3D Space Combat Game

John Mitchell
University of California, Santa Cruz

Abstract
In this paper, I present a 3D space combat game implemented using XNA Framework 4.0. The game makes use of animation techniques including particle systems for effects and behaviors for controlling enemy ships.

1 Introduction

While particle systems and behaviors are two of the simpler techniques for controlling objects in games, they can work quite well if used correctly.

2 Related Works

Relatively few space games have been released in recent years, most notably *X³: Terran Conflict*, released in 2008. It makes use of similar techniques for effects. My aim was to create a game with a similar feel to *X³*. In order to promote this, as well as avoid spending large amounts of time on areas other than the main focus of this project, almost all of the models and textures I used were borrowed from *X³*.

3 Entity Representation

All in-game entities - "GameObjects" inherit from a single class, which stores basic information present in any entity, such as its 3D model, position, orientation, and speed. Position and speed are represented by a 3D vector using single precision floats. This could potentially cause problems at larger distances and higher speeds, although the values in this game are low enough that it makes little difference.

Orientation is represented by a Quaternion object, native to the XNA Framework. Using quaternions completely avoids the problem of gimbal lock – one rotation axis overlapping another due to subsequent local rotations.

The 3D model is loaded through the XNA content pipeline, which makes loading and displaying models quite easy. Most models borrowed from *X³* use a custom shader that enables specular and environment reflections, normal mapping, and self illumination.

3.1 Ships

In addition to all information present in GameObjects, ships contain information about their ship class, weapons, and additional information about their model, including engine positions and cockpit view. The ship class also implements an Update function for controlling its position, and two Draw functions, one for drawing the cockpit view and one for drawing an external view. For ships, movement is implemented in an arcade style, with the speed always parallel to the ship's forward vector. This gives a better feel, as well as makes the player's ship easier to control. Information about each ship type is loaded from a text file and stored in a dictionary, making it easy to add additional types.

3.2 Missiles

In addition to all information present in GameObjects, missiles contain information about how much damage they can deal and how long they can fly. The missile class also implements the same functions as the ship class, although the Draw from cockpit function does nothing because missiles have no cockpit. Unlike ships, missiles use a simplified Newtonian movement model. The speed is a

vector independent of the forward vector of the missile, and is changed by at most the acceleration of the missile to be as close to parallel to the forward vector as possible. Information about each missile type is also loaded from a text file and stored in a dictionary.

3.3 Projectiles

The projectile class is the most simple of the classes that inherit from GameObject. It only contains information about the projectile's range and damage. The Update function is also much simpler than the other classes, as projectiles only move in a straight line.

3.4 Weapons

Weapons are a special type of object, as they are only present attached to the hulls of ships. In addition to information about their model, orientation, and position on the ship's hull, they contain information about the projectiles they fire. Their Update function is also simple, as it only decrements the delay before it is ready to fire again. Information about each weapon type is also loaded from a text file and stored in a dictionary.

4 Shaders

Two custom shaders are used to display all models. The first, IgnoreLight, is a simple shader that outputs just the color read from a texture. It also contains the same input variables as the second shader in order to make the two swappable with a single change to the code.

The second, XModel, is responsible for all implemented texture effects present in X^3 models. This shader is built from the shader in the XNA normal mapping sample. This shader makes use of three additional texture maps – a normal map, a specular map, and a self illumination map – in order to compute specular and environment reflections, and illumination. First, the normal vector of the position of the screen pixel on the surface is computed. This direction is then used to compute the direction of specular reflections and the direction reflections will come from. The specular reflection is then calculated using a phong model, and the environment reflection is retrieved from a cube map that was created at runtime. The color from the self illumination map is then added and the final result is returned.

5 Particle System

Several particle systems are used to create effects including engine trails and explosions. My initial attempt at implementing a particle system ran everything on the CPU, which resulted in unacceptable performance with just several thousand particles. This was completely insufficient even for the trail of one missile. I then switched to using the XNA particles 3D sample, which runs as much as possible on the GPU. This particle system is capable of acceptable performance even when there are tens of thousands of particles at once. It contains a number of optimizations that allow it to run faster. Separate particle systems must be created for each different group of settings. Each system must have the same life. This is due to the fact that the list of particles is implemented as a circular queue. Active particles are always within the same range, so the it can easily find which particles are active and need to be drawn. This allows particle objects to always be kept in the same location, avoiding the performance overhead present when moving particles between lists of active and inactive particles. Each system must also have the same texture, range of sizes, and range of speeds, although this to make creating new particles faster rather than updating particles faster. To further optimize performance, the NoOverwrite flag is used when writing to the vertex buffer, which tells the GPU that any data it is currently using will not be changed in this write, so it can update the buffer without waiting for the current frame to finish drawing. Therefore, four

indices are required – firstActiveParticle for storing the index of the first particle that is still alive, firstNewParticle for storing the index of the first particle that has not yet been uploaded to the GPU, firstFreeParticle for storing the index of the first particle that can be reused, and finally firstRetiredParticle for storing the index of the first particle that has died but could still be actively drawn by the GPU, since it could potentially be up to two frames behind. By moving these indices around instead of the actual particle objects, performance is greatly increased.

The ParticleEffect shader is responsible for doing almost all of the computation related to drawing the particles. The custom vertex struct used does not contain the current position of the particle, only the initial position and velocity, so the shader must first compute the current position of the particle based on its age. It can then compute the position of the corners of the particle quad based on the size and rotation of the particle. Once this has been calculated, the shader can output the color of the pixel, which is read from the particle texture and multiplied by the color of the particle.

5.1 Particle Effects

The main use for particle effects in this game is drawing engine trails. These consist of stationary particles that are dropped at regular intervals along an object's path, approximately one per world unit. One side-effect of this approach is that when the player is moving backwards, the particle trail appears to flow over the front of the player's ship. Each particle fades and disappears over a span of one to two seconds, depending on the exact kind of trail it is. The texture is a noisy circle that fades out towards the edges, created by manipulating materials in 3DS Max.

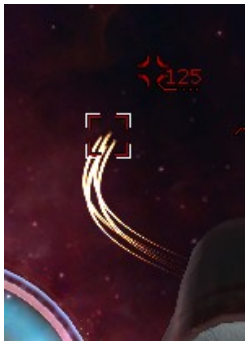
There are also three different particle systems for creating explosions of various sizes. These consist of fast-moving particles that fly away from the center point of the explosion. The texture is a small fireball that has been darkened to avoid oversaturating due to additive blending. The largest of the explosion particle systems is used for when a ship is killed. The smallest is used for drawing when a projectile impacts a ship's hull. The remaining explosion particle system is used whenever a missile explodes, whether it is from impacting a ship's hull, being shot down, or self destructing because it ran out of fuel.

The final particle system is used to draw dust from the nebula the player is flying through. It consists of a small number of short-lived large particles, randomly spawned around the player's current position. These particles are stationary in order to give the player a sense of how fast they are moving, although they rotate slowly in order to avoid repetitiveness.

6 Behaviors

All enemy ships are controlled using a set of behaviors based on the behaviors presented by Craig Reynolds. Enemy behaviors consist of two main modes, combat and patrol, plus obstacle avoidance that is always active. All behaviors are implemented using a “seekTo” operation, which attempts to turn the object to face the specified point.

The combat behavior is further divided into two phases. In the first phase – chasing the player – the enemies attempt to seek to the player and fire weapons when the player is in range. Once the enemy and player are within a certain distance of each other, the enemy ship will switch to the second combat phase. In the second phase – fleeing from the player for



another attack pass – the enemies attempt to seek to a point in the opposite direction of the player. In addition, if the player is directly facing an enemy ship it will instead seek to a random point, changing if the player manages to continue facing directly at that enemy ship. Finally, after either the enemy ship has reached its maximum weapon range or a certain amount of time has passed, it will switch back to the chase phase.

The obstacle avoidance behavior is relatively simple. If an enemy comes within a certain distance of another object, it will attempt to seek to a point in the opposite direction of the obstacle. While not perfect, this works quite well in most cases.

The patrol behavior is also relatively simple. When enemies are patrolling, they attempt to seek to waypoints on their patrol path. Once it is within a certain distance of the current waypoint, the enemy will switch to the next waypoint and continue on until it finds the player within its radar range, at which point it will switch to the combat behavior.

When combined, these behaviors result in somewhat complex movements. In particular, it is quite difficult to hit an enemy while it is in flee mode. Unfortunately, actually hitting the player with their weapons is quite difficult for the enemy ships due to the relatively slow speed of the projectiles – just five to eight times the maximum speed of the player's ship. By the time the projectile reaches the player, he will have likely moved from the point he was expected to be in. The same is applicable to the player as well, since aim assist is used due to the distances involved.

7 Collision Detection

Collision detection between projectiles or missiles and ships is done in two stages. In the first phase, the line between the projectile's current position and the position it will be in next frame is tested against the bounding sphere of each ship. If a possible collision is detected, the same line will be tested against each triangle of a simplified version of the ship's model. This model is loaded at the same time as the high quality model, and stored along with that model. If no collision model is present, the actual model is used instead.

8 Future Work

Currently, the game does not have any kind of menu system or collisions between ships. For future versions, these would be two of the main priorities. Further work would see all art borrowed from *X*^{3} replaced with unique art.

9 Conclusion

This paper has presented a 3D space combat game that makes use of behavioral algorithms for controlling enemy ships. Overall, the behaviors are quite successful, although they are too challenging for the player to put in a final game.

References

Egosoft. X^3: Terran Conflict – Info. http://www.egosoft.com/games/x3tc/info_en.php

Normal Mapping Sample. http://create.msdn.com/en-US/education/catalog/sample/normal_mapping

Particles 3D Sample. http://create.msdn.com/en-US/education/catalog/sample/particle_3d

Craig Reynolds. Steering Behaviors for Autonomous Characters. http://www.red3d.com/cwr/steer/