Concordia University

COMP 371: Computer Graphics

Sunday, August 16, 2015

Final Project Report

Space Shooter 5000

Team 6

Ajayveer Aujla 26595863

Ahmed Dorias 26649874

Ian Kelley 26507549

Simon Labute 40012570

Mark Massoud 26599486

Project Overview

The idea of this computer graphics project is a space shooter simulator. The game structure and game logic are quite minimalistic, which allowed for greater focus on the graphics related features of the project.

The simulator is a first person shooter set in outer space. The background itself consists of rich visual effects, littered with stars and debris of space, as well as other enemy ships in the distance and asteroids of various textures and sizes. The asteroids are programmed to bounce off of each other, along with any projectiles shot at them, which adds for extra visual effect. This was done by implementing physics to determine the respective trajectory of the asteroids after a collision.

The projectiles can be voluntarily fired by the player by left clicking on the mouse. Holding down the left button engages the spaceship in rapid firing, while clicking the button results in projectiles being shot one at a time. The player is able to shoot projectiles anywhere in the world, specifically towards the asteroids, which would result in a collision between the two if the user timed and aimed the shot appropriately. Enemy spaceships also fire projectiles throughout the world, and towards the asteroids.

The player is free to roam around the world in the spaceship, with the scope following the view of the ship. The camera is able to move around the spaceship in a circular motion by pressing 2 on the keyboard. This allows the user to explore a 360° view of the world around the spaceship, along with viewing the spaceship from all angles.

The main focus of this project was the graphics being as scenic as possible, with all of the elements being tied together. The dark background with the scattered lights of stars, the different sizes and textures of the many asteroids, their collisions between each other and the projectiles; all of this in the background, as well as the different ships and projectiles shooting around, all from the view of the main spaceship. All of these visual effects captures the many different light sources and textures, resulting in a graphically pleasing scene.

Initially, the expectations from this project was to have a game comprised of four screens, each with a wave of asteroids on a trajectory towards the player. The user would focus on and magnify one of the four screens with the aim to destroy the incoming asteroids based on how close the asteroids are. The player has three lives in the game, and would lose one life every time an asteroid hits them due to the user failing to destroy the asteroid with projectiles. Once the three lives are over, the game is over and the user must start a new game. The plan was to have different types of projectiles, as well as scope that would be used as a zoom to assist in aiming the projectiles at the asteroids.

The final project was similar in some aspects of the proposed project, but different in others. The space theme with the scenic background, a main ship, asteroids, and the shooting aspect of the proposal was kept in the final project. Ultimately, it was decided to create the project as a simulator, rather than a game, which removed the three lives aspect of the project, the score tracking, etc. Enemy spaceships were added into the background that shoot projectiles towards the main spaceship. Furthermore, one type of projectile, rather than having multiple types, was settled on. Shooting the projectile out of the main ship is dependent on the user, as well as the speed it is fired out of the ship.

It was decided not to include the scope, and rather than having four screens, the idea to project the entire project on one screen was kept. We also kept the camera angle specifically focused on the main ship, and gave one option to circle the camera, still focusing on the ship, for a different angle.

Methodology

Major Features**:**

- Cameras:

  The framework initially provided the implementation for a first person camera and a static camera. Derived from the first person camera was the third person view which placed the camera at the radius of a sphere always looking towards the center where the spaceship was located.

  Adding to the third person view controlled by the user, a scenic third person camera was incorporated to follow a track where the camera movement controls are

disabled. At first this was set on an ellipse. Later on, an idea spawned for it to traverse across a randomly generated path using b-splines.

We randomize the ellipse every 2 cycles around the camera. On each generation of a new ellipse, we check for intersection with the sun (at the origin of the world). This demonstrates our ability to check for our camera curve intersecting with objects despite not having time to implement this feature for all objects. To view this feature, position the main character on the side of the sun and switch to elliptical view and allow it some time to adjust.

- Projectiles:

  The Projectiles were placed inside a container (vector of Projectiles) within the Spaceship class. They are encapsulated within that class and as such the creation, modification, and destruction details lie within the appropriate class.

  Since collision detection can occur between projectiles and other objects (asteroids, planets, etc.), it was not ideal to place the collision detection code within this class. This will be discussed below in the collision detection section.

  Upon seeing the successful implementation of the Asteroid System (derived from the Particle System), it was suggested to utilize the same efficient method for the Projectile system as it was very efficient. This idea spawned only because issues existing with the existing system. However, those were resolved and the idea to utilize the Particle System as inspiration for a new Projectile system was abandoned. Each respective system worked well and there was no need to refactor this one in the end.

  The projectiles can be shot individually with single clicks of the mouse, or in a rapid fire by holding down the mouse button.

- Asteroids:

  The asteroids and asteroid system as a whole were implemented with the particle system from the COMP 371 assignment frameworks as a foundation.

  Asteroids were considered to be implemented as particle systems initially. However, it posed an issue for detecting collisions between Projectiles (a subclass of Model) with Asteroids (which would not be a subtype of Model if implemented as Particles within a ParticleSystem).

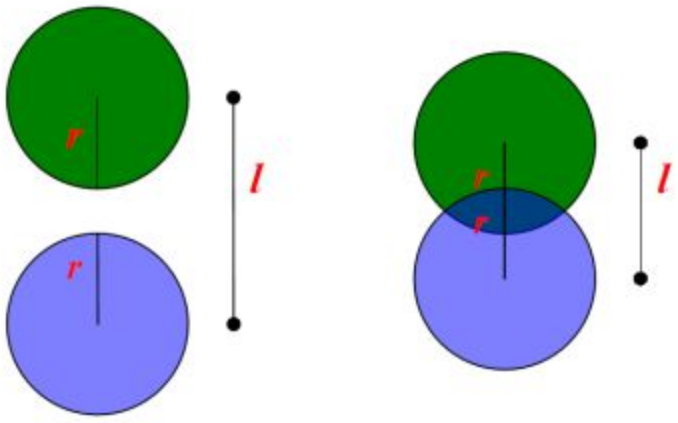
As a result, the AsteroidSystem uses similar logic as the ParticleSystem, but generates Asteroids, subtype of Model. It was then possible to attach a ParticleSystem to each Asteroid, giving it a more realistic effect. Since they are now a subtype of Model, it makes it possible to apply the Collision Detection to them.

- Collision Detection:

  The user's spaceship shoots projectiles at oncoming asteroids and other satellites within the spatial environment. If contact is made between the objects, certain actions are taken.

  To determine the collisions, calculations are made using bounding sphere calculations. The bounding spheres encapsulate the objects in the smallest space possible to contain the object in its entirety. More specifically, the sum radii of the objects are compared against the norm the vector between the model positions. In the case that the norm is smaller than or equal to the sum of radii, a collision has occurred, in which case certain action must be taken (including setting new velocities based on the principles of momentum conservation).

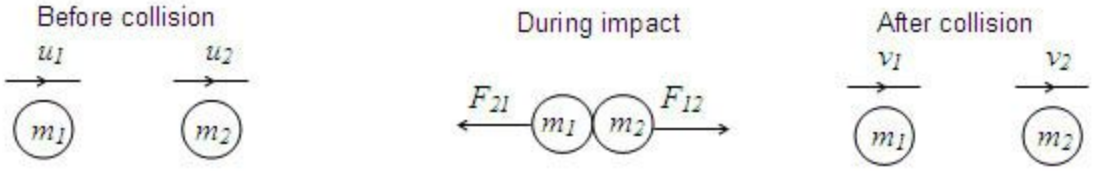
$$\|P_1 - P_2\| \leq r_1 + r_2$$



The following equations must be satisfied in order to abide by the principles of momentum and energy conservation for a comple.

$$M_1V_{1i} + M_2V_{2i} = M_1V_{1f} + M_2V_{2f}$$

$$\frac{1}{2}M_1\|V_{1i}\|^2 + \frac{1}{2}M_2\|V_{2i}\|^2 = \frac{1}{2}M_1\left\|V_{1f}\right\|^2 + \frac{1}{2}M_2\left\|V_{2f}\right\|^2$$



- Texturing/Skybox:

  The framework provided the appropriate vertex and fragment shaders required texture mapping with UV coordinates (2D coordinates). As such, the task was to simply modify the existing draw functions to take into account texture loading to the GPU for the models which did not currently make use of texture mapping.

  A new requirement was to map textures to a cube using 3D texture coordinates for a skybox in the world. This used a cubemap to incorporate a 2D texture to each face of the cube.

  The alternative to using a cubemap for the skybox was to individually texture each face instead of joining them into a single cubemap texture. While feasible, it resulted in the seams being visible at the edges of the skybox, thus negatively affecting the appearance and immersiveness of the skybox. The only advantage it could have offered would be not having to develop new shaders to take into account the 3D texture coordinates, however that task in itself was relatively simple.

  Ultimately, texturing a cubemap was logically the same idea, where the cubemap texture would be considered as a single texture (joining the six textures of each face). This provided a seamless look to the entire skybox and was preferred over the previous solution as it provided a more natural viewing experience for the user.

- Planets:

The whole SolarSystem is composed of instances of nine Planets, a subtype of SphereModel, and the Sun, a SphereModel itself. The Sun is positioned in the origin of the world, and planets are all placed in their real life order. They also have a mass relatively correct, meaning that if in real life, the Earth is lighter than Jupiter, it also is in the SolarSystem. Those masses are used during the rotate process, making them rotate faster or slower.

Planets also have a size (scaling) relatively correct, as per NASA's research.

A few attempts were taken before correctly implementing a perfect rotation. Each alternative presented an issue of their own until a final solution was found providing the appropriate animations. At first, the animation keys were used, but the rotation was not smooth like a perfect circle unless a large number of points were expressed to change the trajectory of the path. Then, the glm::rotate(...) function was used, using the Y-axis as the rotation axis, however this also gave unexpected results. Another try was to use the equation of a circle (X^2 + Y^2 = RADIUS^2). At each update cycle, the X value would be incremented by 1, and Y would get calculated based on its new value. It was also giving unexpected results. Finally, using the cos() and sin() for determining new positions worked and planets are rotating smoothly on a perfect circle.

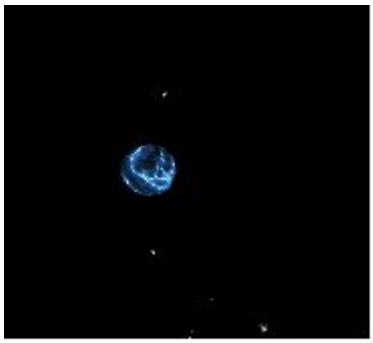
Results

Major Features

- Camera:

This game has several different cameras which can be switched with certain keys. There is a first person camera, a third person camera, a scenic third person camera, and a static camera. All of these cameras are of good quality and are set at a good angle. The user is able to switch to any camera at any time without any performance issues. There is also the ability for the user to vary the field of view by scrolling the mouse, which is compatible with all cameras. This does not degrade the performance in any way.

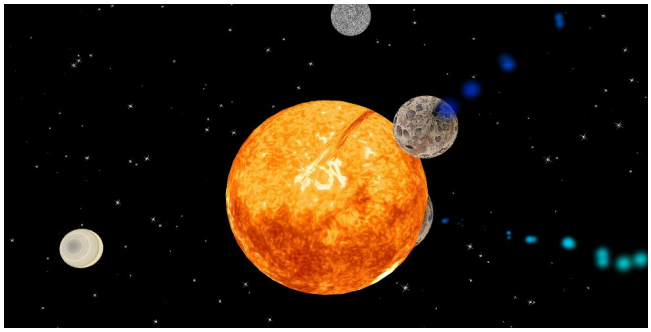
- Projectiles:



The shooting aspect of this game is that the spaceship shoots projectiles at incoming asteroids. The player must hit the asteroid before the asteroid hits the player. The shooting aspect performs well, the player is able to shoot many projectiles without it negatively affecting performance.

- Collision between projectile and asteroid:

If a projectile makes contact with an asteroid, the collision detection will trigger a function which will result in the asteroid to go flying off into space. This collision detection is done by comparing the mass and radiuses of the two colliding spheres. As soon as the spheres touch, the asteroid will fly off into the opposite direction. This collision detection algorithm has good performance, it runs quickly and efficiently without affecting program speed.



- Textures/Skybox:

The final result of the game includes a few textures and a skybox that looks like outer space. The texture mapping to the models worked as planned as they all wrap around their objects as they are supposed to do. They look pleasing to the eye and do not cause any performance issues. Similarly for

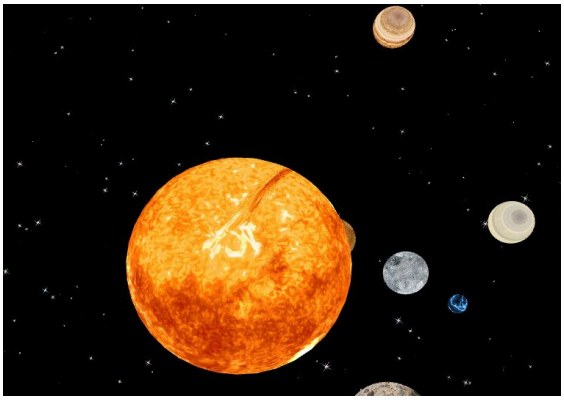
the skybox, it covers the whole world without causing any bugs or other issues.

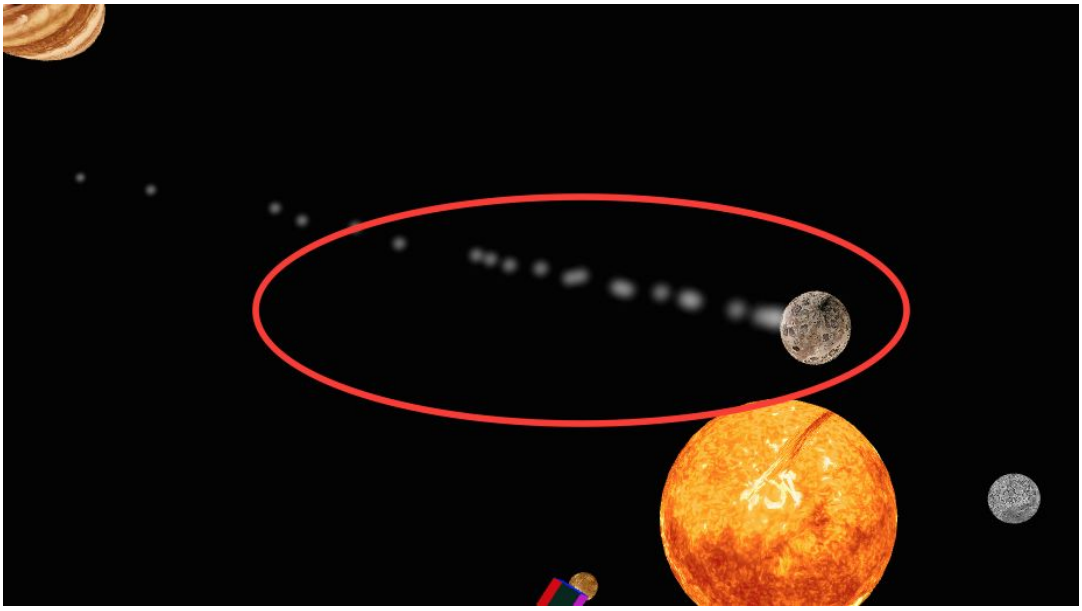
- Spaceship movement:

The spaceship is able to move around the world using the A and D keys for x-axis movement, CRTL and SPACE for y-axis movement, and W and S keys for z-axis movement. The spaceship is able to move fluidly without causing lag or any other performance issues.

- Asteroid movement:

The asteroids are implemented to move on their own towards the player. Their goal is to reach the a designated point without being shot by the player. There can be many asteroids in the world at once. The more there are, the more difficult it becomes for the player. The asteroids move towards the origin at a steady pace and do not cause any major performance issues. As more asteroids are initialized, the potential for negatively affecting the performance is possible, however has been accounted for and thus never affected it to the point of lagging or any other more serious problems (the maximum amount of asteroids present at a time has been limited to prevent these issues).





- Planet rotation around Sun

The Sun is a SphereModel in the origin of the world. Planets are rotating around it at different speeds, depending on their mass. The heavier they are, the slower they rotate. After attempting various implementations for determining the correct rotations and revolutions, no performance issues arose and a smooth movements were viewed across each cosmic element.

Discussion

First of all, if we had more time, we would have completed our initial game idea, which consisted of shooting asteroids before they hit our ship. This means also having an incrementing Score, a certain amount of lives that if all lost, the game is over, and even different levels.

However, since the scope of the project was to integrate graphic related stuff, rather than game logics, we decided to focus on integrating more features. However, there was still a couple of enhancements we would have liked to have. For example, having asteroids follow a B-Spline curve to give it a more realistic movement, rather than moving on linearly.

Also, planets are currently rotating around the sun, but are not rotating around themselves, which would have given them a more realistic effect. Another addition is having the moon rotate around planet Earth, while Earth rotates around the sun.

Various other miscellaneous features that would have been interesting to implement, including mirrors decals where objects collided would have been interesting to include.

Object loading was ultimately unimplemented due to several issues being faced. While the functionality of the project is there, it is not as aesthetically pleasing as we would have liked it to be.

We had absolutely no knowledge of OpenGL prior to this course, nor any basis about graphic design theory. We learned all the basics from the assignments, from setting up the camera properly, converting from Model View up until the Projection View. We learned how to scale, translate and rotate objects in the world, the Billboard design pattern, and how to play with lighting/shading. We successfully applied all of it in our project to give a final result which is very realistic of our Solar System.

User Manual

This project used the COMP 371 assignment framework as its foundation. As such, all the existing libraries remained and required no additional work to utilize.

No other instructions are required to compile or run the project.

Library	Version	Description
GLM	0.9.5.4	OpenGL Mathematics. Library providing mathematics functionality.
GLFW	3.0.4	Open source library for creating windows with OpenGL contexts. Additionally, it supports inputs and events.
GLEW	1.10.0	Extension Wrangler Library. Open source library for determining which OpenGL extensions are supported on the target platform.
FreeImage	3.17.0	Open source library supporting various image formats. Used for importing images to be used as textures for objects in the scene.

The user controls the spaceship and / or the camera depending on the selected camera (for example, the user cannot control the camera or spaceship when viewing from the static camera or the scenic third person camera, whereas there are no restrictions when manipulating the spaceship through the first and regular third person cameras). Inputs are received from the keyboard and mouse.

Key Binding	Action	Mouse Binding	Action
W, A, S, D, CTRL, SPACE	Move camera along each axis for compatible camera (first person and third person cameras) ● X-axis: A, D ● Y-axis: CTRL, SPACE ● Z-axis: W, S	Mouse Movement	Change vertical and horizontal angle (on first person and third person cameras)
0, 1, 2, 3	Switch various cameras (first person, third person, scenic third person, static, etc.)	Mouse Scroll	Change field of view of perspective view frustum
R	Reset camera field of view for perspective view frustum	Mouse Left Click	Shoot projectiles
M	Toggle wireframe mesh on models		
ESC	Close program		