AsteroidBlaster

By Ryuho Kudo

California Polytechnic State University

Winter 2011 – Spring 2011

Advisor: Prof. Zoë J. Wood

Table of Contents

Introduction

Role of 3D Games

Atari's Asteroids is one of the most popular arcade games of all time. Released in 1979, it has been one of history's most influential video games. The game works like this: The player controls a small triangular ship. There are asteroids floating around the world along with the player. The player's ship has a gun that is used to shoot the asteroids. When the asteroids are hit, they explode, breaking into smaller asteroids. Occasionally, an enemy ship will fly across the screen and shoot at the player. The object of the game is to get the most points by shooting everything while avoiding being hit.

Over the years, Asteroids has seen hundreds of clones for many platforms. Some of these have included graphical improvements, while many have added new gameplay mechanics. *Asterax*, for example, is a shareware adaptation of Asteroids for the Macintosh. In *Asterax*, exploding asteroids have a random chance of dropping crystals that the player may collect and spend to upgrade his or her ship.

This project, Asteroid Blaster, is a new approach to Asteroids in 3D. It combines elements from several video game genres, including first-person shooters, flight simulators, and role-playing games. It is not the first 3D game in the style of Asteroids, but it combines fast-paced Asteroids-style action with attractive graphics and some new game mechanics, while still remaining fun to play.

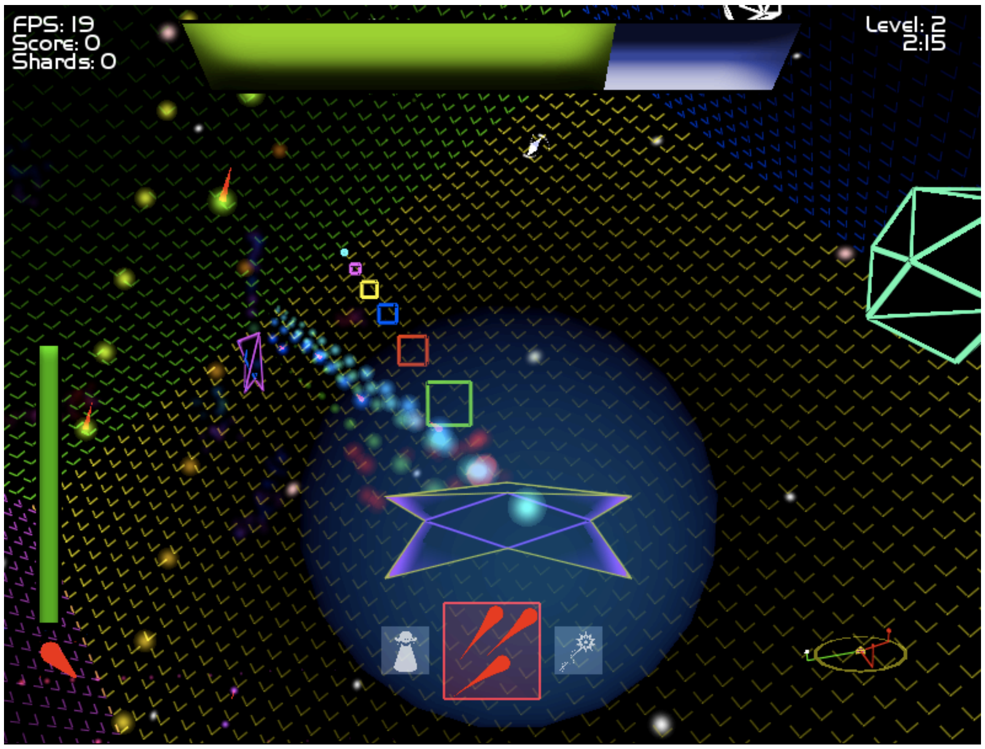
Course Structure

Asteroid Blaster is part of a Zoë Wood's CSC 476++ class. The project was primarily developed by a team of five students: Taylor Arnicar, Chris Brenton, Sterling Hirsh, Jake Juszak, and Ryuho Kudo. Additional AI work was performed by Sean Ghicel, Justin Kuehn, and Mike Smith.

Project Overview

Genre and Settings

Asteroid Blaster is a 3D space adventure game for players of all ages. The player can fly freely within a bounded environment in outer space and shoot asteroids, much like the classic arcade game, Asteroids. When asteroids are shot, they break apart into smaller asteroids, and sometimes they drop crystals that the player can gather. Players can fly to these crystals and touch them to collect them. The player's ship is also equipped with a tractor beam to attract crystals.



A ship is firing a weapon called Blaster to another ship.

The game takes place inside a multi-colored cube. The cube's faces are colored red, blue, green, yellow, cyan, and magenta. These colors help the player stay oriented in the world. There is no concept of "up" or "down" in the game, since players are free to pitch or roll however they please. Some players find this disorienting at first, but most players become comfortable with the world within the first several minutes.

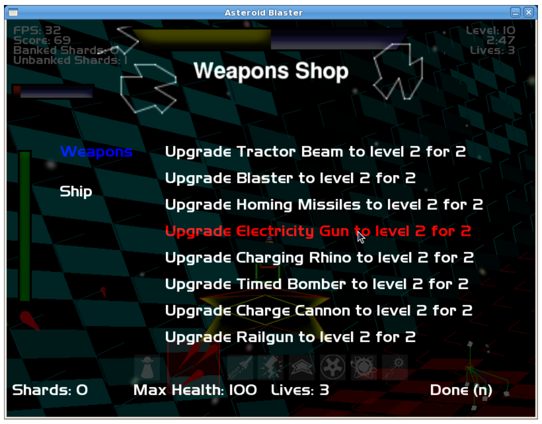
Objectives

The object of the game is to destroy all asteroids and collect all the crystal shards released while taking as little damage as possible. To this end, the player starts with two weapons, the Blaster and the Tractor Beam. The Blaster shoots many lightly damaging projectiles in quick succession. The The Tractor Beam projects a continuous cone from the ship, pulling in crystal shards that are inside it.

Asteroid Blaster is played in levels. A level ends after three minutes or when all asteroids and shards are collected, whichever comes first. The player has infinite lives, but must wait several seconds after dying before respawning. A new weapon is given to you after completing each level to encourage players to get to later levels. The game is over after eight rounds have been completed. This number may be adjusted for multiplayer.

At the beginning of each level, any remaining crystal shards and asteroids are cleared, and new asteroids are randomly created. The number of new asteroids is equal to the level number. That is to say, level one has one asteroid, level eight has eight asteroids.

Between levels, the player is taken to the store, where he or she spends crystals to purchase ship upgrades, such as better armor, weapon upgrades, or a faster engine. Players may continue to upgrade their weapons, however upgrades get more and more expensive as the player progresses through the levels.


Asteroid Blaster
FPS: 32
Score: 69
Banked Shards: 0
Unbanked Shards: 1
Level: 10
2:47
Lives: 3
Weapons Shop
Weapons
Ship
Upgrade Tractor Beam to level 2 for 2
Upgrade Blaster to level 2 for 2
Upgrade Homing Missiles to level 2 for 2
Upgrade Electricity Gun to level 2 for 2
Upgrade Charging Rhino to level 2 for 2
Upgrade Timed Bomber to level 2 for 2
Upgrade Charge Cannon to level 2 for 2
Upgrade Railgun to level 2 for 2
Shards: 0
Max Health: 100
Lives: 3
Done (n)

Players can upgrade weapons and ship in the Store Menu

Beginning with level two, players compete for crystals against a computer-controlled AI that fights against the player and shoots asteroids. Every level after level two adds an additional AI player. Asteroid Blaster also features networked multiplayer. Several players may join a game and play in the same world, similar to the way the AI players do during single player.

Look and Feel

We always had a clear art aesthetic that we wanted to get in our final product. We got our inspiration from games like Geometry Wards or Beat Hazard. These games had bright colors with dark background, saturated neon particles and visual effects, and retro art style that utilized wire frames a lot. We believe we have reached this goal in our end product.

Story

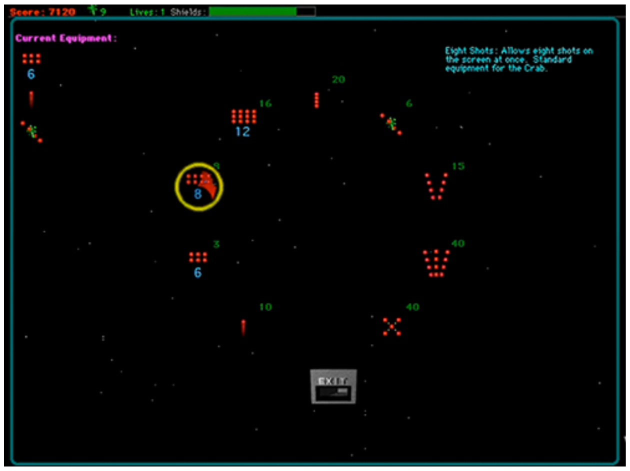
Our game is very influenced by retro games. Most retro games had sadistic difficulty and addictive gameplay but lacked in story telling for most of the part. The situation is similar in our game. The game doesn't give you a reason to shoot the asteroids so that they explode, collect shards, or blow up other ships. The player is simply thrown into the action without any context.

Technologies Used

We used C++ for our programming language since it was required by the class. We used g++ as our compiler since majority of people were using lab machines which had g++. The most natural choice for a graphics library for our environment is OpenGL. We used SDL for windowing, font, image, and audio libraries. Boost was used for threading, networking and serialization library.

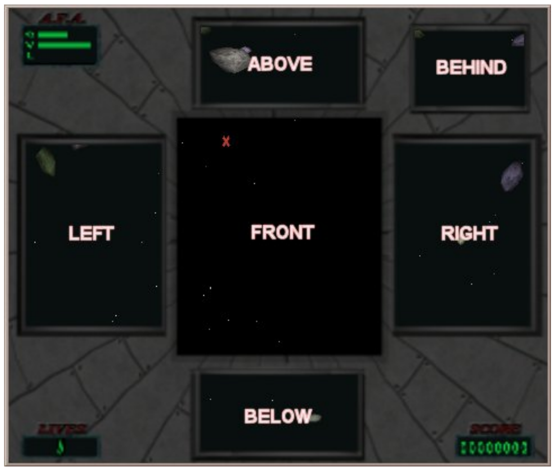
Related Works

A lot of games have influenced the design of Asteroid Blaster. The biggest influence was Asterax, by Arvandor Software. Asterax is a clone of Asteroids for the Macintosh featuring crystals that asteroids drop when they explode. When collected by a player's ship, crystals either replenish health of the ship or serve as currency for the in-game store. The store appears between levels and allows the player to purchase upgrades, items, points, and extra lives. Asterax is in 2D, however, and the goal of Asteroid Blaster is to create a 3D game. So a direct copy of Asterax is insufficient, but many of its ideas can be adapted to work in 3D.



Eight Shots: Allows eight shots on the screen at once. Standard equipment for the Crab.

Asterax's shop menu



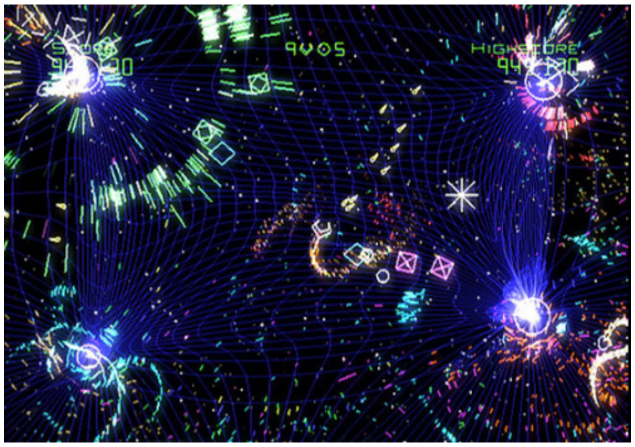
3D Asteroids' view scheme

A three-dimensional Asteroids clone exists, aptly titled 3D Asteroids by Grass Games. According to its website (grassgames.com/asteroids), it is the first 3D clone of Asteroids. 3D Asteroids maintains the dark, lonely feel of the original, but moves the game to a fully 3D environment. Unfortunately, 3D Asteroids has a confusing control scheme and an unintuitive view. Before playing the game, the player is forced to endure a 15-minute tutorial.

In part because of the confusing controls and view, 3D Asteroids loses the fast-paced excitement of the original. Additionally, 3D Asteroids retains the infinitely looping world of the original. A player may travel in a single direction indefinitely without being stopped by any walls. This works well in the 2D original, since the entire world may be seen at once, but in a 3D setting, players can (and do) run into unseen asteroids when looping around the edges of the world. Asteroids appear to pop into and out of existence when these boundaries are crossed, and it is easy to become disoriented.

Quake III Arena was released by id Software in 1999. It features a very clean deathmatch experience. There are plenty of weapons and several items. It had little story if any at all, focusing instead on gameplay. In Quake III Arena, players must kill everyone else while attempting to stay alive. Although the game does not play like Asteroids, it does share the same goals for the player. Also, Quake III Arena is very straightforward and easy to learn. This makes it a good model for several aspects of Asteroid Blaster.

Weapon effects in Quake III


Art style used by Geometry Wars

Geometry Wars has a unique visual style comprised of primarily lines and particles effects. This combines elements of retro games like Asteroids with more modern graphical techniques. This is largely what Asteroid Blaster is aiming for.

Algorithms Overview

Here are the list of technologies we implemented in our game. People who worked on it is listed in parentheses. This paper will talk about the bolded topics.

- **Network (Ryuho, Sterling)**
- **Menu System (Ryuho, Sterling)**
- **Input System (Mike, Ryuho)**
- Procedural Modeling (Chris)
- Bloom Lighting (Chris)
- Spring System (Chris)
- Multiple Render Targets (Chris)
- Ammo for weapons (Taylor)
- Design & interface of shooting AI & flying AI (Taylor)
- Shooting AI weapon selection (Taylor)
- Shooting AI target selection (Taylor)
- Shooting AI difficulty levels (Taylor)
- Weapon unlock system per level (Taylor)
- View frustum culling for player's view (Taylor)
- Radar (Taylor)
- Minimap (Taylor, Sterling)
- Text/Font (Ryuho, Taylor)
- SoundEffect/Music (Sterling, Ryuho)
- Weapon Upgrade (Ryuho, Sterling)
- Weapon Price / Balance (Ryuho, Sterling)
- Levels (Ryuho)
- Spectator Mode Camera (Ryuho)
- Particle System (Sterling, Ryuho, Taylor, Chris)

Algorithms Details

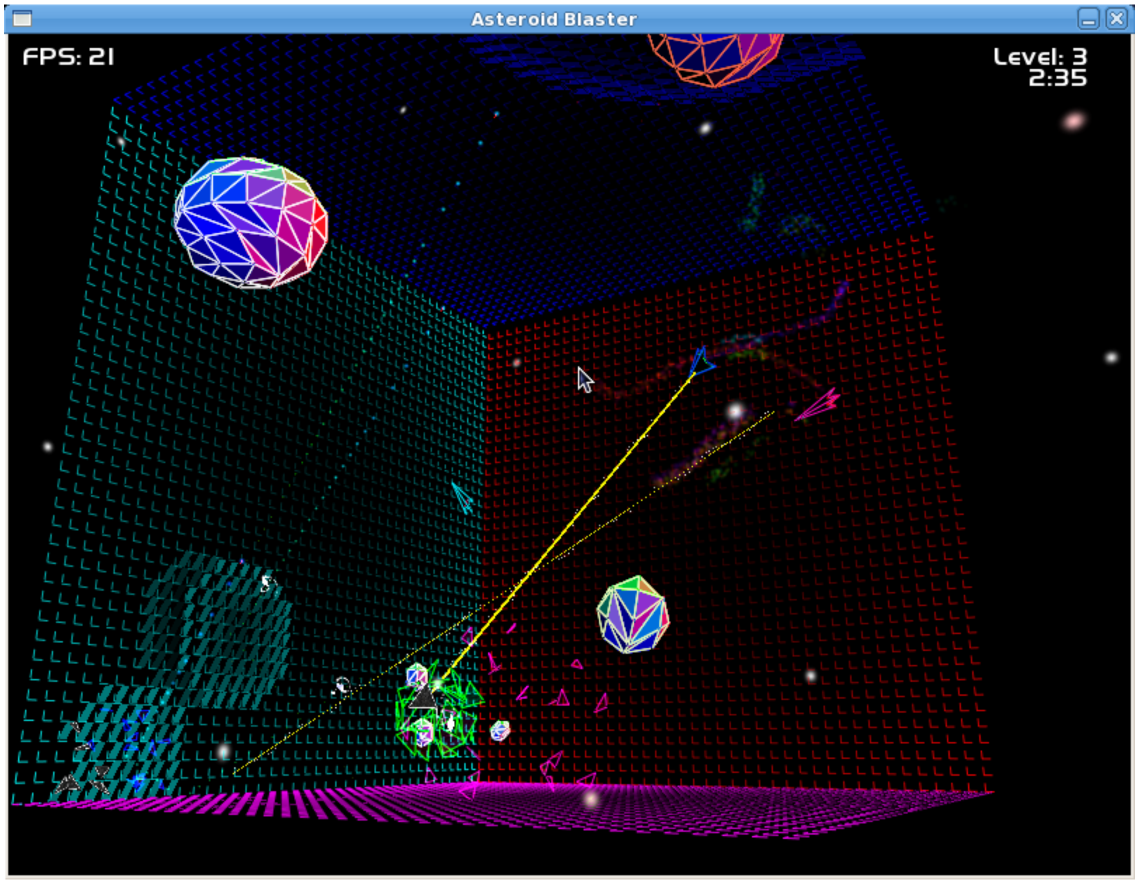
Libraries

Deciding what libraries to use in a project is an important part of the initial development process. The libraries used will define the program's limitations such as choice of OS, platform or programming language. Knowing this, we made a couple of decisions early on about what kind of libraries we wanted to use. We really wanted to use SDL instead of GLUT for several reasons. SDL is a window manage with not only graphics, but audio and input in mind. This is mainly due to the fact that SDL is designed for games and interactive programs. The original program Sterling made used GLUT extensively and we realized there were some overhead required to get our programs to use SDL exclusively. Limitations caused by GLUT such as keyboard input and lack of standardized modules to handle audio, sound, networking and fonts made us decide to use SDL as our main library. We were able to convert our project to only use SDL about half way through our first quarter.

The computer science department's lab did not have the necessary up-to-date libraries for this project. I feel compelled to talk about this in my senior project paper because it affected how we work on the senior project a lot. This situation forced us to compile Boost on our own directory since the lab's Boost library was several versions old. We also had to compile SDL, SDL_Image, SDL_Mixer, SDL_ttf on our own because it was not on our lab machines.

Networking

We hinted that we would have a multiplayer mode for this game when we pitched the game in our class. We ended up not implementing network related features in the first quarter. Later in the second quarter, we began implementing network related algorithms. Everyone on the team was content with server client model. The following subsections will discuss the problem we faced while working on the networking portion of our game and what we decided to do to try to solve it.



Server hosting two players

Serializing OO classes

Our program is very object oriented. Every thing drawable is derived from *Drawable* class and every collidable object is derived from *Object3D* class which its self is derives from *Drawable* class. As result, various member variables are hidden in these layers of abstraction. For example, position and velocity member are declared in *Drawable* class. However, the up, right, forward vectors, angle and rotation are in the *Object3D* class. This gave us the chance to reduce duplicate code when serializing.

Instead of manually coding to serialize position and velocity information on every kind of class we want to serialize, we can abstract out the process to a super class where it will tack on the information automatically. For example, when we are serializing Shard object, it only has to worry about Shard specific information such as rotation amount and rotation speed. Everything else is already serialized by Shard's super class which are *Object3D* and *Drawable*. This kind of serialization technique is supported by Boost's serialization library. Here is the code that is responsible for serializing *Object3D* class. This is in the file Network/Object3D.h:

```
// Serialization
public:
  template<class Archive>
      void serialize(Archive & ar, const unsigned int version) {
    ar & boost::serialization::base_object<Drawable>(*this);
    ar & id;
    ar & acceleration;
    ar & axis;
    ar & up;
    ar & right;
    ar & forward;
    ar & angle;
    ar & rotationSpeed;
    ar & yawSpeed;
    ar & pitchSpeed;
    ar & rollSpeed;
  }
```

You can see that the serialization process involves calling the serialization method owned by the super class (*Drawable* class).

Net Objects

Even with abstraction of serialization, serializing classes with pointers are very hard to do. For example, each *Drawable* class has a pointer back to the GameState class. If we serialize a *Drawable* class without considering pointers, we would create an instance of GameState class for each and every single class we deserialize. Creation of new instance for each pointer is how Boost handles pointers in serializer library. This is a problem because all of the *Drawable* class in a game is suppose to point to the same instance of GameState class.

To fix this problem we decided to create series of classes solely used for serialization. We would serialize primitive values and data collection classes and describe pointers with unsigned ints. This unsigned int will be looked up by the Custodian class which gives unique ID for each *Drawable* class. The serialized object would be sent across network, received on the other side, deserialized, and the unsigned int value will be looked up by the Custodian class and the appropriate pointer will take the place when the actual object is created on the other side. If the ID of the actual object which is being reconstructed does not exist, a new instance of that object is created with the information from the Net Object.

For example, a Shot class, which is derived from *Object3D* class has a member called owner which is a pointer to *AsteroidShip* class denoting who shot the object. Since each *Object3D* class has a unique unsigned int ID, that pointer to *AsteroidShip* can be described by just remembering an int. When serializing, an unsigned int is serialized and when being deserialized on the other side, the ID lookup will provide the pointer to AsteroidShip.

UDP vs TCP

When working on a networking program, you have the choice of sending data over network through UDP or TCP. After doing some research, we concluded that UDP was our best choice for several reasons. UDP protocol is fast and unreliable protocol. Video games often require fast connection because it often requires immediate reaction. The unreliability of UDP protocol could have been bad but we send redundant data so it doesn’t really matter. TCP is reliable connection oriented protocol and thus is a bit slower than UDP. Each packet sent through TCP requires that an acknowledge packet be sent from the receiver. This would create lots of latency problem and by the time data recovery happened, the packet’s information is out of date and is not of use. For these reasons, we decided to go with UDP. We also considered using both protocols but we realized that is a lot of work to set up both kinds of connection so we decided to use just UDP.

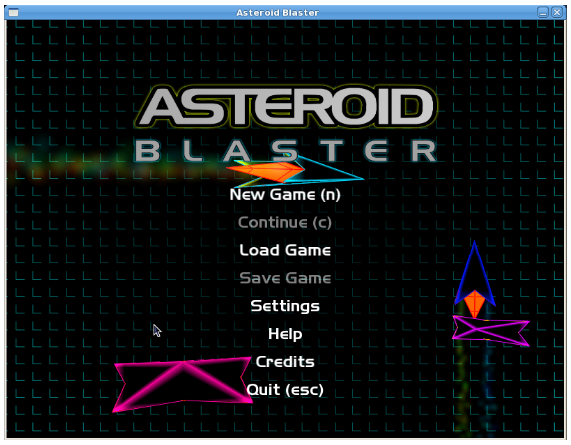
UDP_Client and UDP_Server Class

We had several choices for implementing a program that has Client/Server mode. We could have developed a separate program from single player program that only has server or client code. Separate program in this context means creating different mains in different cpp files. We were planning to have Single, Client, and Server programs. We also had the choice of only developing one program and switch casing behavior depending on which mode it was. The advantage of separating two independent programs is that bugs and defects are independent of each other. A bug that is introduced into the server program does not affect client directly. The disadvantage of separating server and client code is that development of the game now has to keep track of multiple mains and enhancements and bug fixes has to happen in more than one location. These disadvantages made us decide to combine all of the programs into one with multiple modes.

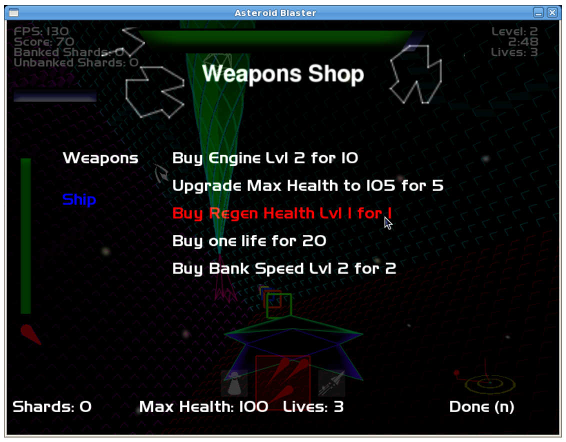
To support server and client code in one program, we made a *UDP_Server* and *UDP_Client* classes. the *UDP_Server* enables the main program to receive packets from clients and either specify the client to send packets or send packets to all clients connected. Both classes use AISO library from Boost and thus is threaded which runs concurrently with the main game code.

Menu System

Our game has purchasable and upgrade-able weapons and ships. This requirement by its self was not a problem since we just had to hold an integer for each weapon signifying its level. The game would then calculate the various properties of the weapons such as firing rate, damage and range depending on the level. However, we did have to figure out how to enable players to buy these options. A menu system was obviously needed. We already needed one for the main menu so we created a *Menu* class that is very configurable which is a super class for *MainMenu* and *StoreMenu*.



The main menu



The store menu

Extending the *Menu* class makes it easier to create GUI that is easy to navigate. We created credit’s page and help menu utilizing the same class. The moue hover and click detection is delegated to the *Text* class where it has functions that take in the x and y value. From these values, it detects whether or not the text should change color or call functions to invoke action.

The weapon classes and *StoreMenu* class works closely together to automatically detect newly implemented Weapon classes and add them to the store menu with out any changes to the code. The *StoreMenu* requests the upgrade text message to display for each weapon and how much each weapon related item costs to the *Weapon* classes so that *StoreMenu* class does not need any change.

Input System

Handling input is an important part of writing interactive programs. The general structure of input handling in C++ programs are usually the same. The input is handled in the main loop using SDL or GLUT and, depending on the input, a switch case statement parses through the user input to invoke a reaction from the program.

We wanted an organized way to do this for multiple classes that behave differently. For example, *GameState* class needs input from the user when the game is actually run. Menu classes are another examples that needs user input in order to navigate through the menu. To solve this problem, we derived every class that needs input from the *InputReciever* class. Each class implements a virtual function such as the following:

```
struct InputReceiver {
  virtual void keyUp(int key)=0;
  virtual void keyDown(int key, int unicode)=0;
  virtual void mouseDown(int button)=0;
  virtual void mouseMove(int dx, int dy, int x, int y)=0;
  virtual void mouseUp(int button)=0;
};
```

These functions are implemented in the classes that derive *InputReciever* and handles the actions accordingly.

A single *InputManager* class exists in our program which handles all incoming inputs and distributes them to the *InputReciever* classes. This is done by ‘adding’ an *InputReciever* to the *InputManager* class initially so that when the main loop comes along and detects an SDL input, the *InputManager* class will loop through all the *InputReceiver* classes and invoke the appropriate methods such as mouseMove or keyUp with the input information needed.

This enabled us to simplify input handling for a lot of the classes. Adding additional classes that require user input is fairly easy because of this. In fact the Menu system utilizes this design and thus does not need to know anything outside of the Menu class to carry out menu operations.

Multiple OS Support

Supporting multiple operating system was a feature we really wanted to implement. Multiple OS support is important because it enables us to deploy our game to a wider audience.. Our team developed mainly on Fedora and OSX but also was able to compile in Ubuntu. Our game was naturally cross platform between Red Hat, Debian, and OSX platforms. Gaming on computer is mostly done in Windows so we naturally really wanted to support the platform. We specifically picked platform agnostic libraries so that cross platform support is always possible to implement for our project.

Compiling on g++ and compiling on Visual Studio C++ turned out to be fairly different and it caused quite a few problems. After a while, we were successful in compiling the project on Windows with Visual Studio 2008. Unfortunately, it is extremely unstable and is not ready to be played by a user. This feature is wanted by most of our team members but it is not critical so the implementation of stable game on Windows will most likely be pushed back to after this project is completed. We might even try to sell our game on game store platform such as Steam if we get the chance to polish our project.

Results

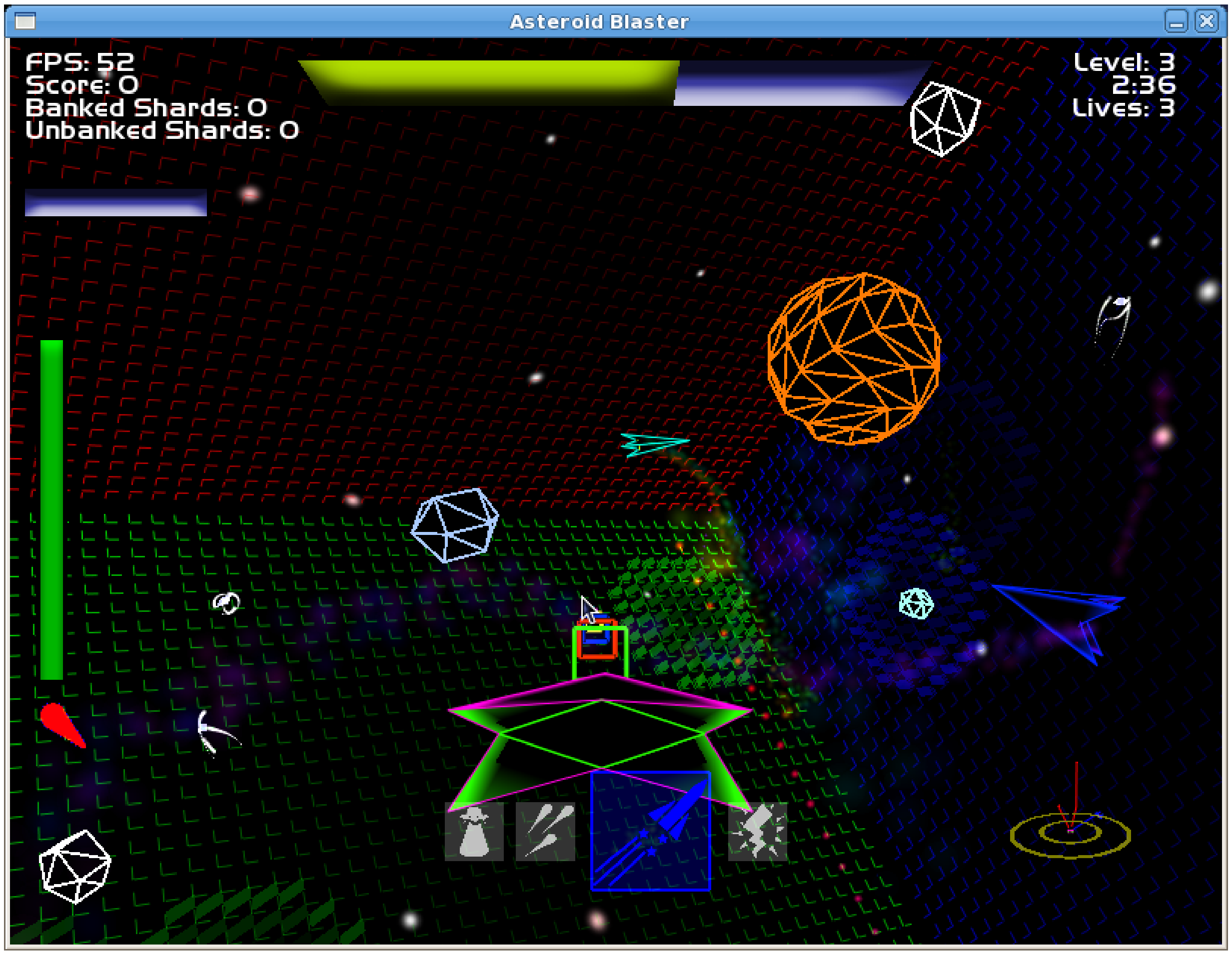
Asteroid Blaster turned out to be a fun, playable, single-player game. The game's graphics have an arcade feel that fits well with the style of gameplay. There are enough levels that a player could reasonably play for over an hour, and the game would still offer an increasing challenge at each level.

Throughout *Asteroid Blaster*'s development process, the development team had friends, family, and other students play test it. Nearly all who played agree that *Asteroid Blaster* is pretty. The glow squares received more positive feedback than any other single part of the game. Most players also agreed that the early levels were appropriately easy for new players. Several players thought the game did not get difficult fast enough, so we tweaked the AI to make it self-adjust to the player's performance. Now, when people play, they frequently report that the AI provides a challenge without being impossible.

When some people tested *Asteroid Blaster*, they would always respawn three seconds after dying. Players complained about feeling like there was no point to playing, since the game did not end, even if the player performed poorly. To address this, we added the concept of lives to the game. This gave players a way to lose, which gave them an reason to care about playing. This single change greatly contributed to the fun of the game.

Regarding the development process, this project was a great success. We used the Trac ticketing system with Subversion for task management and revision control. Over the course of two quarters, *Asteroid Blaster* saw about 1000 commits to its Subversion repository and over 200 Trac tickets. After the final version was created, we used the Gource repository visualization software (pictured) to create a video representation of our commit history. This video is available from the project web page.

As a team-based school project, *Asteroid Blaster* was somewhat unique. Where most school projects focus on achieving one or more goals specified by an assignment, this project allowed our team to decide what challenges to accept and to what degree. This gave us an enormous amount of freedom to experiment with the graphics and gameplay without worrying about whether or not our work met the requirements of the project.



Game play screenshot with two AI ships on the screen

Conclusion

I thought I knew what it would be like to develop a video game with four other people because I have experience working in a team to develop a piece of software. I took the two software engineering courses offered in Cal Poly (CSC 308/309) before taking this class and 309 actually involved making video games for the course. I have also interned for a physics collaboration where we developed software with ten to fifteen people. However, the experience was very different from what I thought it would be. In retrospect, this makes sense because the past projects had very clear goals and specifications. This was not the case with our senior project since we were the ones deciding what features to implement.

I realized early on that deciding on core features is a major time sink. We argued for hours about how the game should be. For example, we talked about what the mouse x-axis should do. Personally, I wanted to turn left and right because that is how normal FPS games handles movement. However, others wanted to roll the ship because he wanted to model the control scheme from flight simulators. We went back and forth about what the majority of people are used to and who our target audience is. In the end, we decided to roll the ship when the mouse cursor moves left and right. Conflicts like these also shows the difference between individual motives and team objectives. We, as a team, had general goals that we wanted to achieve but each member also had opinions as to what our final project should be.

Splitting the work was not difficult since everyone specialized in separate technologies. I specialized in networking and menus, Taylor specialized in AI, Chris specialized in graphics and so on. We had trouble with bugs once our code got complex enough. We had graphic overlay function break AI code and many other weird combinations of features that interfered with each other. Declining code readability of our source code is partly responsible for this problem. We were aware that as the project gets bigger, it is more difficult to navigate through source code and understand it in a reasonable amount of time. I think everyone experienced this first hand with our project.

Future Work

Time constraints and difficulty implementing technologies like networking and deferred shading hindered our ability to put all the features. Networking was not completed because we decided to focus on single player game experience. Deffered shader was not completed because of time constraints. Our code still has lots of TODO tags for optimization and stabilizing the program. Our trak page still has some left over tickets. Taking care of these incomplete tasks will give this project the polish it needs. We have yet to decide weather or not network is feasible because of available man power after this quarter. Our group member responsible for graphics expressed interest in finishing the deferred shading algorithm.

Aside from left over tasks from this quarter, there is another important choice we have to make. We have to decide weather or not if it is possible for us to polish this game enough so that we can sell it as a product. This requires additional content to be put into the game and very rigorous testing to make sure that our project has high enough quality. We also have to port this game to Windows OS if we want to sell this project with any success. The resource and time needed to carry out these tasks are a bit much so we are still considering the options. We really do want to continue working on it but we are not sure what direction we want to pursue as of now.

References

Particle System

http://nehe.gamedev.net/data/lessons/lesson.asp?lesson=19

Particle system was based off of this NeHe lesson. It also provided the source code in every combination of language, OS, and library imaginable so this will surely be useful to many people working with other development environment.

GLSL

http://www.evl.uic.edu/aej/594/lecture01.html

This website really helped us when implementing GLSL for the first time. It explained what GLSL does, the alternatives to GLSL such as HLSL/Cg and gave examples of using GLSL.

Server/Client model using Boost

http://www.boost.org/doc/libs/1_36_0/doc/html/boost_asio/tutorial.html

This page had lots of great examples on how multi-threaded TCP and UDP servers and clients are suppose to be implemented. Our server/client code is heavily based on this tutorial.

Serilization using Boost

http://www.boost.org/doc/libs/1_46_1/libs/serialization/doc/tutorial.html

This page described how to serialize using Boost very clearly with some examples and explanation. Important concepts such as serializing derived classes, pointers, arrays, STL collections, class versioning were covered.

Game Development Articles and Tutorials

http://gafferongames.com/networking-for-game-programmers/

This blog helped out a lot in deciding what protocol to use for network. It covers topics such as UDP vs. TCP, Virtual Connection over UDP, Reliability and Flow Control, Debugging Multiplayer Games. This blog also talks about physics, physics in 3D, and physics with networking which can be used in the future.