Astrophobia: A 3D Multiplayer Space Combat Game with Linear Entity Interpolation

A Senior Project

presented to

the Faculty of the Computer Science Department

California Polytechnic State University, San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Bachelor of Science in Computer Science

by

Luke Larson

December, 2013

Astrophobia: A 3D Multiplayer Space Combat Game with Linear Entity Interpolation

By Luke Larson

Abstract

Astrophobia is a *Descent*-like 3D networked multiplayer space combat game with linear entity interpolation (client-side animation between game-state packets to give the illusion of continuous game updates) similar to entity interpolation implemented in *Valve Software*'s *Source* engine. Additionally, *Astrophobia* procedurally generates unique levels, has zero-gravity physics, ship and projectile wall bouncing, hit detection, OpenGL 3D graphics, Phong lighting, ship model and texture, and a simple HUD that provides visualization of health points, aiming crosshair, and player scoreboard.

Problem Description and Motivation

The goal of this senior project, *Astrophobia*, was to create a *Descent*-like 3D networked multiplayer space combat game with smooth entity movement, procedurally generated maps, OpenGL 3D graphics, Phong lighting, physics, collision detection, entity modeling, entity texturing, a HUD, and fast-paced engaging gameplay. Though all features described are important, the primary focus of this senior project was to create a multiplayer game, which entails solving network related problems including client/server socket programming, transferring game state, and smoothing entity movement.

Astrophobia is set in the distant future. The solar system has been fully explored for some time. Planets that are habitable, like Mars, have been colonized. To keep up with human-kind's need for energy, companies have ventured into the outer limits of the

solar system in order to extract water contained in the huge asteroids that drift in that region. Using an efficient method, water is broken down into hydrogen gas, the primary fuel used in all space vehicles. Almost as soon as this process was discovered, a war for control of these distant regions began. Combat commonly occurred in the space-constrained caverns carved in the asteroids from the water mining process, necessitating the use of a nimble fighter. For this purpose alone, a fighter was invented by the largest arms dealer in the solar system. They called it the Manticore, and, because of its low cost and reliability, it became the most common weapon in the solar system. Now, factions across the solar system are engaged in an endless energy war, sending their best pilots to secure the huge water-rich asteroids that they seek to control. For these pilots, there is only one mission—destroy as many of the enemy's ships as possible.

The inspiration and motivation behind this project is a game developed by *Outrage Entertainment* called *Descent 3,* popular for its fast-paced multiplayer space combat (2). *Descent 3* screenshots are shown in Figure 1.

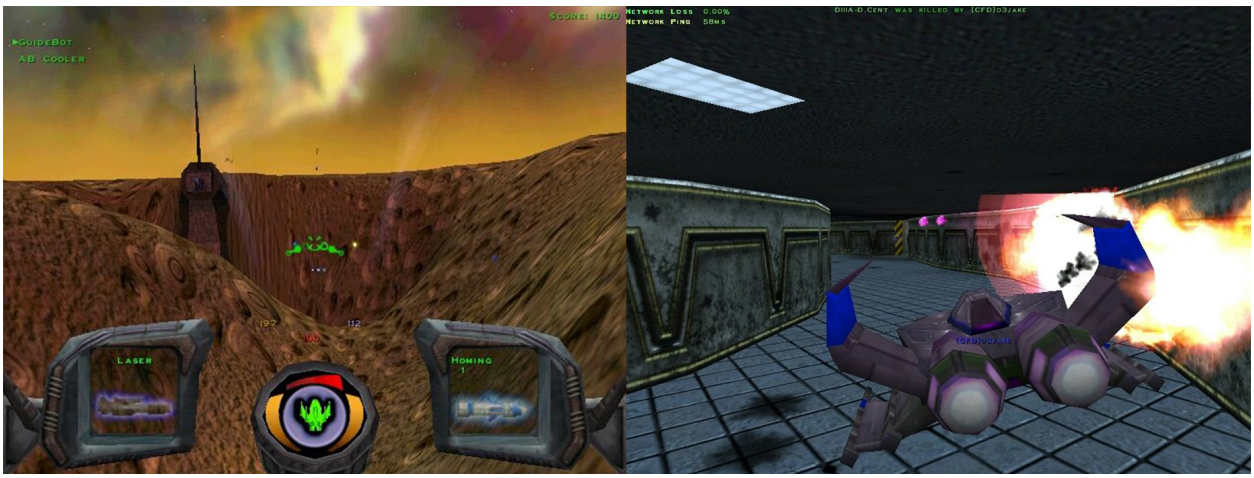

Figure 1: Screenshots of Outrage Entertainment's Descent 3.

The problem with *Descent 3* is that entity movement in multiplayer play appears jittery and choppy, causing aiming to be difficult and game immersion to be diminished. This choppiness is caused by a small delay between each game-state update sent to clients. Additionally, because servers update game state more frequently than they send game state to clients, *Descent 3* clients only see periodic snapshots of entity movement, putting players at a disadvantage. For instance, if players wish to hit moving targets, they must aim at the empty space in the future path of moving targets because on the server all moving objects have continued to move beyond their positions in the last snapshot sent to the client. This requirement creates awkward gameplay, requiring players to be aware of the networking architecture and manually compensate their aiming.

The primary challenge in developing *Astrophobia* was to address the networking issue encountered in *Descent 3*'s multiplayer play. The issue is caused by the availability of only a finite amount of server bandwidth. Game servers must send game-state updates to many clients, requiring significant bandwidth. A study performed on *Quake,* a game with comparable networking features to *Astrophobia,* found that to support 32 players in one game a server requires approximately 5.7 Mb/s upload bandwidth, increasing polynomially with the number of players (5). Because of this significant bandwidth usage, game servers are unable to send enough game states to keep up with the screen refresh rate on client computers. To reduce bandwidth usage, most game servers wait a constant amount of time (approximately 50 milliseconds) between server-to-client game-state updates (6). Consequently, if clients simply draw

each state as it is received, the delay between state updates will cause the objects drawn to the screen to appear noticeably choppy during movement, making small teleportations rather than moving smoothly.

Related Work

Developers at *Valve Software* implemented client-side entity interpolation to achieve smooth entity movement in their *Source* game engine. *Valve Software* described the idea in its paper entitled *Source Multiplayer Networking* (1). The paper describes a method whereby clients buffer several game states and interpolate between them over a time value. This allows clients to always have a local approximation of entity positions and render smooth entity movement. The negative consequence of using this method is that network latency is increased because of the requirement for at least two states to be buffered before drawing can occur. However, the method is still used because the positives gained from entity interpolation outweigh the negatives associated with a slight increase in latency.

Phong illumination is a shading technique used to generate specular reflections in 3D graphics. The technique was invented by Bui Tuong Phong at the University of Utah and published in his 1973 Ph.D dissertation (7). Phong illumination is used in game development to give surfaces smooth specular highlights resulting in a glossy or shiny look (8).

Collision detection is a technique used to detect when 3D game objects are intersecting or touching. Sphere bounding is a particular collision detection method that entails bounding objects in spheres and calculating distances between sphere centers.

When the distance between two spheres is less than or equal to the sum of both sphere radii, the objects are considered to be touching (9).

Blizzard Entertainment implemented procedural map generation in its popular *Diablo II* RPG (10). Procedural map generation is any technique used to generate maps according to some algorithm, opposed to loading static maps from a storage medium. In *Diablo II*, procedurally generated maps provide a replayable gaming experience, giving players a new dungeon to explore every game.

Implementation Details and Algorithm

Astrophobia implements Phong illumination as a fragment shader written in GLSL. Using GLSL takes full advantage of modern multicore GPUs in order to increase performance. Ship and projectile collision detection is implemented using the bounding sphere technique, meaning each ship and projectile is enclosed in a sphere and simple distance calculations are used to determine whether a collision has occurred. Maps are procedurally generated by randomly traversing a grid using a depth-first traversal method. Walls are created by tracing the randomly generated path. Once a map has been traced, the grid is used to create square wall objects. Collision detection is achieved by constraining the y-dimension between the ceiling and floor and, while iterating through each wall object, causing a collision if an entity attempts to move into a wall.

To address the networking issue encountered in *Descent 3*'s multiplayer play, *Astrophobia* implements entity interpolation based on the buffering technique described in *Valve*'s *Source Multiplayer Networking* paper. However, the buffering technique alone

is not enough to achieve entity interpolation. *Astrophobia* implements linear interpolation to interpolate entity positions between game-state updates. In order to explain the algorithm *Astrophobia* uses to achieve linear entity interpolation, it first helps to understand the data structures, system architecture, and network protocols used to implement multiplayer play.

Networking

To provide multiplayer capabilities in *Astrophobia*, a client/server game networking architecture was chosen. The client/server game architecture calls for having a central server maintaining total authority on game state and a number of clients sending input commands and receiving game-state data from the server. *Astrophobia* is composed of both client and server applications. The client is responsible for connecting to a server, capturing and sending player input to it, and then drawing the game state to the screen in 3D as it is received from the server. The server is responsible for receiving connections from clients, spawning player threads, receiving player input, processing game-state changes, and sending game-state information to clients.

Astrophobia uses several data structures to achieve its multiplayer capability. The two primary data structures allow for the transmission of data to and from the server. The client sends “command” packets to the server, which contain information about the state of the player’s keyboard and ship rotation, and the server sends “game state” packets to each client containing a list of entities that make up the game world. Client “command” packets are made up of an array of Boolean values for each

supported keyboard key, indicating whether each key is being pressed or not, and alpha and beta values containing angle values for the ship's pitch and yaw state respectively. Server "game state" packets are composed of a list of "entity" objects which contain entity state information including: unique identifier, position vector, velocity vector, acceleration vector, pitch and yaw, score, and health points. Clients send "command" packets approximately every 30 milliseconds and receive "game state" packets from the server every 50 milliseconds.

Astrophobia was implemented utilizing two common networking protocols, the reliable TCP protocol and the unreliable UDP protocol. TCP is used to handle initial connections and transfer information to establish UDP connections. The TCP protocol is inappropriate for sending most game data because of the large amount of data and processing required to ensure data transfer is reliable. The TCP protocol buffers sent data into chunks before sending, then requires receivers to reply with acknowledgment packets to ensure that data was actually received. This means that senders must wait for receipt of a packet before they can send another, and if an acknowledgment packet is not received after a predetermined amount of time, the same packet must be sent again, repeating the process until everything is reliably sent. Furthermore, the client must rearrange incoming data chunks into the correct order in the event that they are received out of order. These procedures all add latency to the game, meaning, the time between when the players take action and see a response on screen increases significantly, causing aiming to be frustratingly difficult. Instead, the unreliable but fast UDP protocol is used. The speed gained from UDP far outweighs network reliability

because, in fast-paced shooting games, latency often has a far greater negative effect on gameplay than the occasional packet loss. The server is sending enough packets per second, combined with client-side entity interpolation, so that losing a few packets is often unnoticeable to players.

Linear Interpolation

To achieve linear entity interpolation, *Astrophobia* buffers "game state" packets and then goes back in time to interpolate between them. A diagram of this process is shown in Figure 2.

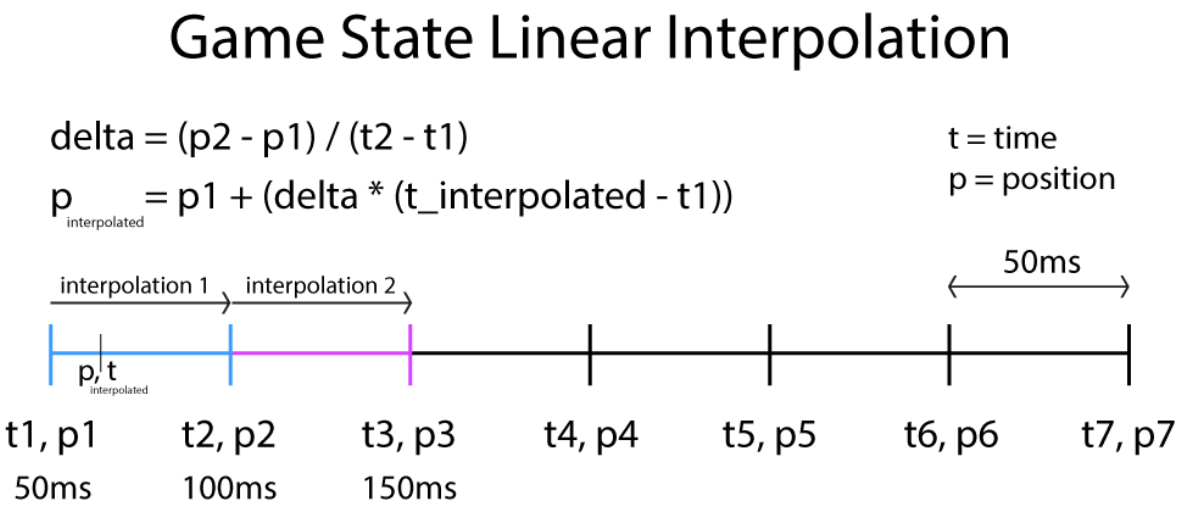

Figure 2: A diagram of client-side linear entity interpolation.

Specifically, the client buffers the latest two "game state" packets along with the time each was received from the server in a queue data structure. When two "game states" have been buffered, the client begins the interpolation process starting by calculating the elapsed time between receiving both game states. Using this elapsed

time value, the client calculates a linear equation over a time value that will smoothly interpolate an entity's position vector between the two buffered states over a span of time equal to the duration between each "game state" packet was received. As soon as another "game state" packet is received, the client pushes it into its "game state" buffer causing it to become the new end state and the previous end state to become the current start state therefore allowing the process to repeat.

Results

Utilizing linear entity interpolation in *Astrophobia* eliminated the type of entity jittering that *Descent 3* multiplayer games featured. Before linear entity interpolation, fast ship or projectile movement made aiming and dodging extremely difficult. With it, entity movement is smooth, even during quick changes in movement. Adding this feature to *Astrophobia* greatly increased its gameplay and entertainment value and together with physics, small levels, and projectile weapons makes for the type of fast-paced space combat game that this senior project originally sought to achieve.

When an *Astrophobia* server is started, a level is randomly generated and displayed in console output as shown in Figure 3.

Figure 3: A screenshot of the server console output on started.

The characters shown compose a top-down view of the generated map. A breakdown of what the symbols stand for is as follows. The ‘@’ symbols define the map boundaries, the ‘%’ symbol defines the generated map walls, the ‘#’ symbol is unused space inside the boundaries, and the ‘.’ symbol is explorable space inside the generated level.

After joining a game, a player is placed in a random location in the randomly generated map. The player HUD is shown in Figure 4.

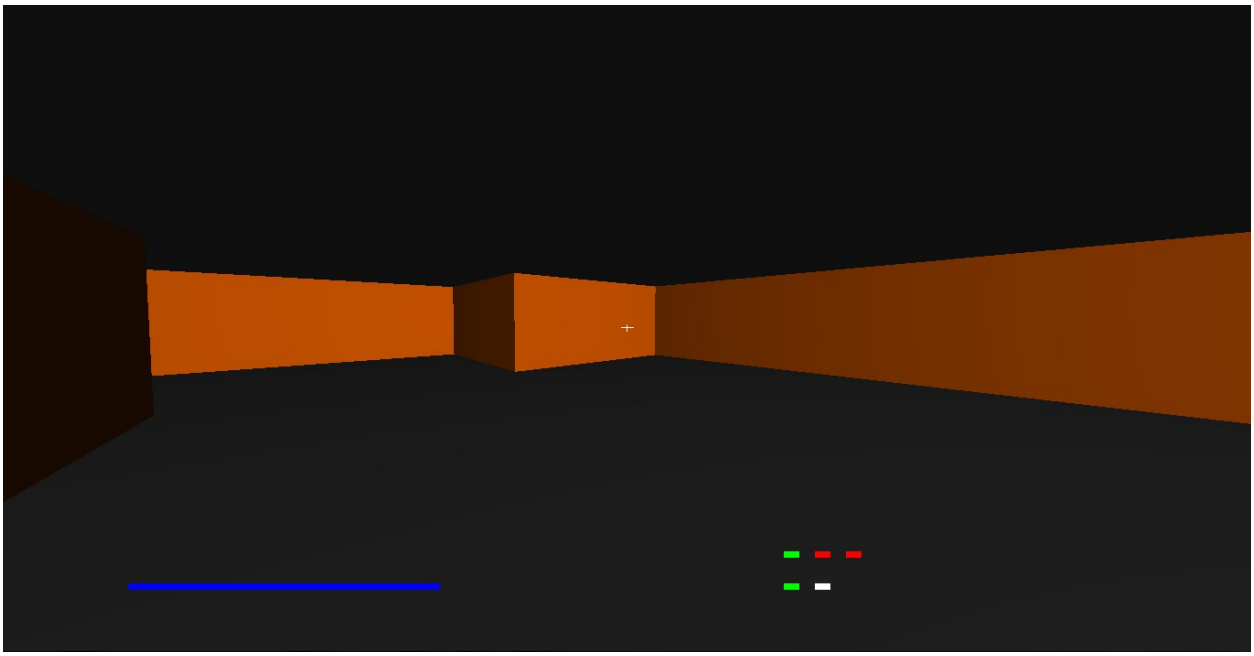

Figure 4: A screenshot of the player's HUD.

A breakdown of what each HUD element visualizes is shown in Figure 5.

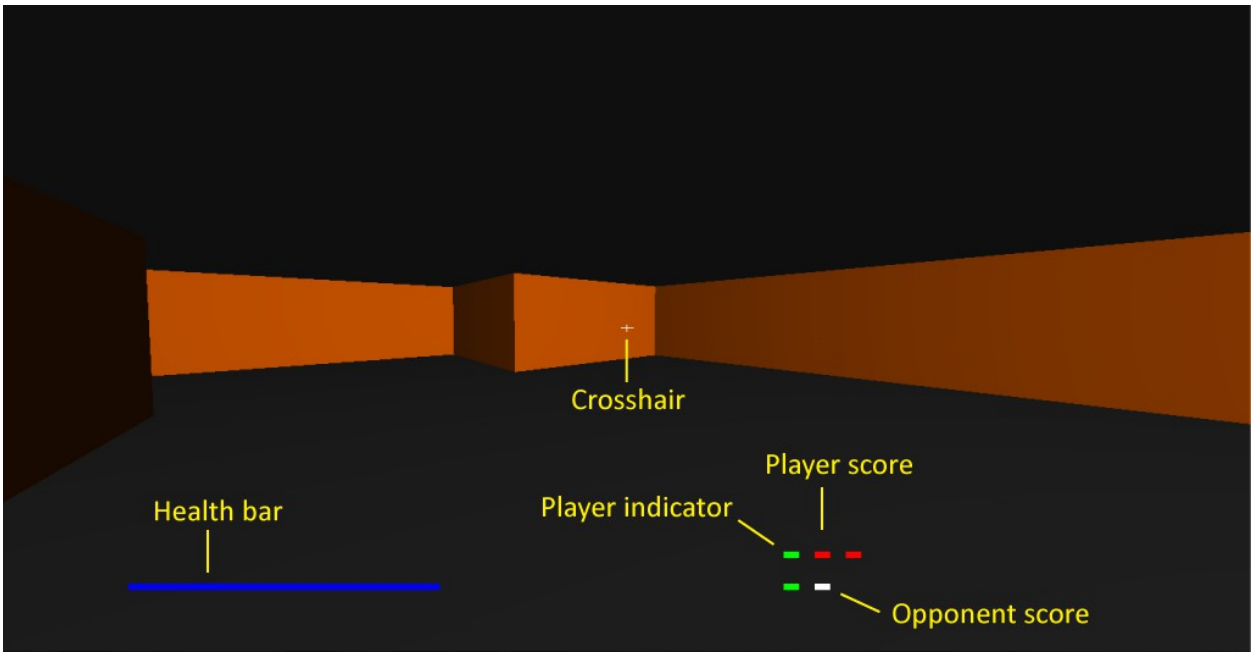

Figure 5: A screenshot of the player's HUD with annotations.

The health bar displays the player's current health points, the crosshair in the middle of the screen is intended to aid players in aiming, the green rectangles indicate players' existence and mark the location of their score, and the number of red or white rectangles add up to the number of kills either the player or the opponent has respectively.

The players' ship, named Manticore, is shown in Figure 6.

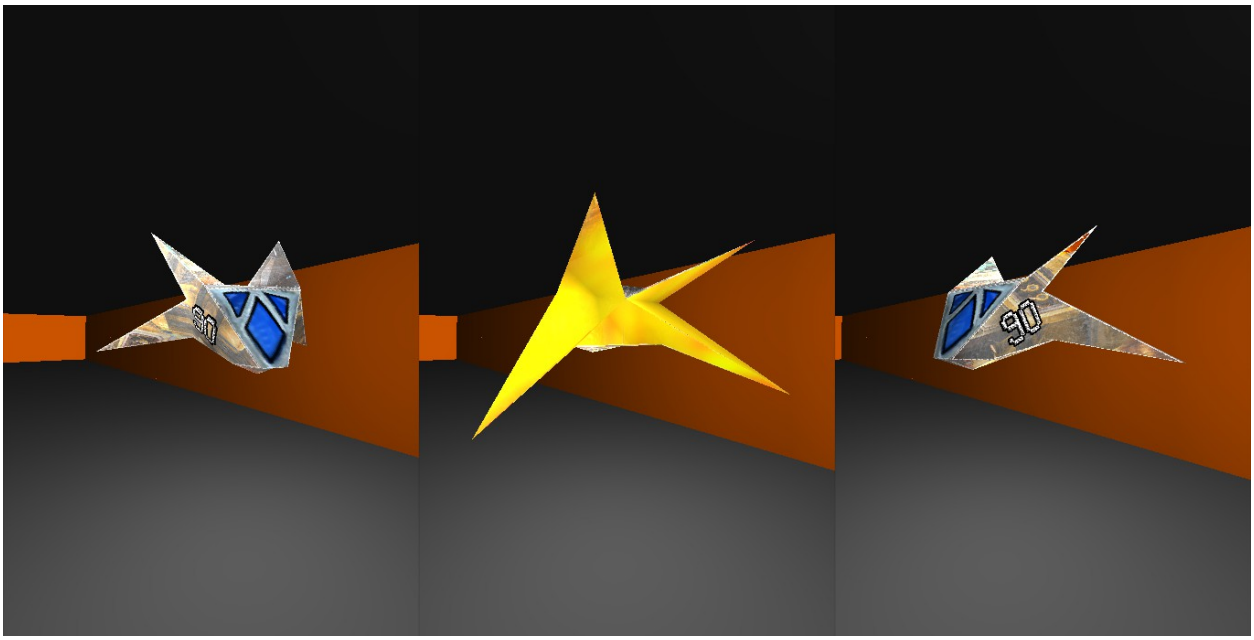

Figure 6: A screenshot of the player's ship, named Manticore.

The Manticore ship model was created using the open-source *Blender* 3D computer graphics software (4). Texturing was achieved by creating a UV map with *Blender* and painting the texture in an image editor. The UV map and model data were exported and loaded into OpenGL with *The Open Asset Import Library* along with the texture file (3).

There are two types of projectiles in *Astrophobia*: primary fire and alternate fire. Players can shoot primary fire and alternate fire projectiles by pressing the left and right mouse buttons respectively.

The primary fire projectile is shown in Figure 7.

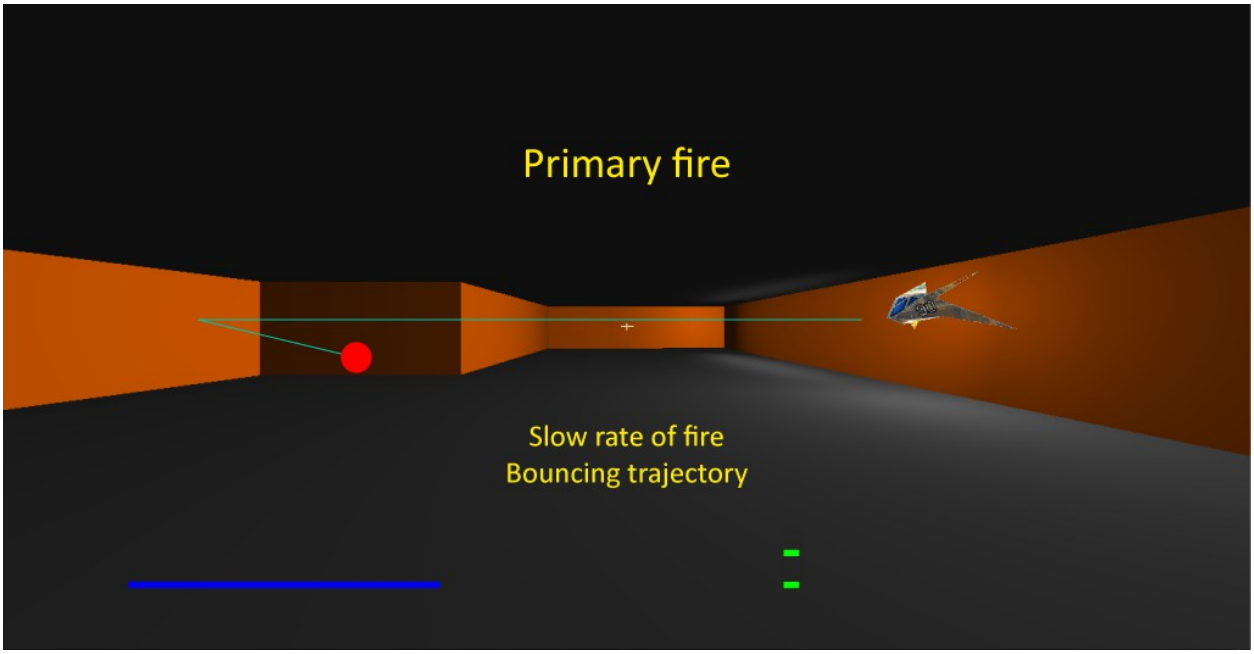

Figure 7: A screenshot of a Manticore firing primary fire projectiles with annotations.

Primary fire projectiles appear red, have a slow rate of fire, fast speed, and fly in a straight line. As the green line annotation indicates, primary fire projectiles bounce. In particular, primary fire projectiles bounce off of walls, ceilings, and floors allowing players to bounce projectiles around corners to hit enemies.

The alternate fire projectile is shown in Figure 8.

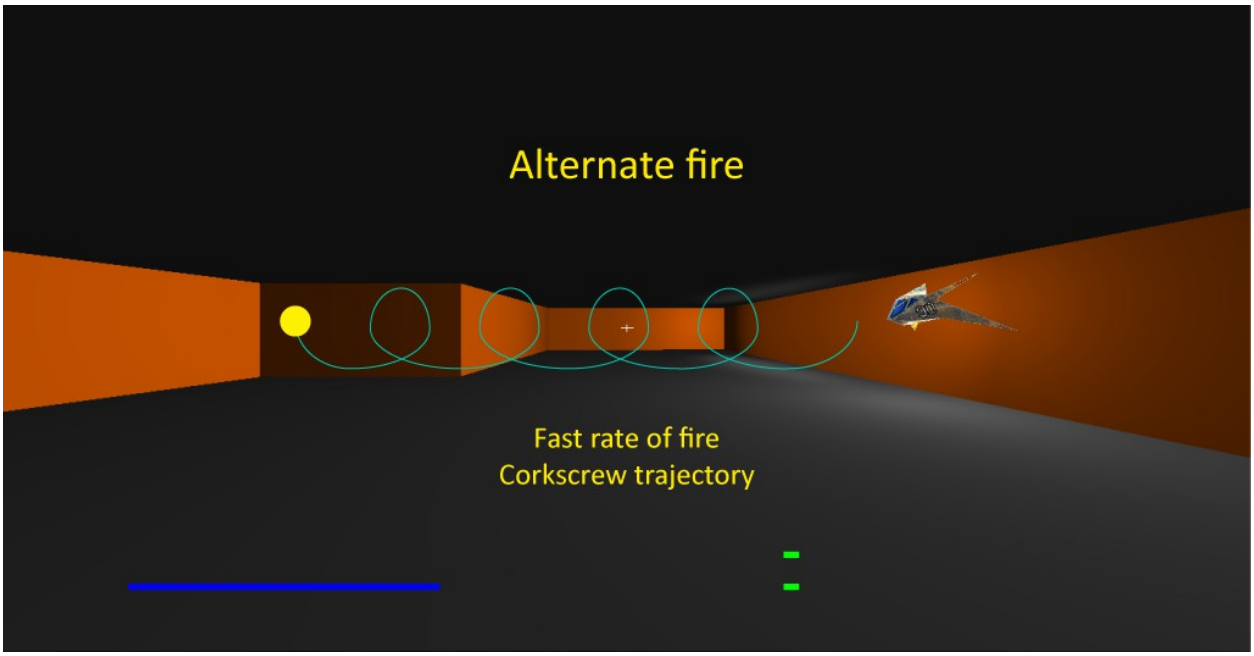

Figure 8: A screenshot of a Manticore firing alternate fire projectiles with annotations.

Alternate fire projectiles appear yellow, have a fast rate of fire, slow speed, and fly in a corkscrew pattern making them useful for rapidly firing at nearby opponents but relatively ineffective at accurately hitting opponents at longer distances.

Astrophobia implements basic ship physics, including acceleration, deceleration, and collision detection. When a movement command is initiated with one of the W, A, S, D keys, an acceleration vector is applied to a ship's velocity vector, providing smooth accelerations up to a maximum speed. When movement keys are released, a resistance vector is applied to a ship's velocity vector until the ship's movement stops, simulating thrusters automatically stopping the ship. If a ship hits an obstacle, the ship bounces off of it. *Astrophobia* applies multiple acceleration vectors if multiple movement keys are held at the same time, allowing for a high degree of freedom in movement.

The game play in *Astrophobia* is designed to be quick matches of one-versus-one play. Each player starts with 30 hitpoints, allowing ships to get hit a maximum of three times before death. Upon death, a player's ship explodes and is respawned at a random location in the map with full hitpoints. Each match goes until a player wins the match by reaching 10 kills.

Conclusion

Astrophobia was designed to be a fast-paced networked multiplayer space combat game. The techniques implemented and described in the previous sections—including linear entity interpolation, procedurally generated maps, OpenGL 3D graphics, Phong lighting, physics, collision detection, entity modeling, entity texturing, and a HUD—all work together to meet *Astrophobia*'s design goal. The most important technique implemented to achieve this goal was linear entity interpolation because it overcame the choppiness issue. Without it, the game is practically unplayable. For this reason, the feature that turned out the best was linear entity interpolation.

In my experience playing *Astrophobia*, the features that influence gameplay most are linear entity interpolation, collision detection, the input system, and physics. The combination of collision detection and physics provides ship and projectile collision detection and rebounding, the input system makes movement feel quick, responsive, and free, and, finally, linear entity interpolation makes player and projectile movement look and feel liquid smooth.

Future Work

With any game, improvements are often progressively made over long periods of time, and *Astrophobia* is no exception. *Astrophobia*'s graphics could be improved by adding full level texturing and more level details such as multiple and differing light sources. Networking could be optimized to support more players and player names. More information could be added to the HUD such as a minimap, weapons information, and a textual scoreboard. A variety of ships could be added with differing attributes and weapons. A menu could be added with a server browser for easy game finding. Chat support could be added in menu and in game to allow for player socialization. Bots could be added so players could play without an internet connection. Automatic matchmaking could be added, alongside a laddering system, to encourage players to play longer and get better to improve their ladder ranking. As the number of clients increases, lag hiding features could be added like client-side prediction. Finally, cheating is always a problem in gaming so anticheat features could be added to the game to catch and ban cheaters from ruining the experience for legitimate players.

References

1. "Source Multiplayer Networking." *Valve Developer Community*. N.p., n.d. Web. 1 Dec. 2013. <https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking>.

2. "Descent 3 - D3 Game Info." N.p., n.d. Web. 1 Dec. 2013. <http://www.descent3.com/d3_main.php>.

3. "Open Asset Import Library." *Open Asset Import Library*. N.p., n.d. Web. 1 Dec. 2013. <http://assimp.sourceforge.net/>.

4. "Blender." *blender.org*. N.p., n.d. Web. 28 Nov. 2013. <http://www.blender.org>.

5. Cronin, Eric, Burton Filstrup, and Anthony Kurc. "A Distributed Multiplayer Game Server System." *University of Michigan* (2001): 16. Web. 4 Dec. 2013. <http://warriors.eecs.umich.edu/games/papers/quakefinal.pdf>.

6. Ratti, Saurabh, Behnoosh Hariri, and Shervin Shirmohammadi. "A Survey Of First-Person Shooter Gaming Traffic On The Internet." *IEEE Internet Computing* 14.5 (2010): 60-69. Web. 4 Dec. 2013. <http://www.discover.uottawa.ca/publications/files/MMOG-Survey_IEEE-IC.pdf>/

7. Phong, Bui Tuong. "Illumination for Computer Generated Pictures." *Graphics and Image Processing* 1 (1973): n. pag. *Northwestern Engineering*. Web. 5 Dec. 2013. <http://www.cs.northwestern.edu/~ago820/cs395/Papers/Phong_1975.pdf>.

8. "GLSL Programming/Unity/Smooth Specular Highlights." - *Wikibooks, open books for an open world*. N.p., n.d. Web. 5 Dec. 2013. <http://en.wikibooks.org/wiki/GLSL_Programming/Unity/Smooth_Specular_Highlights>.

9. "Bounding Volumes and Collisions." *Bounding Volumes and Collisions*. N.p., n.d. Web. 5 Dec. 2013. <http://msdn.microsoft.com/en-us/library/bb313876.aspx>.

10. "Area Size (Diablo II)." *Diablo Wiki*. N.p., n.d. Web. 5 Dec. 2013. <http://diablo.gamepedia.com/Area_Size_(Diablo_II)>.