COMP 477 Project

Hide-and-Seek, Pathfinding, Collisions, and Particle Systems

Presented to Dr. Serguei Mokhov

Developed by Team 07
Jason Brennan
Charles-Antoine Guité
Maxim Sinelnikov
Omar Al-Farajat

Concordia University

Fall 2019 COMP 477

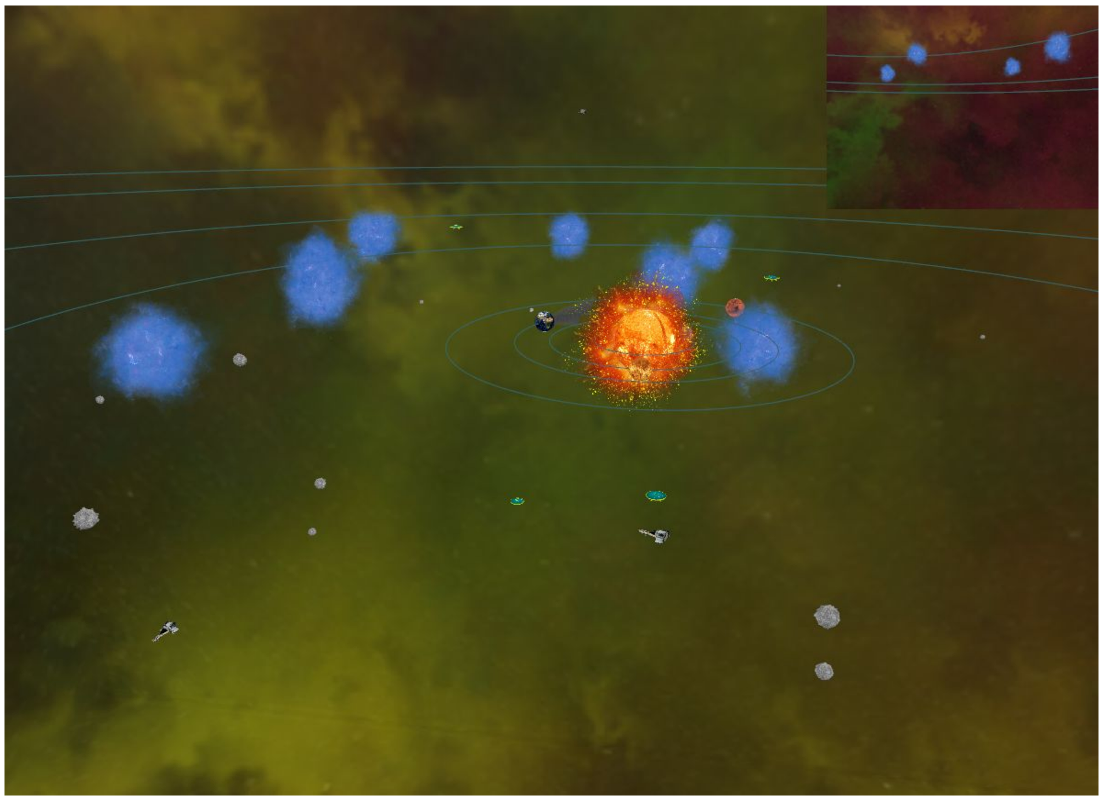
Outline

1. Background

The purpose of our project was to create a scene where hawks hunted smaller birds through obstacles like fire, clouds, and smoke. The user would be able to toggle the camera through world, hawk, and bird views.

Since making our proposal, the scope of the project has remained the same, however, we decided to have a space-themed aesthetic instead. Spaceships are the **hunters** that seek out alien UFOs as **prey**. In this new space theme, the obstacles are now **fire** (from the imploding sun in the center of the scene), celestial **storms** (the blue clouds), and **smoke** from an erupting mega-volcano on Earth, as well as planets and asteroids.

Spaceship "Hunter" (left) versus UFO "Prey" (right).


A solar system scene populated with hunters, prey, and obstacles.

2. Tools and Technologies

Blender
We used Blender to modify obj files for the hunter and prey models to correct their orientations and to have them be compatible with the OBJ loader used in the framework.

GIMP
Used to fixed alpha channels in certain textures.

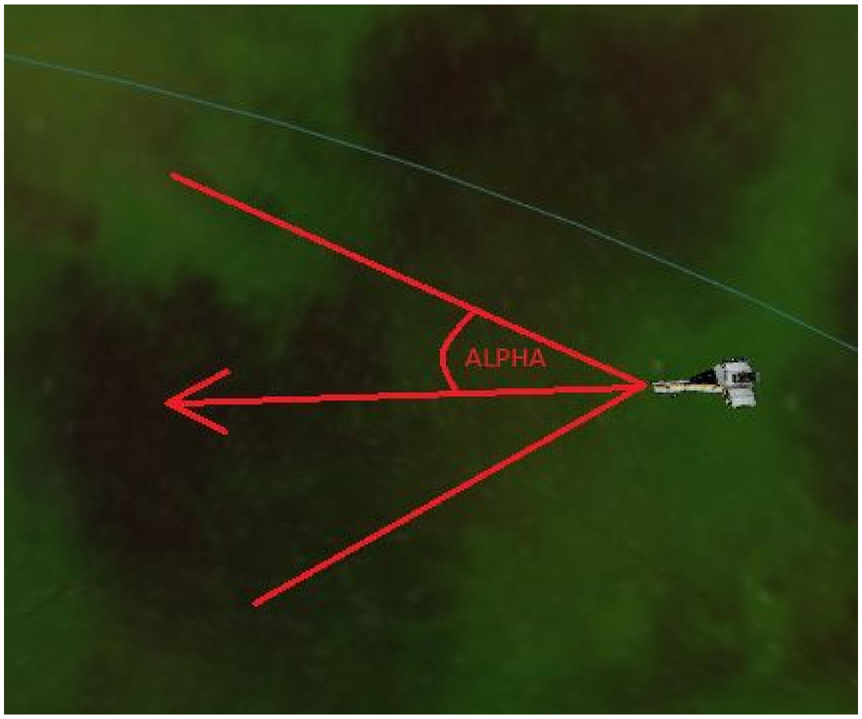
Visual Studio, COMP 371 Framework, and OpenGL
Our source code is managed through a Visual Studio project which contains an OpenGL-based framework used in COMP 371. The framework was adapted to our needs. The project was built and tested for the Windows platform.

Bitbucket, Sourcetree, Trello Boards, and Discord Webhook
Bitbucket and Sourcetree were used for version control with Trello Board integration for issue tracking. A webhook was used to send repository updates to a channel on our Discord server, which we also used for asynchronous communication.

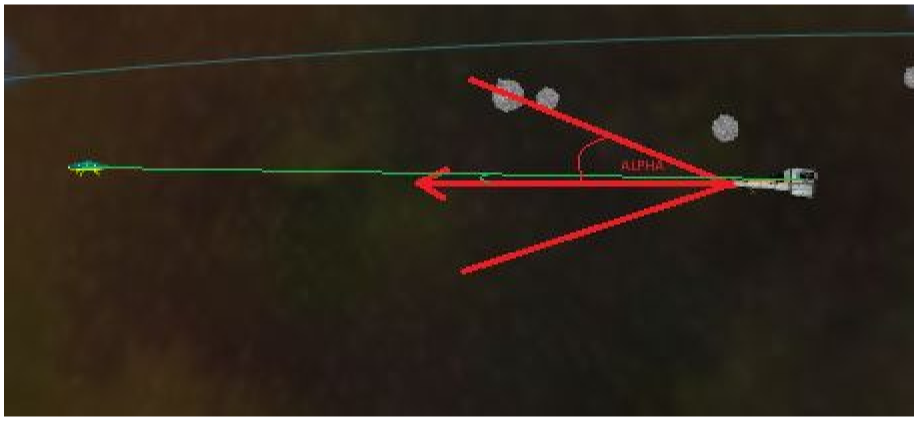
3. Features

3.1. Pathfinding and Raycasting

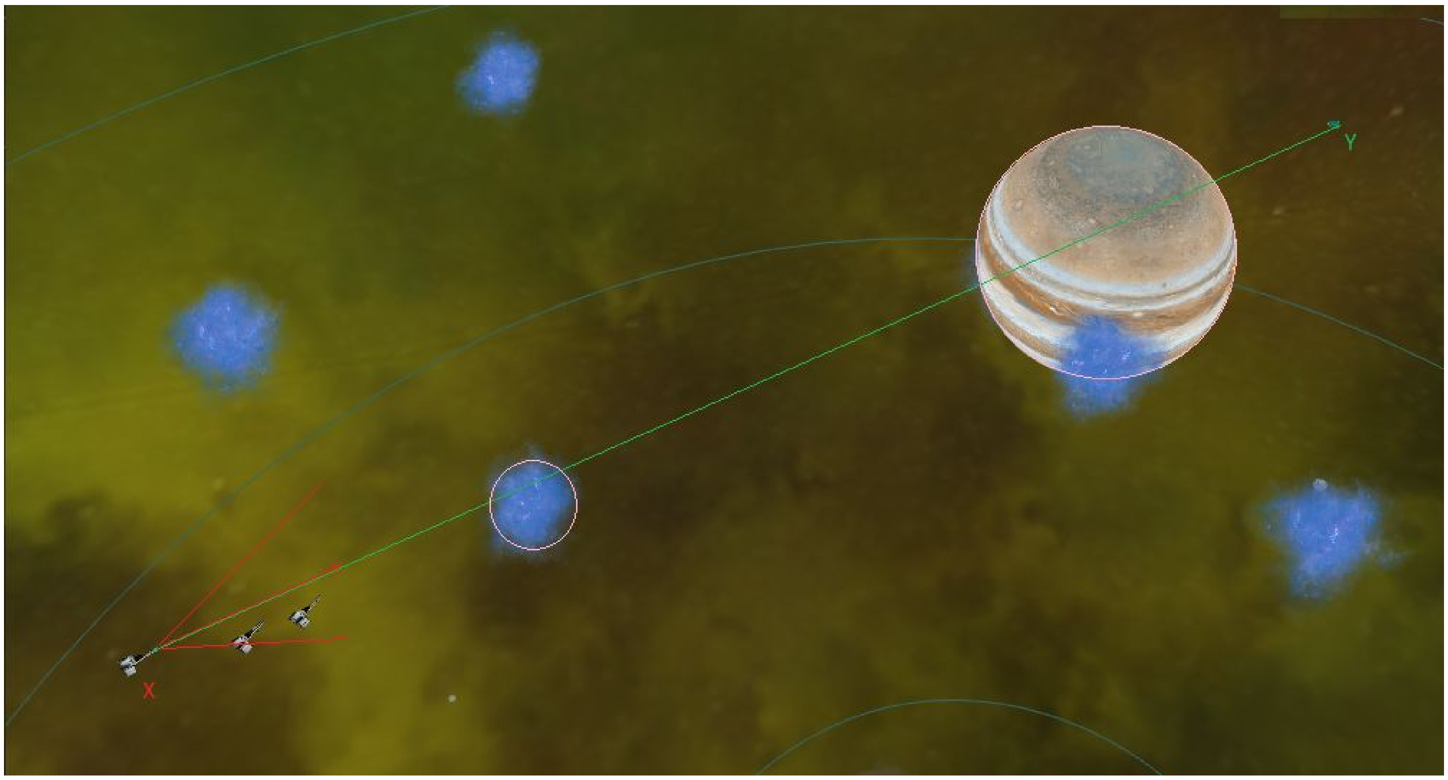
Our initial plans for the hunter's target acquisition included per-pixel raycasting in the entire viewport. Preliminary performance tests were discouraging. Performance tests were run on an i7-8700k processor with 32GB of RAM and a GTX 1070Ti graphics card. Running at a viewport resolution of 1200x900 and around 20 objects to query meant 21.6M intersection tests per frame! Average frame rate was around 5 frames per second when running in release mode. Further trials were attempted with lower viewport resolutions but we saw little in terms of frame rate increases. To maintain a smooth frame rate, a decision was made to implement raycasting only in the direction of the velocity vector of the hunter after a prey was detected to be within a conical field of view which is defined by alpha where:



For a prey to be detected, the angle between (the hunter's position and the prey's position) and the hunter's directional vector has to be smaller than alpha:

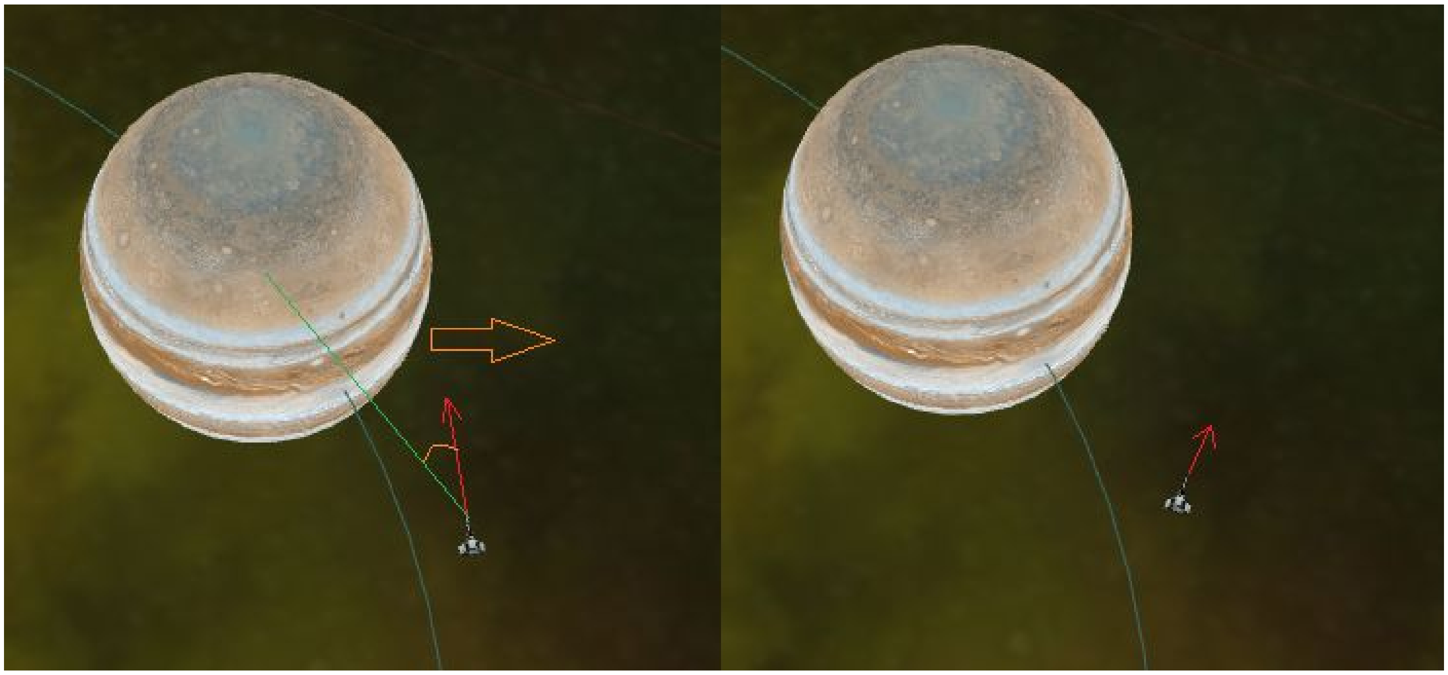


After the prey are detected the hunters will then verify that there are no planets, asteroids, or particle systems that are obstructing the prey. The hunter verifies this by using the model's <u>bounding spheres</u>: (hunter X view of prey Y will be obstructed by the planet and particle storm)

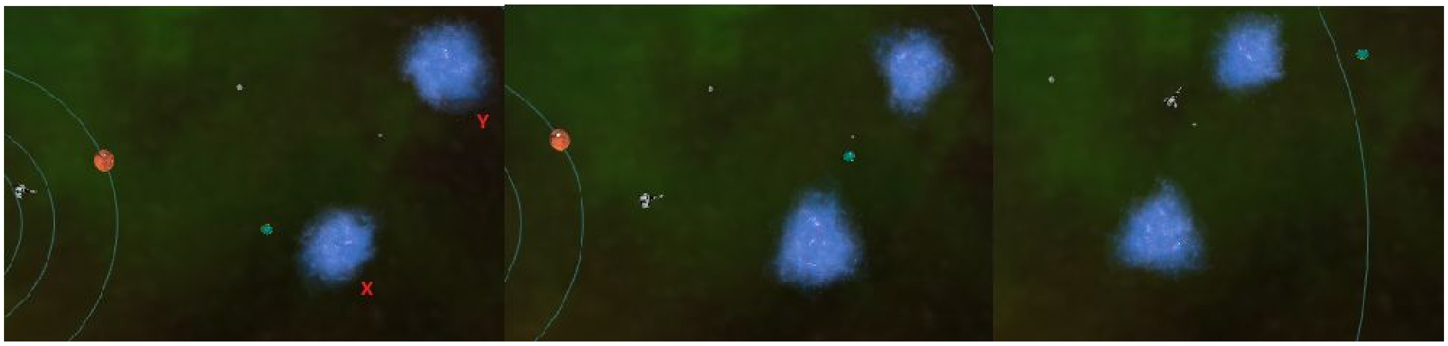


After a hunter has a list off all the prey in it's sight it will choose the closest one to it and change it's directional vector to point towards the prey.

When a hunter is in search mode it will spin in a circle and attempt to find a prey. To avoid flying into planets, it will send a ray in it's directional vector every update call to see if it is about to fly into a planet. If a hunter detects such a collision it will change it's directional vector depending on it's and the planet's position, the radius of the planet, and the angle between it and the center of the colliding planet:
(the following image shows 2 continues frames representing such an offset and change in directional vector)



When a hunter starts chasing a prey, our program will notify that prey that it is chased and the prey will try to escape by flying through all the storm clouds which are closest to it in the hopes of the hunter losing sight of the prey: the following 3 frames show how the prey will fly throw the next closest storm to escape (first storm X then storm Y) which will cause the hunter to lose sight of the prey in frame 3.



The hunter and prey models are oriented by default in the <0,0,1> direction. To draw them every frame we use the dot product between the spaceships normalized directional vector and <0,0,1> to find the angle in between and create a rotation matrix using that angle about the cross product vector between the spaceships directional vector and <0,0,1>.

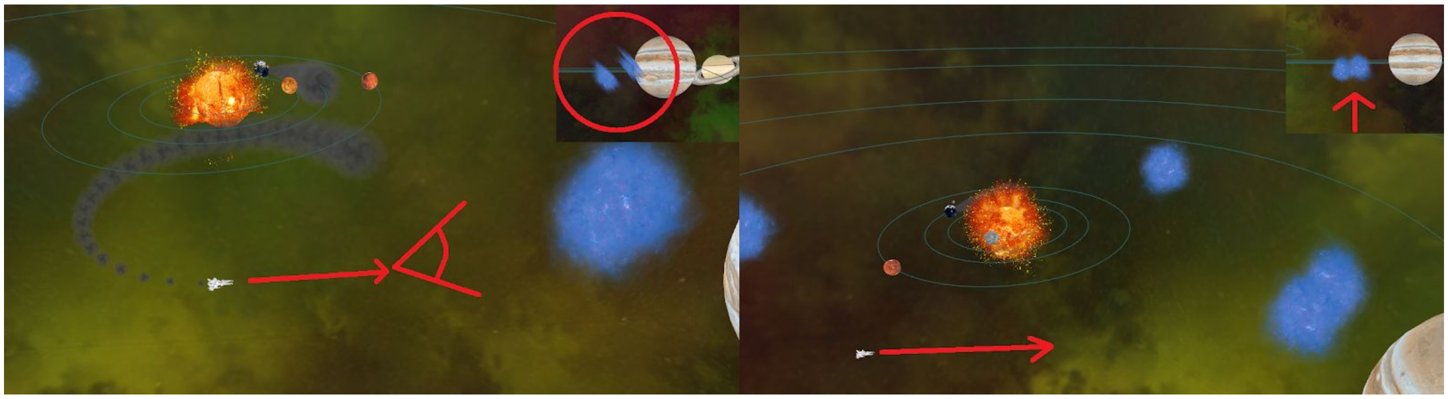
3.2. Collisions

Each model has its own bounding radius, which is determined primarily by its scaling. In the main world update loop, iterators go through the model list and constantly compare the distance between objects to the sum of both their radius. If the distance is smaller, a collision is detected. If both those objects are prey and hunter, the prey is taken out of the model list and destroyed. In other cases, models are displaced at impact.
This is done first by calculating a collision normal and a collision point from both models position. Then, a normal velocity and a tangent momentum can be calculated from both model using the previously found collision data and their current velocity. Finally, we take into account the mass of each model to influence the

final velocity. This is so that planets, for example, being such massive objects, are not displaced by an asteroid hitting it.

3.3. Particle Effects

The way the framework was designed, particle systems are attached to models defined in a scene file, as such we chose to use cubes that are scaled to 0 (so as not to be visible) and which are omitted from collision tests. Billboarding is the method by which particles are realized. They are populated into a BillboardList which is used to cycle through the particles as they are emitted from their source. The framework code was adapted to support more than one particle texture. Careful consideration was also made so that the billboards were oriented correctly to the line-of-sight depending on the viewport. The above mentioned raycasting feature also checks for the particles so as to obstruct a hunter's view from the prey.



Incorrect billboard orientation (left) vs. Correct billboard orientation (right).

3.4. Camera

There are three cameras in the project:

1. The main camera is a free-flying user controlled camera that can move and look anywhere in the scene.
2. The second camera is a third person camera that follows any prey or hunter model as it moves through the scene. This camera's lookAt vector is locked on the model however the user can use the mouse to orbit around.
3. The last camera is a first person camera aligned with any of the hunter's points of view.

Camera switching is accomplished through keyboard input:

1. Pressing 1 will activate the main camera.
2. Pressing 2 will activate the third person camera on the first model in the model list (hunters and prey). Pressing 2 again will cycle to the next model.
3. Pressing 3 will activate the first person camera on the first hunter model. Pressing 3 again will cycle to the next hunter model.

3.5. Viewports and Rendering

In order to provide more visual information to the user, we opted to use a two-pass render loop with two viewports. The procedure for drawing the scene is as follows:

1. Create a reference to the currently active camera
2. Clear the color and depth buffers
3. Activate the current shader program
4. Construct a view-projection matrix from the currently active camera and send this to the active shader
5. Send lighting information to the shader
6. Begin first render pass
7. Disable depth test, draw the skybox and re enable depth test
8. Draw all models (hunters, prey, planets, asteroids, etc)
9. Change active shader to planet path lines shader, send the camera view-projection
10. Draw key-frame animations
11. Change active shader to a textured shader, send the camera view-projection
12. Draw all billboards
13. Clear the depth buffer bit and enable the secondary viewport
14. Begin second render pass
15. Switch active camera to the first person camera, send view-projection to shader
16. Update all billboards so that they face the first person camera
17. Repeat steps 7-12
18. Enable the primary viewport
19. Re enable the camera from step 1

4. Code Structure

4.1. Framework

For this project, we used an adapted version of the COMP 371 Assignment Framework created by Dr. Nicolas Bergeron and updated by Gary Chang. It allows for the instantiation of custom scenes and objects such as models, animations, particle emitters, and so on. Objects can be readily inserted in a scene file which is parsed and processed at the beginning of the program.

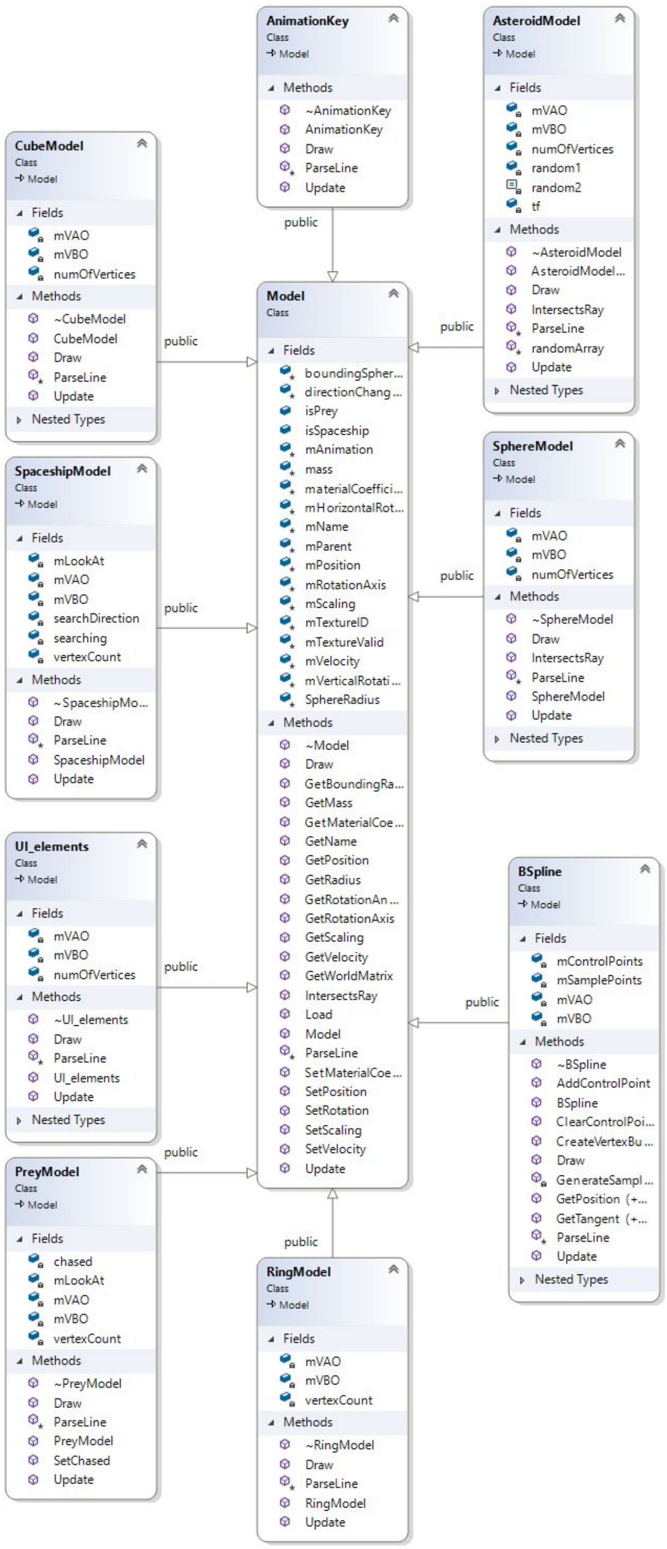
The framework already included some space-themed assets and a basic solar system scene, which were added as part of a COMP 371 Final Project. However, we added more assets such as models for the hunters and prey, as well as additional particle textures to distinguish different obstacles. It had to be modified to support the earlier mentioned features and other requirements for this project. Some examples of these adaptations are:

- The framework initially only supported one texture for particles which were implemented using billboarding, so changes were made to the World and ParticleSystem classes to support additional textures.

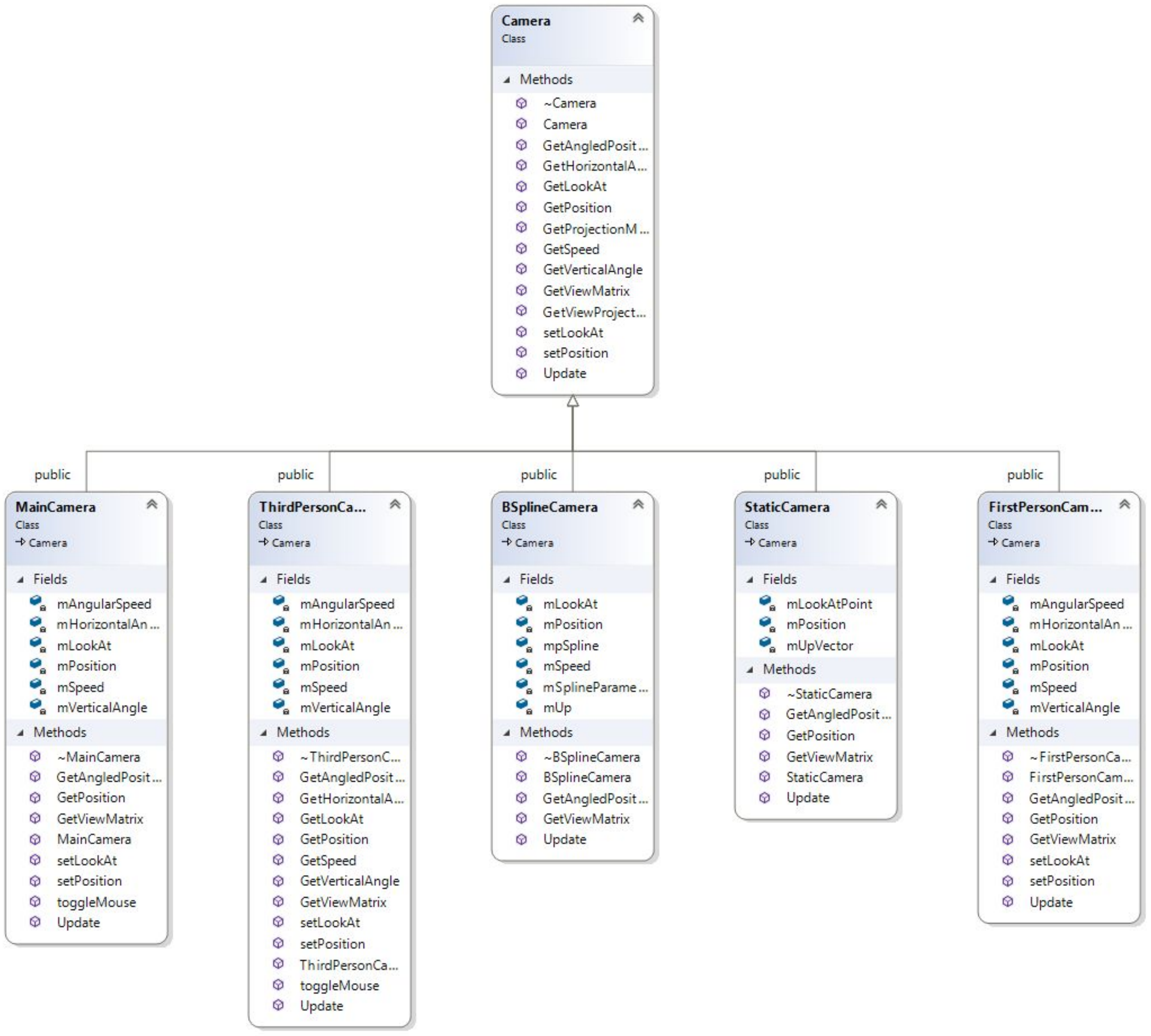
- The creation of the Prey and Spaceship class, which extends the Model class.

- Modification of various existing classes to change behaviour on instantiation or to add additional new utility functions, such as raycasting, retrieving lists of certain types of objects in the scene.

4.2. Model Class Diagram



4.3. Camera Class Diagram



5. Limitations and Future Work

One of the main limitations of our project is the amount of possible objects/models that can be in a certain scene at once. Having too many storm particles emitters (upwards of 40) or too many asteroids (upwards of 200) increases drastically the amount of calculations of particle position or collision detection necessary at each frame, resulting in a decreased frame rate which looks “laggy”.

As recommended by the professor during the project demonstration, a good element to add to this project would have been a decision logger, which observes each hunter and prey and outputs to an external file every decision that was taken by the agents, such as deciding to chase a specific prey, deciding to escape to a certain storm cloud, etc. With such a feature, it becomes a lot easier to evaluate the various algorithms used by the agents by having access to tangible data that reflects the accuracy and intended behaviour of said algorithms. Since the project demonstration, we have implemented the decision logger and now have better insight into how to tweak and improve the decision-making algorithms.

As mentioned earlier, the existing framework we used was managed as Visual Studio project and all of us worked on Windows-based machines, as such, the project was only tested and packaged for the Windows platform. In the future, once we’ve addressed some of the other limitations, we could look into making this project supported on other platforms

6. References and Credits

Smoke Texture: https://berserkon.com/transparent-particles.html

Storm Texture: https://cliparton.com/explore/nebula-png-orange.html

COMP 371 Assignment Framework: Created by Nicolas Bergeron on 8/7/14, updated by Gary Chang on 14/1/15. Copyright (c) 2014-2019 Concordia University. All rights reserved.

Hunter Model: https://www.turbosquid.com/FullPreview/Index.cfm/ID/711148
Royalty Free License - *All Extended Uses*

Prey Model: https://www.turbosquid.com/FullPreview/Index.cfm/ID/1333537
Royalty Free License - *All Extended Uses*