Hook – A First Person Shooter Game

CS 470 – Capstone Project

Wesley Rathburn

April 29, 2010

Table of Contents

[Abstract 3](#_Toc260354583)

[1 – Introduction 4](#_Toc260354584)

[2 – Project Overview 5](#_Toc260354585)

[2.1 The Basic FPS Game Movement 5](#_Toc260354586)

[2.2 “Hook” Gravity 6](#_Toc260354587)

[3 – Project Requirements 8](#_Toc260354588)

[3.1 Functional Specifications 8](#_Toc260354589)

[3.2 System Specifications 9](#_Toc260354590)

[4 – Game Design 10](#_Toc260354591)

[4.1 Game Sprites (2D) 10](#_Toc260354592)

[4.2 Game Models (3D) 10](#_Toc260354593)

[4.3 Class Objects and Data Structures. 13](#_Toc260354594)

[4.4 Game Architecture and the XNA Content Pipeline 14](#_Toc260354595)

[4.5 Physics and Algorithms 15](#_Toc260354596)

[5 – Software Development Process 17](#_Toc260354597)

[5.1 Testing and Debugging 17](#_Toc260354598)

[5.2 Benefits of Prototyping 17](#_Toc260354599)

[5.3 Work Breakdown 17](#_Toc260354600)

[6 – End Result 19](#_Toc260354601)

[6.1 Final Prototype 19](#_Toc260354602)

[6.2 Future Releases 21](#_Toc260354603)

[7 – Summary and Conclusion 22](#_Toc260354604)

[Appendix A – User Manual 23](#_Toc260354605)

# Abstract

The video gaming industry has undergone significant and very successful growth over the past few decades. More recently, the evolution of networked multiplayer games has seen tremendous growth, and to date, the online multiplayer First Person Shooter (FPS) game is one of the most popular formats. FPS games are available and played on both gaming consoles and personal computers. Games such as Call of Duty: Modern Warfare, Halo, and Counter-Strike Source are just a few of the more popular FPS games on the market. Each of these games has their own unique features. There exists one similarity to nearly all FPS games, which is a single vector orientation for gravity. Though some FPS games allow for a reduction in gravity acceleration, no games that I have found allow for the axis of gravity to be changed. This project, Hook, was developed to implement a networked FPS game with a new twist on traditional FPS game gravity to allow for each individual player to control their orientation for gravity independently of other players.

# 1 – Introduction

This game project, Hook, was developed to fulfill the requirements of the CS A470 Applied Software Development course at the University of Alaska Anchorage. My decision to design a First Person Shooter game was based on a few reasons. Firstly, I wanted to learn the basics of game development and all of the aspects involved in the creation of a game. I also wanted to improve my knowledge and skill base programming in an Integrated Development Environment (IDE) using a language that was still fairly new to me. Lastly, the FPS game format is a favorite of mine, and I wanted to introduce a new and unique feature to FPS game play that has the potential to increase the level of difficulty but provide a new level of enjoyment. I developed this project alone but with the help many online resources.

# 2 – Project Overview

First Person Shooter-(FPS) games make up a large portion of the computer and console games market. Most FPS games put the user in control of a person or entity with which the user can navigate through a three-dimensional map world in search of enemies or objects to destroy. Enemies can be either computer controlled bots or users connected to a single game through a network. Free-for-all play or more commonly Teams of players combat each other to reach a defined objective.

The desired goal for this project is to develop a FPS game and incorporate a unique game feature that redefines the traditional FPS game style. The added difference will be the ability for each player to independently control and at-will change the orientation of gravity. This feature will introduce a new level of difficulty to the game-play as players will be able to move on any surface of the map. Searching for enemies will become more difficult and inevitably attempting to destroy the enemies will become increasingly more challenging as movement will not be limited to a single plane.

This document will provide technical specifications for the game software, the game play, the game engine architecture, and the game design. Also, there will be references of traditional FPS game design and standards throughout.

## 2.1 The Basic FPS Game Movement

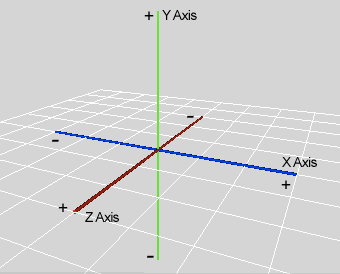


Figure 2.1 – Right handed 3D coordinate system.

Most FPS games have gravity locked to a single plane or vector direction to simulate realistic planetary gravity. This type of gravity references the negative (-) Y vector in right hand three dimensional (3D) coordinate spaces. Figure 2.1 shows a typical right hand 3D coordinate space and a wireframe plane that would in a FPS game make up the surface or ground mesh. Here, jumping or falling takes place along the Y axis and falling or the downward acceleration due to gravity would be in the negative (-) Y direction. Forward, backward, left, and right movement occurs on the plane made up of the X and Z axis. This type of gravity physics is easily understandable and translates well in a first person point-of-view game where you control a human like entity.

Movement for nearly all 3D games is generally calculated using 3D Vectors and Matrices. Game objects are oriented and positioned using a single point and two vectors. They are:

* POSITION – a point in 3D space defining a location for an object.
* FORWARD – a 3D vector with unit length that defines which direction is forward.
* UP – a 3D vector that defines which direction is straight up for the game object.

Using these three variables, a world matrix can be generated for use in scaling, rotating, and translating a game object. Translations are simply a movement of the POSITION point. For example, a jump action will generate a translation along the Y axis (the UP vector). A falling action, or downward acceleration of gravity, is a translation in the opposite direction of the UP vector. In a standard FPS game, the UP vector is constant and always points in the same direction. One note to this point is that a first person camera is able to pitch up and down which changes the cameras UP vector but not the game object’s UP vector. Rotation about the game objects UP vector will result in a changed FORWARD vector which will be reflected in both the game object and the first person camera.

Physics calculations in conjunction with defined movement keys and mouse motion will trigger translation and rotation of a game object and a camera. Keyboard and mouse controls for this game can be found in Appendix A.

## 2.2 “Hook” Gravity

This game is built on the basics described in the previous subsection with the addition of a new feature I refer to as “hook” gravity. The “hook” gravity concept is a slight modification of an idea discussed in a book by Orson Scott Card entitled *Ender’s Game*. In the book, two teams would battle each other using guns that freeze their opponents. The battles would take place in large zero gravity rooms with free floating obstacles. The “hook” was a tool used by team leaders to control their position in the zero gravity rooms without the need for momentum interference from the room walls or other objects. For this project, gravity will always be enforced, but each player will be able to “hook” themselves to any surface of the game map and allow that surface to provide the vector direction of gravity.

The technical aspect of this idea is reflected using the UP vector. When the UP vector is rotated the inverse of the UP vector, gravity, is shifted along with the orientation of the game object and the camera. The game object in the case of this game will be an individual player model.

Limits on UP vector rotation are limited to the six direction vectors extending from the origin of a 3D coordinate system. They are the positive (+) X, negative (-) X, positive (+) Y, negative (-) Y, positive (+) Z, and negative (-) Z vectors. This limit of six orientations of gravity is enforced because the map model used in this game is composed of rectangular shaped rooms and hallways with a mesh surface in every direction.

Custom keyboard and mouse controls for performing “hooks” were developed to blend easily with standard FPS game controls.

# 3 – Project Requirements

The requirements for this game were proposed by me, and they were evolved and devolved as necessary due to the difficulty of this game’s development. I chose a prototyping approach to assist in testing game aspects throughout the development process. Some of the initial proposed requirements were not completed, but the core idea, “hook” gravity, was fully implemented. The functional specifications outline the areas of the game that are completed.

## 3.1 Functional Specifications

1. The game displays a title screen that can be reverted to during game play.
2. Game data for each individual player will consist of health, weapon status, number of kills, and a world matrix that defines position and orientation.
3. When game play begins, the player will see a first person view of the map world and will be able to navigate in all directions on the plane in which they are oriented using a combination of keyboard and mouse input. An appropriate model animation will reflect the current movement.
4. A heads-up-display will be shown including the player’s health, weapon status, and a crosshair. Health is scaled from 0 to 100.
5. The player will be able to choose from seven different crosshairs to display in the center of the screen.
6. The player will not be able to run or fall through any surface of the map. (i.e. collision detection on map surfaces)
7. The player may change their orientation of gravity and move on the newly oriented surface using keyboard and mouse input. This specification defines the “hook” gravity concept.
8. The player will be able to shoot projectiles in any direction in an automatic fashion using mouse input.
9. The player’s weapon status will update to reflect it is current heat condition as will the weapon status indicator on the heads-up-display.
10. The player will be restricted from firing projectiles if the weapon status reaches the overheat condition.
11. Projectiles can cause damage at a rate of 10 health points per hit on the body and 100 health points for hits on the head.
12. When a player’s health reaches 0%, the player will no longer be able to move and an appropriate animation will be displayed for the death condition.

## 3.2 System Specifications

The game was developed using Microsoft XNA Game Studio 3.1, Microsoft Visual Studio 2008, and Autodesk Mod Tool for XNA. The game requires a computer running Windows XP SP3, Windows Vista, or Windows 7 as the operating system. Also at a minimum, the computer will require Microsoft .NET Framework 3.5, a graphics adapter supporting DirectX 9.0c and Shader Model 1.1, XNA Framework Redistributable 3.1, 1 GB of memory, and a CPU running at 1.0 GHz. Because a full installable game has not been released, a full install of XNA Game Studio 3.1 must be installed to build and run the game.

# 4 – Game Design

Microsoft’s XNA Framework provided the foundation of the game with a class library designed specifically for creating Windows and Xbox games. This game was strictly developed for a Windows based PC but could be expanded to allow support for the Xbox. The built in XNA Framework also provided a content pipeline for handling run-time loading operations and updating of game content. A custom model processor built by Autodesk Softimage was used to import models using the XSI model format.

## 4.1 Game Sprites (2D)

The game crosshairs and weapon status indicator were the only two dimensional game sprites that were custom built. Microsoft Paint was used to design each crosshair sprite and each different weapon state sprite. Figure 4.1 shows a few examples of each of the custom sprites.

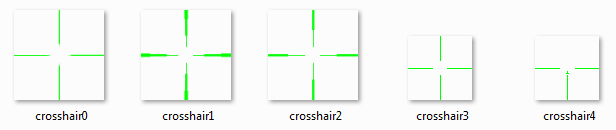




Figure 4.1 – Crosshair sprites and weapon status sprites.

## 4.2 Game Models (3D)

The 3D models used in this game were modified and/or developed using Autodesk Softimage Mod Tool version 7.5. This tool was a free download from the Autodesk website and open for use on non-commercial games. All the 3D models used in this game were exported using this tool.

The 3D player model used was an older Valve Half Life model that was freely downloaded from FPSBanana.com. The model, textures, and most of the original animations were imported into Softimage Mod Tool and modified to conform to XNA model requirements. This and all the other game models are rendered using Direct X 9. Figure 4.2 displays the player model in the Softimage Mod Tool environment. While the hands of the model are formed for holding a weapon, no weapon model was imported to save time in the modeling stage of development.



Figure 4.2 – The 3D player model.

The map model was obtained from and XNA Game Studio partner called Turbo Squid. Turbo Squids website provides many free 3D models that are free to download and use in non-commercial games. The map model was also imported into the Softimage Mod Tool and modified to conform to XNA requirements. Most of the map meshes were retextured with higher resolution texture files. The inclusion of these new textures did not significantly increase and processing requirements. The model for the game map was carefully selected to suit the “hook” gravity concept. Figure 4.3 is a bird’s eye view screenshot of the map model displayed in the Softimage Mod Tool environment.

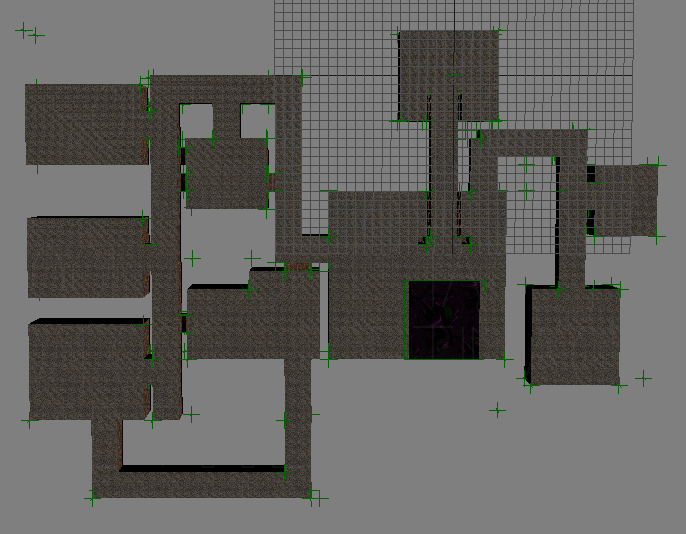


Figure 4.3 – The 3D map model.

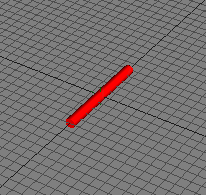


Figure 4.4 – The 3D projectile (laser) model.

A simple cylinder mesh was model from scratch for use as the 3D projectile (laser). A real time Phong shader was applied to the mesh and tinted to give the cylinder a red shiny look.

## 4.3 Class Objects and Data Structures.

XNA game development is geared for object-oriented design, so many of the predefined XNA Framework objects were used to develop this game. Specific objects that were used include:

* Vector3 – a simple 3D vector object.
* Matrix – an object with many useful methods specifically for use in computer graphics.
* Model – an object used to hold the DOM data for models.
* Texture2D – used for storing sprite texture data.
* BoundingBox – a box object with methods for performing collision detection.
* BoundingSphere – a sphere object with methods for performing collision detection.
* TimeSpan and GameTime – classes with methods for doing game time calculations.

Autodesk’s custom model processor came with classes for supporting animations and a custom data container for correct rendering of the exported XSI model objects.

Also, class structures were developed for a Player object and a Map Asset, a Model Asset, a Camera Asset, and a Bullet Object. Player objects contain all types of assets except the Map Asset which is held by the default XNA Game class.

For this game, Array and List data structures are used for pooling different class objects. The choice for either using a List or an Array for object groups was based purely on what I determined would be the best way to access and manipulate the data in those structures.

The generic Game class contains a List of Player objects although only one Player is added to the list in this implementation. The List<Player> was implemented to allow for multiplayer functionality which is not completed to date. The Game class also includes a global array of BoundingBox objects which are built using the model bones of the map object. Each bounding box in the array corresponds to a surface of the game map. This array is initialized before the game begins and is static once instantiated.

The Player class also contains many data structures. First, the map bounding box array is copied in the Player class to enhance collision detection performance. Also stored in this class is an array of BoundingSphere objects. The bounding spheres array contains all the necessary bounding spheres that are needed for performing collision detection with walls and bullet objects. Additionally, an array of BulletObjects is needed to allow data to be stored about each individual projectile that is live and active. Within the ModelAsset class of a Player object, a List of Animations is stored. Currently there are six (6) animations and/or model poses loaded in the List of animation content objects.

## 4.4 Game Architecture and the XNA Content Pipeline

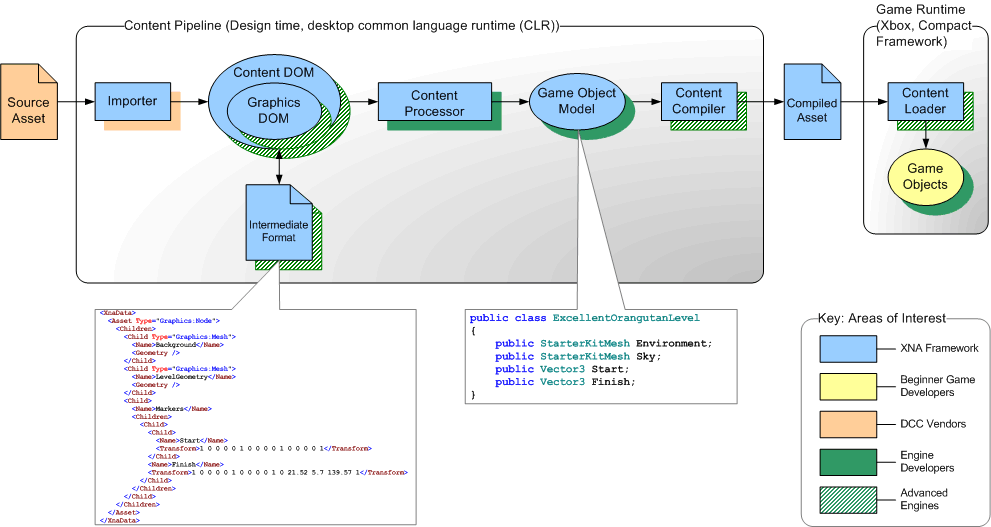


Figure 4.5 – The XNA Content Pipeline for importing assets and runtime loading.

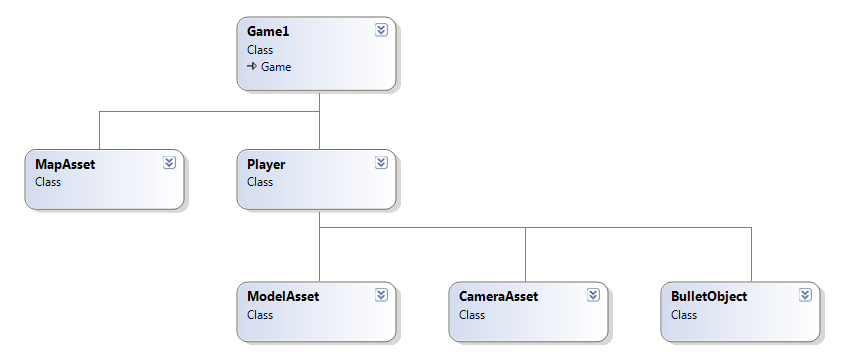


Figure 4.6 – Hook class diagram.

The XNA Content Pipeline helps to simplify the game development process. The standard XNA importer supports FBX files and the Direct X file model format. As mentioned, a custom importer designed by Autodesk was used to streamline the integration of the modeling software with an XNA game project in Visual Studio.

The structure of the game is very simple. The Game class is generated when a new XNA project is created, and it controls all the necessary routines for runtime operations, everything from initialization and content loading, to updating and drawing.

## *4.5 Physics and Algorithms*

The game physics such as acceleration, velocity, and gravity were developed simplistically, but the physics calculations look and feel realistic. Velocity is simply calculated by adding an acceleration component over time and clamping it to a maximum value. This applies to both “ground” movement and “jumping/falling” movement. An optional improvement to this game could be to use a separate code package for handling game physics.

“Hook” gravity is calculated using the inverse of a Player object’s current UP vector. The user controls this vector and if the appropriate keys are pressed an algorithm will determine the new UP vector for the player. Two possible “hooking” conditions currently exist:

1. “Hook” on a wall to the left, rotating feet first to the left.
2. “Hook” on a wall to the right, rotating feet first to the right.

Determining the best new UP vector is contingent on the direction of the current FORWARD vector. The algorithm for “hooking” to the right is as follows:

Largest = (The X, Y, Z component of the Forward vector with the greatest magnitude);

if (xComponent is Largest)

if (xComponent > 0)

Next UP Vector is determined from the cycle: +Y, +Z, -Y, -Z;

else

Next UP Vector is determined from the cycle: +Y, -Z, -Y, +Z;

if (yComponent is Largest)

if (yComponent > 0)

Next UP Vector is determined from the cycle: +Z, +X, -Z, -X;

else

Next UP Vector is determined from the cycle: +Z, -X, -Z, +X;

if (zComponent is Largest)

if (zComponent > 0)

Next UP Vector is determined from the cycle: +X, +Y, -X, -Y;

else

Next UP Vector is determined from the cycle: +X, -Y, -X, +Y;

The same algorithm is used for “hooking” to left except the cycle in each case is reversed. A similar condition for performing a “hook” to the wall directly in front of an object was designed but never implemented. In this case the “hook” rotation would happen feet first in the forward direction. The new UP vector would be determined from the inverse of the absolute Forward vector.

The Algorithms for performing collision detection are already implemented in the standard XNA BoundingBox and BoundingSphere classes. Each of these two objects is capable of detecting a collision with the other. Sphere to sphere collisions are the most efficient calculation because it only involves two points, and two radius vectors. The BulletObjects and Player objects use spheres as bounds and since projectiles are moving very fast, the quick sphere to sphere collision detection algorithm is ideal. Future efforts will be made in minimizing the number of collision detection calls with each update, but the current design is able to run smoothly and accurately.

# 5 – Software Development Process

For this project, development was done using a prototyping methodology. Also, the modeling process was conducted using a similar prototyping method. I defined and implemented a single requirement for each prototype phase. Some of the proposed requirements were not implemented because the prototype had not evolved to level were the requirement was needed or could be applied.

## 5.1 Testing and Debugging

Testing the game was relatively easy since I constrained each prototype to one new design aspect. This allowed me to focus the testing on one new area of code or game play at a time. Debugging, however, was not so simple, and not only did I have to handle debugging game code, in the early stages there was a significant amount of modeling adjustments that had to be made. The modeling turned out to be the most difficult aspect of the development process even though the models were imported from outside sources.

A specific area of debugging that I had trouble with was fixing the necessary movement and camera adjustments for the “hook” gravity. Often, the camera and/or model would, after changing gravity orientation, become flipped or be displayed in the inverse of the desired direction. Using the Visual Studio debugger and stepping through the sections of code involved with gravity helped identify the problems and sparked the ideas for the current implementation of “hook” gravity which functions as expected.

## 5.2 Benefits of Prototyping

I found using prototyping to be extremely beneficial for my first time doing game development. Phasing each design element allowed me to apply my focus to small sections of code at a time. Also, prototyping was great for the development of the “hook” gravity. I began the earlier phases setting requirements for the standard FPS game physics, and afterwards, introduced “hook” gravity in a single phase of development. Finding an ideal control schema for performing “hook” orientations was prototyped as well. Multiple different setups were tested in the process, and I was able to settle on a control system that seamed effortless to use for any user with experience playing an FPS game.

## 5.3 Work Breakdown

I expect this project to be very time consuming, and indeed it was. I planned to work on the project for at least 16 hours every week and I ended up averaging nearly 32 hours a week of work. The high hourly average can be contributed again to the difficulties in the modeling phase, where I had to modify both the map and player models to conform to XNA and the custom XSI importer. Using the custom XSI model processor was necessary for loading the multiple animation sequences and model poses that are current in the latest prototype. Figure 5.1 below closely represents the development process. As can be seen, some areas of the initial design requirement were never accomplished.

Figure 5.1 – Actual Project Timeline.

# 6 – End Result

This game *Hook* is still currently being developed. I have come to enjoy working on this project and will continue to do so for recreational purpose. Currently, the game does not have networked multiplayer support. The current prototype allows for all the requirements described in Section 3 of this document. Also, a batch build has not been released for the current prototype and running the game requires adherence to the system specifications also outlined in Section 3 of this document.

## 6.1 Final Prototype

Several screenshots of the final prototype are displayed below. The “hook” gravity concept can be seen in these screenshots as well.

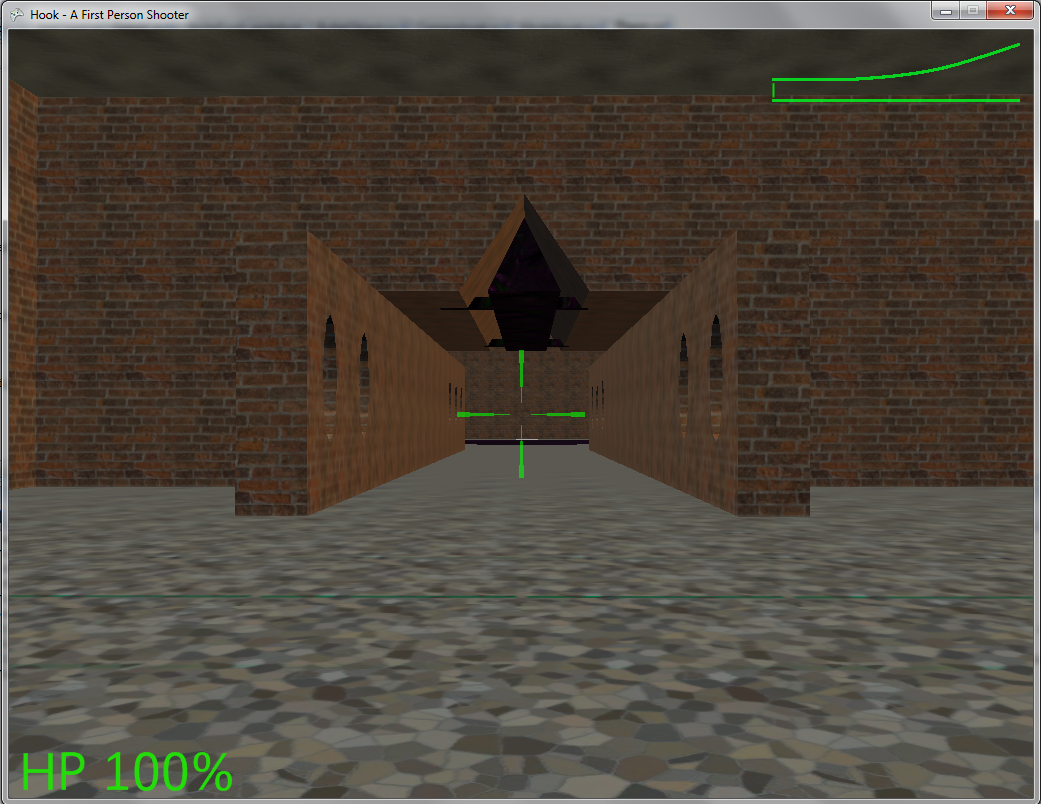


Figure 6.1 – Hook screenshot with default standard gravity in use.

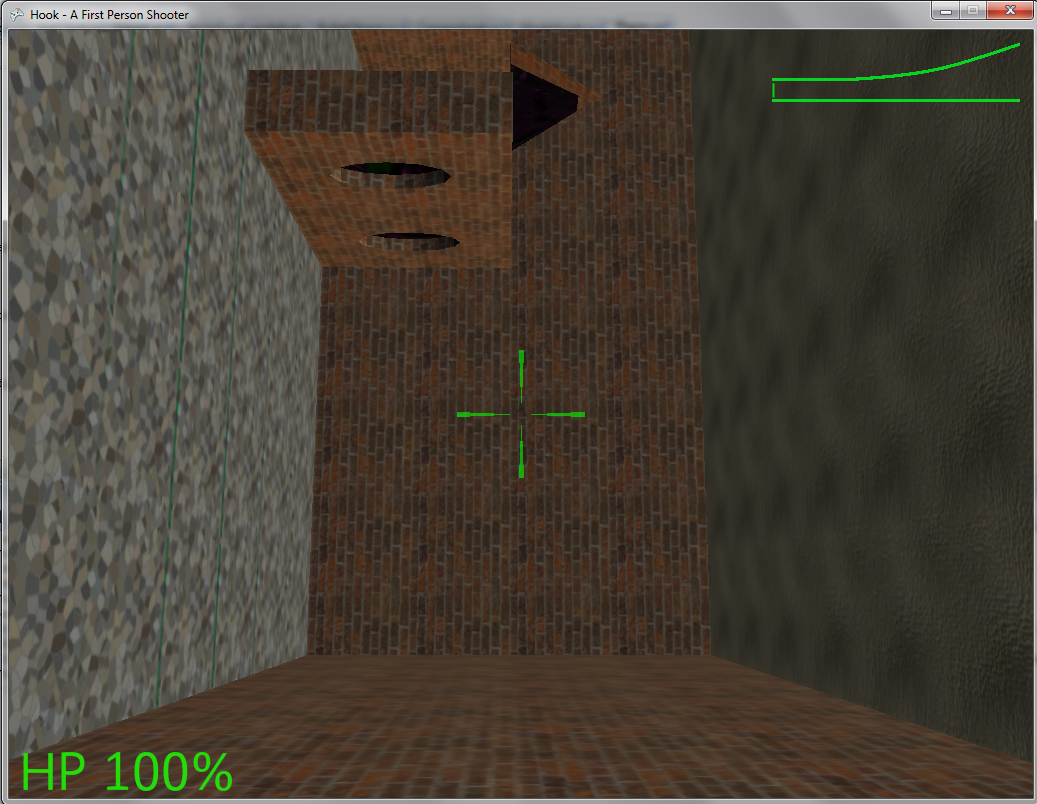


Figure 6.2 – Hook screenshot with a shifted gravity orientation.

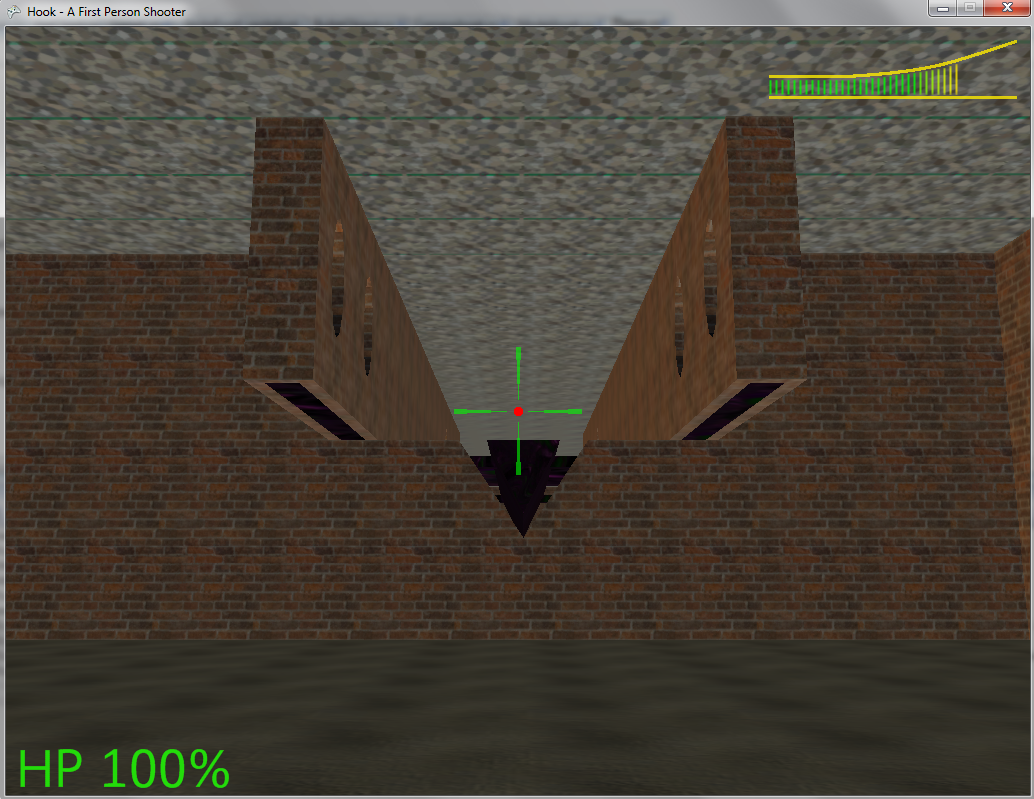


Figure 6.3 – Hook screenshot shooting a bullet object and oriented on the ceiling.

## 6.2 Future Releases

My own enthusiasms for this game will inevitably drive me to reach the goals originally set forth in my proposal. I feel the core idea of the game, “hook” gravity, was implemented successfully but the game in its current form is far from complete considering that no interactive game play has been implemented or tested.. Currently, developments are being made to include custom host/client networking that will allow a batch install on any Windows based PC without the need for a Windows or Xbox Live Account.

# 7 – Summary and Conclusion

The goal for this project was to create a unique First Person Shooter game using Microsoft’s XNA Game Studio. The game introduced a new concept to the standard FPS game model, “hook” gravity. The “Hook” gravity concept offers a unique point of view for players and will hopefully provide an exciting challenge for players of the game. I look forward to developing this project further to meet the goals outlined in my original proposal.

I encountered many challenges in the development process including many difficulties implementing and importing game models. I feel my skills developing in the C# language and in the Visual Studio IDE have grown tremendously. Core graphics knowledge learned in previous CS course at UAA proved to be very useful and limited the creation of bugs in many areas of the game code. Game development is extremely exciting but also extremely time intensive. I can now fully understand why game development today requires a high amount of man power.

The skills and knowledge I have acquired working on this project will be remembered and built upon in future endeavors. Game development is a new avenue or passion I have found, and I can seriously say that I would enjoy the challenge of creating another game in the future or seeking professional education this area of software design.

# Appendix A – User Manual

This manual will serve the purpose of defining the game controls for playing the current prototype of the game. The play the game you must have the ability to build and run XNA Game Studio projects in Visual Studio. Instructions for installing XNA game studio can be found on the XNA Creators Club website and in the Microsoft Developer Network.

System Requirements

* Windows PC (XP SP3 or newer Operating System)
* 1GHz Processor
* Separate Graphics Adapter supporting Direct X 9.0
* Full Install of Visual Studio and XNA Game Studio
* 1 GB of RAM
* A keyboard and mouse

When game builds and runs a title screen will appear with a star background and the game title *Hook* displayed at the top. Press the (F1) key to begin game play. At any time during game play, the (F2) key will return you to the title screen as will the (F1) key with the game screen.

Mouse and Keyboard Controls

* The mouse is used to perform “look-around” and rotation. The left mouse button when held down will initiate firing red lasers.
* W – move forward
* S – move backward
* A – strafe/move left
* D – strafe/move right
* Spacebar – jump
* Left Shift – walk
* Jump + Q – Hook left
* Jump + E – Hook right
* 1-7 – change crosshair
* Escape – exit/quit