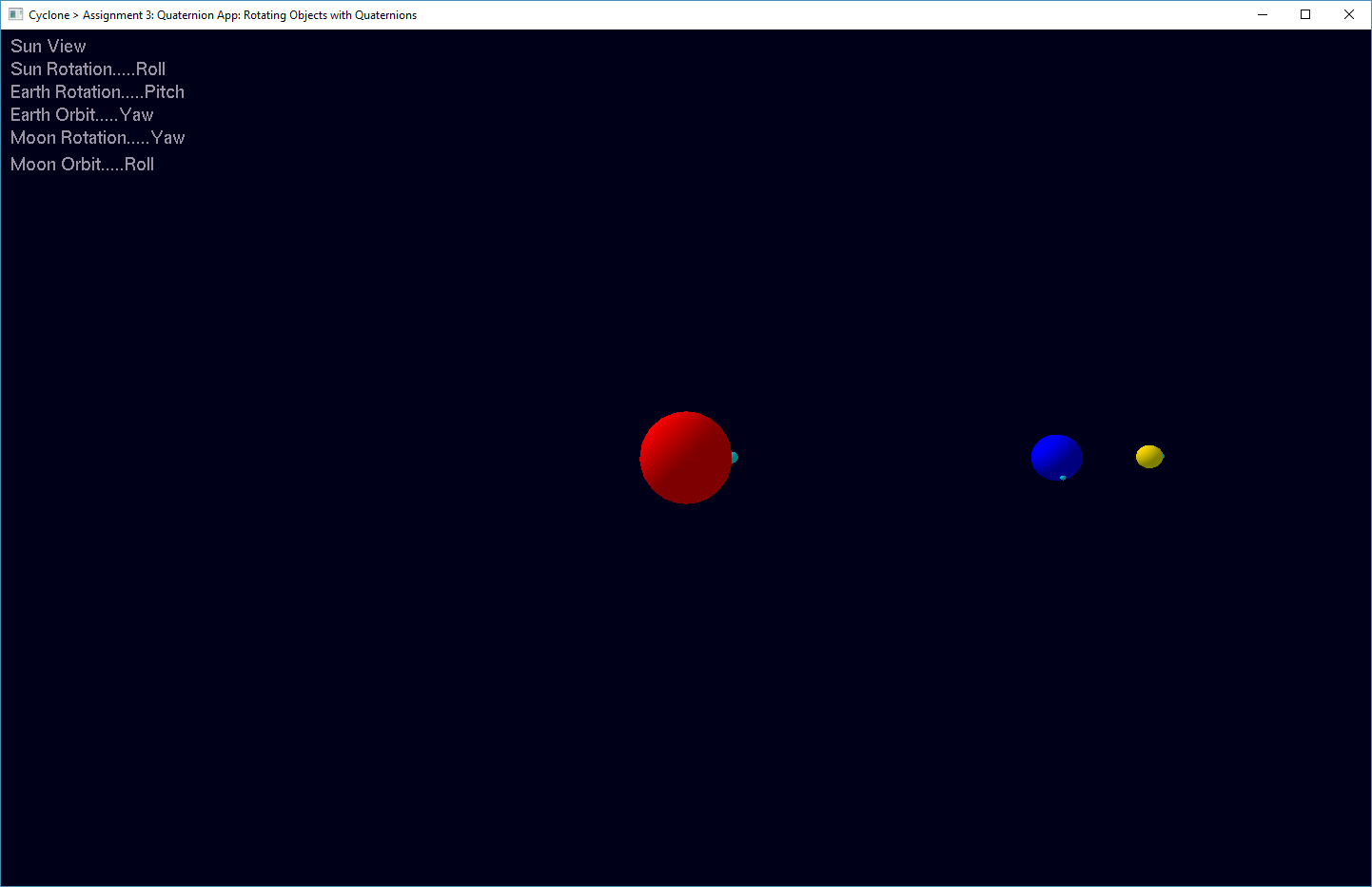
April 15, 2018

## Introduction

For this assignment, the goal was to produce an example of quaternion mathematics by simulating the orbits and spins of 3 bodies in space, referenced as the sun, earth, and moon, thought they do not represent an attempt to accurately simulate those specific bodies. Using the Vector3, Matrix4, and Quaternion classes defined in the Cyclone physics engine is essential to success with this project.

A video of the application in action can be found [here](file:///C:\Users\markb\Documents\MSIM742\MSIM-742_A-3_Brosche-Mark\MSIM-742_A-3_Brosche-Mark_Video.mp4).

The application executable can be run [here](file:///C:\Users\markb\Documents\MSIM742\MSIM-742_A-3_Brosche-Mark\Debug\GamePhysicsApp.exe.lnk).



**Figure 1. Application Starting Point**

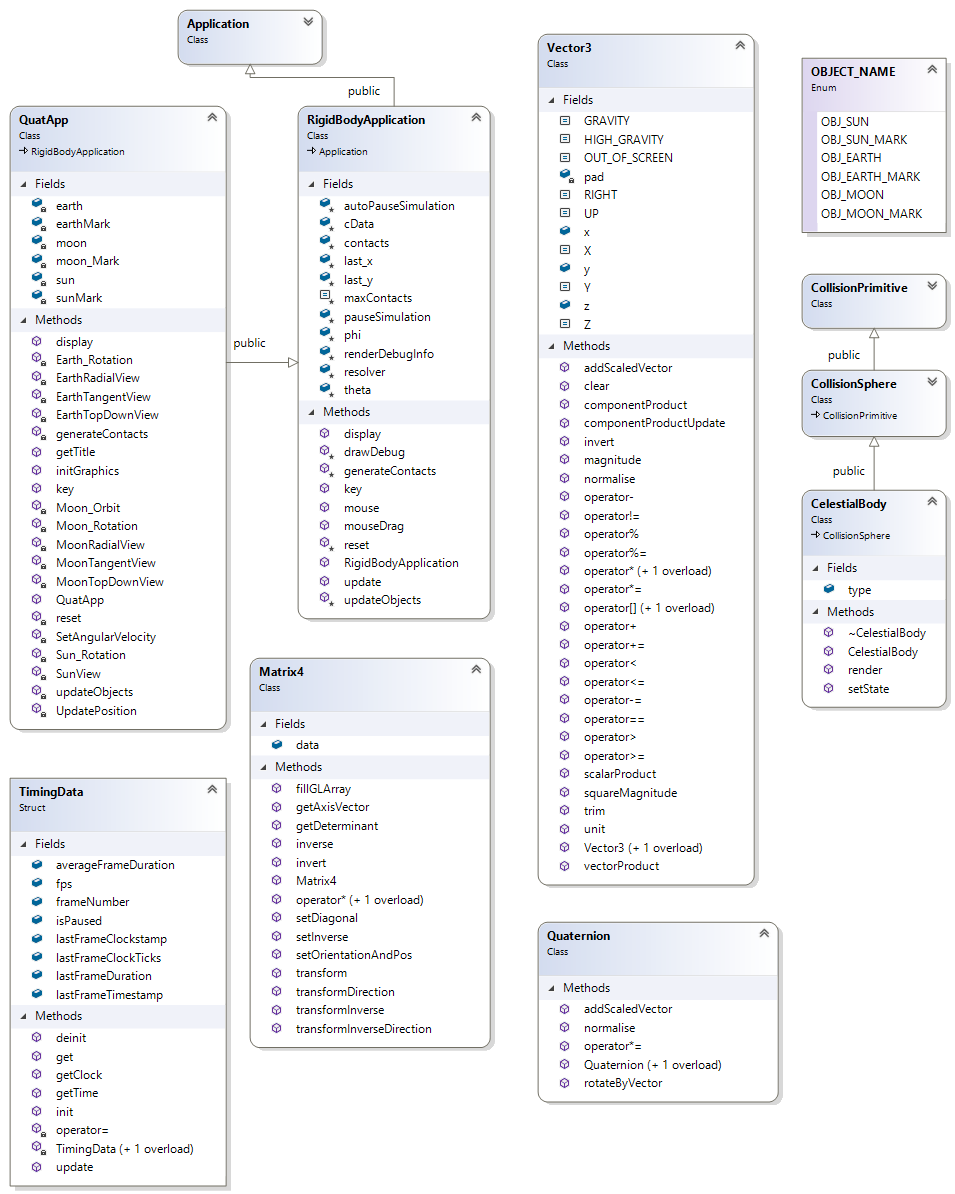
## Program Design

This section identifies the files, functions, models and variables used in the project. Also contained are instructions for running and controlling the application. Since this project was built using the “BigBallistic” demo, only the new additions and changes will be discussed. 3D models were imported using the OBJ parser provided in MSIM 541 with classes and methods outlined below. Another inclusion from MSIM 541 is the Utility.cpp which contains methods for computing rotation matrices and printing strings.

|  |  |
| --- | --- |
| Source | Headers |
| Utility.cpp  QuatApp.cpp  App.cpp  Core.cpp  GamePhysicsApp.cpp | Utility.h  Core.h  Cyclone.h |

### Key Variables & Functions:

|  |  |
| --- | --- |
| Class: QuatApp - The main demo class definition. | |
| Variables:  Public:   * Sun[]: CelestialBody * sunMark[]: CelestialBody * earth[]: CelestialBody * earthMark[]: CelestialBody * moon[]: CelestialBody * moon\_Mark[]: CelestialBody | **Description:**   * Holds the position, orientation, and renderer for the sun, and other spheres, based on the collisionSphere class. |
| Methods:  Public:   * QuatApp(): QuatApp * Reset(): virtual void * generateContacts(): virtual void * updateObjects(cyclone::real duration): virtual void * SetAngularVelocity(cyclone::Vector3 &axis, float period, double \* speed, float \* selfCos, float \* selfSin, cyclone::Quaternion \* frameQuat): void * UpdatePosition(OBJECT\_NAME obj\_name, cyclone::Quaternion \* frameQuat, cyclone::Quaternion \* accumulatedQuat, cyclone::Vector3 \* parentObjPos, cyclone::Vector3 \* oldPos, cyclone::Vector3 \* newPos): void * Sun\_Rotation(unsigned char rotation); * Earth\_Rotation(unsigned char rotation); * Moon\_Rotation(unsigned char rotation); * Moon\_Orbit(unsigned char rotation); * SunView();void * EarthRadialView();void * EarthTangentView();void * EarthTopDownView();void * MoonRadialView();void * MoonTangentView();void * MoonTopDownView();void * Display(): virtual void * Key(unsigned char key): virtual void | **Description:**   * Creates a new demo object. * Resets all objects to default states. * Not used in this project. * Updates objects at the specified duration * Determines the per frame delta for each object. * Updates each object by the per frame delta set by the angular velocity and accumulates the updates in each object’s quaternion accumulator. * Sets the sun’s rotation (x/y/z) * Sets the earth’s rotation (x/y/z) * Sets the moon’s rotation (x/y/z) * Sets the moon’s orbit about earth (x/y/z) * Set’s the camera, focus, and upVector to view the entire system. * Set’s the camera, focus, and upVector to view the earth pointing toward the sun. * Set’s the camera, focus, and upVector to view the earth along the tangent to its orbit. * Set’s the camera, focus, and upVector to view the earth from directly above. * Set’s the camera, focus, and upVector to view the moon pointing toward the earth. * Set’s the camera, focus, and upVector to view the moon along the tangent to its orbit. * Set’s the camera, focus, and upVector to view the moon from directly above. * Refreshes the scene every frame. * Sets listener for keyboard inputs |
| Global Variables:   * double PI\_2\_val * enum OBJECT\_NAME * int viewMode * cyclone::Vector3 camera, focusPnt, upVector * float sunSize, sRefSize, ethOrbit, ethSize, eRefSize, moonOrbit, moonSize, mRefSize * float speedAdj, sunRot, ethRot, ethOrbitPeriod, moonRot, moonOrbitPeriod * double sunSpeed, ethSpeed, ethOrbitSpeed, moonSpeed, moonOrbitSpeed * string sun\_r, earth\_r, e\_orbit\_r, moon\_r, m\_orbit\_r * float sunCos, sunSin, ethCos, ethSin, ethOrbitCos, ethOrbitSin, moonCos, moonSin, moonOrbitCos, moonOrbitSin * cyclone::Vector3 sunColor, earthClr, moon\_clr, refColor * cyclone::Vector3 sunCoords, sunAxis, sunMarkStartPos, sunMarkPos * cyclone::Quaternion sunAccumQ, sunFrameQ * cyclone::Vector3 earthStartPos, earthPos, * earthAxis, ethMarkStartPos, ethMarkPos * cyclone::Quaternion ethAccumQ, ethFrameQ; * cyclone::Vector3 ethOrbitAxis * cyclone::Quaternion ethOrbitAccumQ, ethOrbitFrameQ; * cyclone::Vector3 moonStartPos, moonPos, * moonAxis, moonMarkStartPos, moonMarkPos * cyclone::Quaternion moonAccumQ, moonFrameQ; * cyclone::Vector3 moonOrbitAxisStart, moonOrbitAxis * cyclone::Quaternion moonOrbitAccumQ, moonOrbitFrameQ; | * **radians in a circle** * **enumerations for the bodies** * **holds the current active view mode** * **hold the values for setting the application camera** * **body sizes** * **body rotational and orbital periods** * **body speeds** * **body rotation strings for printing to the screen** * **holds sine and cosine values for computing angular velocity** * **holds color vectors for each body** * **sun coordinates and rotation axis, and reference mark position holder** * **sun frame quaternion and accumulator quaternion** * **earth coordinates and rotation axis, and reference mark position holder** * **earth frame quaternion and accumulator quaternion** * **earth orbit axis vector** * **earth orbit quaternion accumulator and frame quaternion** * **moon coordinates and rotation axis, and reference mark position holder** * **moon frame quaternion and accumulator quaternion** * **moon orbit axis vector** * **moon orbit quaternion accumulator and frame quaternion** |



**Figure 2. The class diagram of the “QuatApp” Application.**

**(Consumed attributes from Cyclone marked with a star).**

### Instructions to run code:

Run **“GamePhysicsApp.exe”** in the **/Debug** folder

Inputs are as follows:

* Q/W/E – switch the self-rotational mode of the sun (Yaw/Pitch/Roll).
* A/S/D – switch the self-rotation mode of the earth (Yaw/Pitch/Roll).
* Z/X/C – switch the self-rotational mode of the moon (Yaw/Pitch/Roll).
* B/N/M – switch the orbital-mode of the moon (Yaw/Pitch/Roll).
* 1 – Sun view
* 2 – Earth radial view
* 3 – Earth tangent view
* 4 – Earth top-down view
* 5 – Moon radial view
* 6 – Moon tangent view
* 7 – Moon top-down view
* R – Reset (resets gun and targets and applicable variables to their starting states.
* ESC – quit application.

## Results

I have organized my results with this project with respect to the tasks section of the assignment document.

### 3.1 Transform Matrix M

To rotate and orbit objects around each other it becomes important to transform the coordinate system and direction of a parent object into those of their children and children’s children. To do this in this assignment, I used the matrix4 class from the cyclone engine to set rotation and position (See “UpdatePosition” method). The matrix is populated with the accumulated quaternion changes from the parent systems and the location of the parent object. For example:

The moon’s local quaternion is now in the global system.

The transformation matrix is set with the quaternion and the parent position using the setOrientationAndPos method (cyclone doesn’t include the bottom row of a 4x4 matrix).

The transformation matrix also transforms directions, which is relevant to the orbital axis of the moon as it is dependent on the location of the earth relative to the sun.

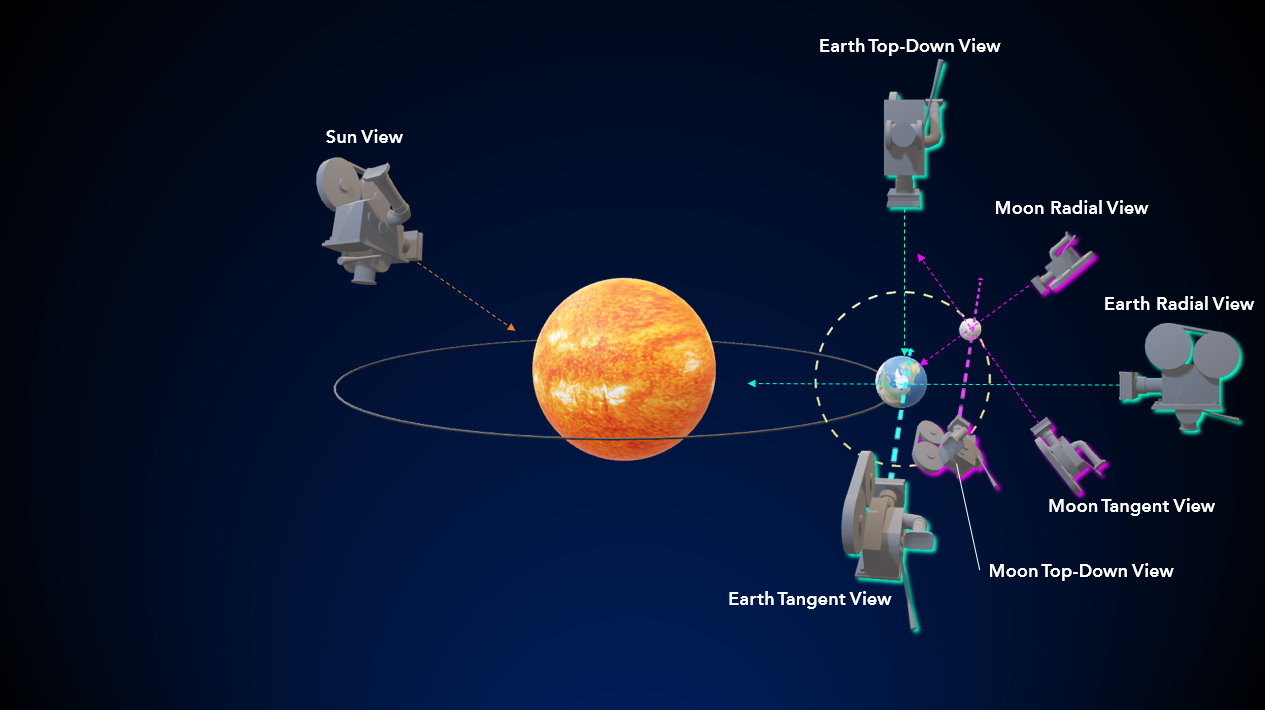
### 3.2 Relativity of Rotations

To build the application, it was imperative to understand how to apply quaternions to an object as well as how to define the object relationships. Key to this was a hint provided in the assignment worksheet, that a negated angle of rotation is needed to properly transform between parent and child since all objects are rotating and thus the dependencies of the orbiting should not follow the rotations of the parent. Exhibiting this, the SetAngularVelocity method applies a negative speed to the rotations of all the bodies.

### 3.3 Rotation Modes and Viewing Modes

Rotation modes and camera views were the las things to be properly set up in my implementation. For rotation modes, I set up functions for each of the bodies with switch statements to alternate between axes of rotation. It took some trial and error to learn that the accumulator quaternion needed to be cleared each time rotation was changed, which also meant the reference mark will ‘jump’ instantaneously from its current position on a body to the starting point defined in the rotation function when toggled.

The camera viewing angles were something I needed to visualize with a diagram before I could begin to understand how to compute them. My initial efforts led me to try to simply following the position of the body with an oversimplified concept of ‘up’. But failures and then diagramming the layout based on second reading of the instructions helped me to understand the need to use vector cross products to find the desired camera angles and up-vectors. In experimenting with views, I deviated from the recommended 10x distance from the subject the case of the earth, because I found better confirmation that the views were correct when I could see all three bodies and their rotations.



**Figure 3. Camera positions and orientations at a given snapshot in time.**

### 3.4 Rotational Speed (Angular Velocity Magnitude)

Angular velocity is what essentially becomes the amount of change in orientation and position per frame of each object. Originally, following the use of the collision sphere class of a Rigidbody application like the previous assignment, I had tried to impart rotation with the setRotation member function of the Rigidbody class. This of course did not work. I determined that I couldn’t use most of the Rigidbody class member functions and seeing that the variables given by the professor in the assignment were global, I figured that angular velocity should set a fixed increment of delta that an accumulator quaternion stores against the starting position of the object, which is fixed by calling the SetAngularVelocity function.

## Conclusion and Discussion

I experienced a great deal of confusion with this assignment at the outset, but review of the chapters and careful repeated scrutiny of the assignment wording and various YouTube tutorials on quaternions helped me to get past the hang-ups with parent-child relationships and the matrix math and the need for normalization. Despite the initial confusion, I feel I was able to make a really well-written application.

Accomplishments:

* Getting it to work.

Difficulties:

* Everything about this assignment was difficult. Mostly I got hung up on the words of the assignment that didn’t start making sense to me until I had tried and failed several times to write anything functional. The slides on rotations were very helpful in identifying Vector3, Matrix4 and Quaternion class functions to use.

Possible Improvements:

* I believe the view methods could be condensed further into one for each body, given more time.
* I also think that given more time, more of the member attributes and functions of the CelestialBody collisionSphere class could have been used instead of the global quaternion accumulators.
* On an assignment level, I would try to organize/compartmentalize the presentation of information to the student better. Overall, reading it and understanding the task at all was a challenge.