# Asteroid Blaster

By Sterling Hirsh

California Polytechnic State University

Winter - Spring 2011

Advisor: Zoë Wood

## Table of Contents

## Introduction

Atari's *Asteroids, released in 1979,* is one of the most popular and influential arcade games of all time. The game works like this: The player flys a small triangular ship around a world with floating asteroids. 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.

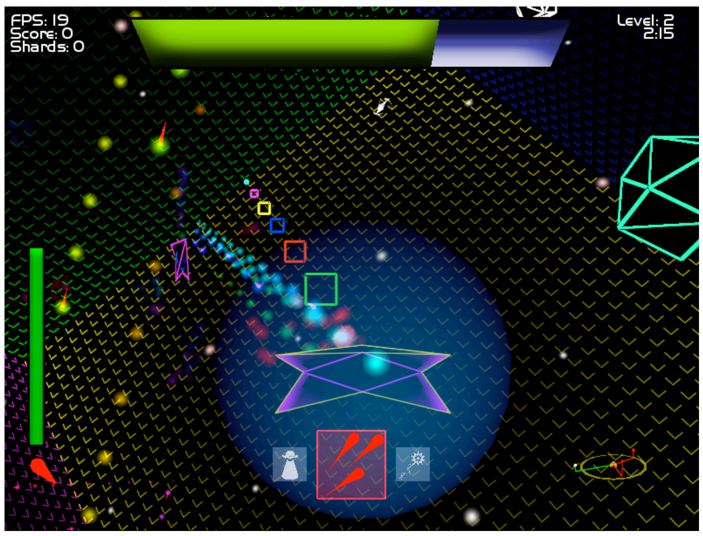
Since then, *Asteroids* has spawned hundreds of clones for many platforms. Some of these have included graphical improvements, while many have added new gameplay mechanics. *Asterax*, for example, is a shareware adaptation of *Asteroids* for the Macintosh. In *Asterax*, exploding asteroids occasionally drop 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.

## Course Structure

Asteroid Blaster is part of Zoë Wood's two-quarter CSC 476++ class, which is a computer science technical elective that also serves, for many students, as a senior project. 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 Ghiocel, Justin Kuehn, and Mike Smith.

## Genre and Setting



*Asteroid Blaster* is a 3D space adventure game for a general audience. 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 by flying into them. The player's ship is also equipped with a tractor beam to attract nearby crystals.

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 and roll however they please. During testing, some players found this disorienting at first, but most players became comfortable with the world within several minutes.

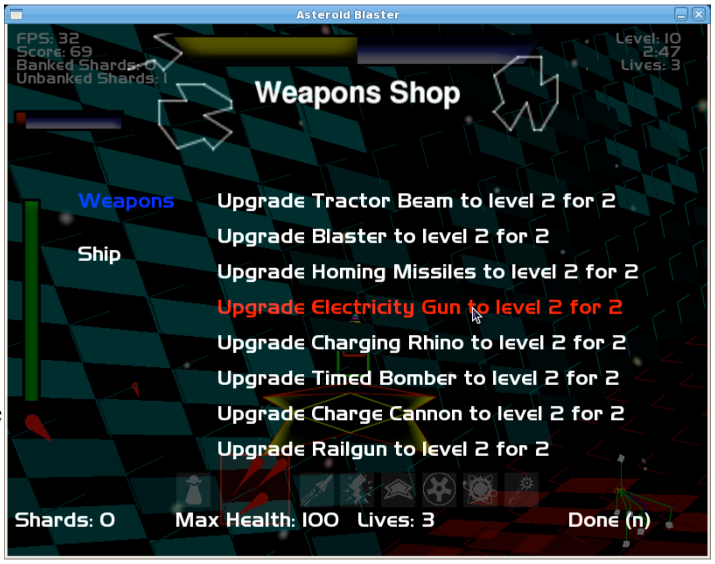
## Objectives

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 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 begins with three lives, and must wait several seconds after dying before respawning. Players receive a new weapon after completing each level to add an incentive to reach later levels. The game is over when the player has lost all his or her lives.

At the beginning of each level, any remaining crystal shards and asteroids are cleared, and new asteroids are randomly created. That is to say, level one has one asteroid; level eight has eight asteroids.  Level eight and beyond, only eight asteroids are spawned. Because the size of the world is constant, a level with more than eight large asteroids is virtually unplayable.

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)

Beginning with level two, players compete for crystals against a computer-controlled AI player that fights against the human player and shoots asteroids. Every level after level two adds one or more additional AI players, up to four per level. An AI player will stay in the world across levels and respawn after being killed until it is killed three times.

## Look and Feel

Since the beginning of the project, we always had a clear art aesthetic in mind, inspired by games like Geometry Wards and Beat Hazard. These games have bright colors, saturated neon particles, and retro wire-frame art styles.

## Story

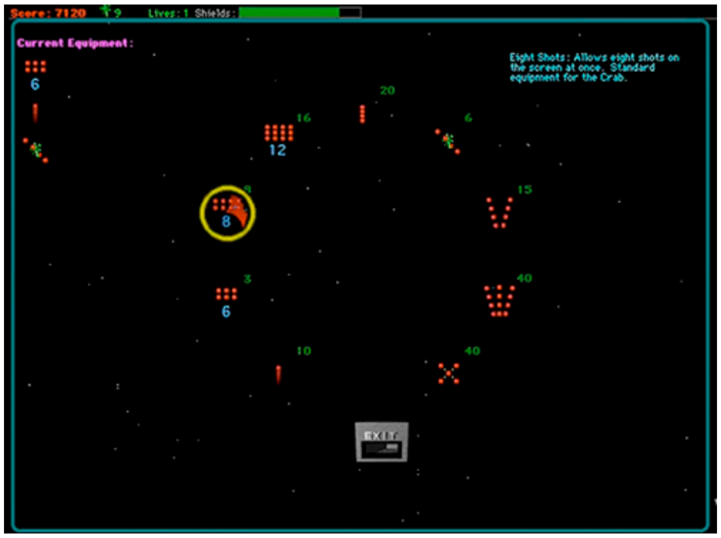
Our game is very influenced by retro games. Most old arcade games have sadistic difficulty and addictive gameplay, but lack a fleshed out story. *Asteroid Blaster* is no different. The game gives no explicit reason to shoot the asteroids, collect shards, or blow up other ships. The player is simply dropped into the action without any context. This confuses some players initially, but due to the popularity of arcade games, most players are able to understand what they are supposed to do as soon as they understand the controls.

## Technologies Used

*Asteroid Blaster* is programmed in C++ as per the class requirements. Because it is free and commonly available, we used g++ as our compiler. The most natural choice for a 3D graphics library for our project is OpenGL, since it is free and portable. We used Simple Directmedia Layer (SDL) for windowing, font, image loading, and audio. Boost was used for threading, networking and serialization.

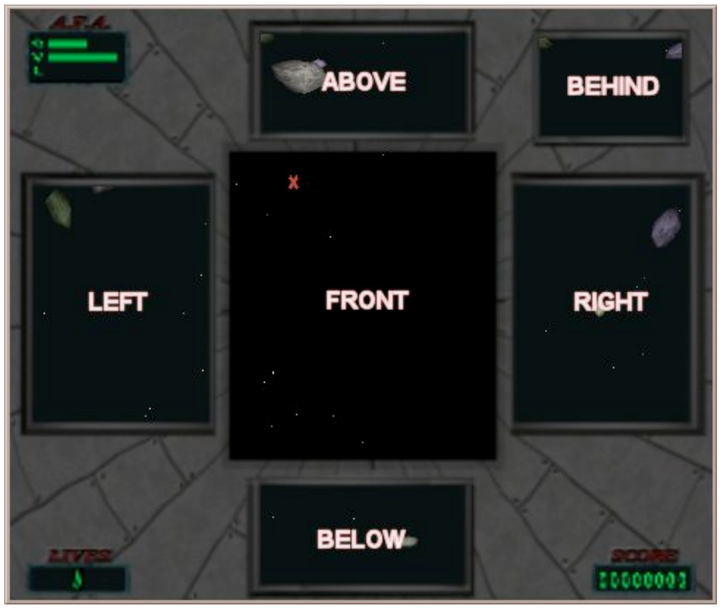
## Previous Work / Related Work

A lot of games have influenced the design of *Asteroid Blaster*. The biggest influence was *Asterax*, by Arvandor Software. When collected by a player's ship, *Asterax's* 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, 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.

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 must play through a 15-minute tutorial.

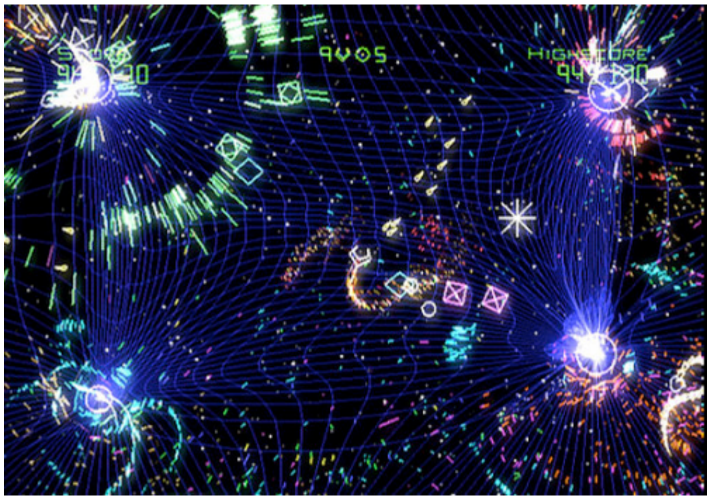


In part because of the confusing controls and view, *3D Asteroids* loses the fast-paced excitement of the original. Additionally, *3D Asteroids* retains the infinitely looping world of the original. A player may travel in a single direction indefinitely without being stopped by any walls. This works 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.

Keel was in the wrong place.
Keel: Triple redundancy in everything except common sense.

*Quake III Arena* was released by id Software in 1999. It features a minimalist deathmatch experience. There are ten weapons and two items. It has 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*, both game share similar objectives. Also, *Quake III Arena* is very straightforward and easy to learn. This makes it a good model for several aspects of *Asteroid Blaster*.

Bizarre Creations's *Geometry Wars* has a unique visual style comprised primarily of lines and particle effects. This style combines elements of retro games like *Asteroids* with more modern graphical techniques, which is our goal with *Asteroid Blaster*.

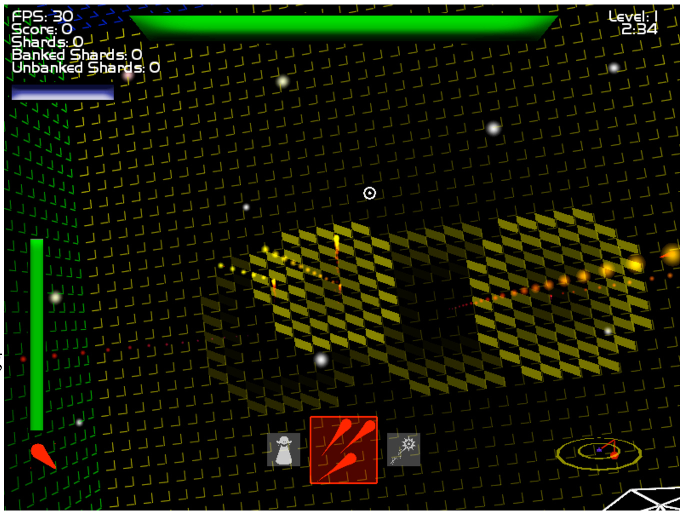


## Algorithms Overview

- 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)
- Network (Ryuho, Sterling)
- Text/Font (Ryuho, Taylor)
- SoundEffect / Music code (Sterling, Ryuho)
- 3D Audio (Sterling)
- Sound Design / Music Compositon (Sterling)
- Menu System (Ryuho, Sterling, Chris)
- Weapon Upgrade (Ryuho, Sterling)
- Weapon Price / Balance (Ryuho, Sterling)
- Levels (Ryuho)
- Spectator Mode Camera (Ryuho)
- Input System (Mike, Ryuho)
- Particle System (Sterling, Ryuho, Taylor, Chris)
- Collision Detection (Sterling)
- Asteroid Explosions (Chris, Sterling)
- Bounding space (Sterling)
- Ship modeling/animation (Jake)
- Weapon and effect modeling/animation (Jake)
- Barrel Roll (Jake, Sterling)

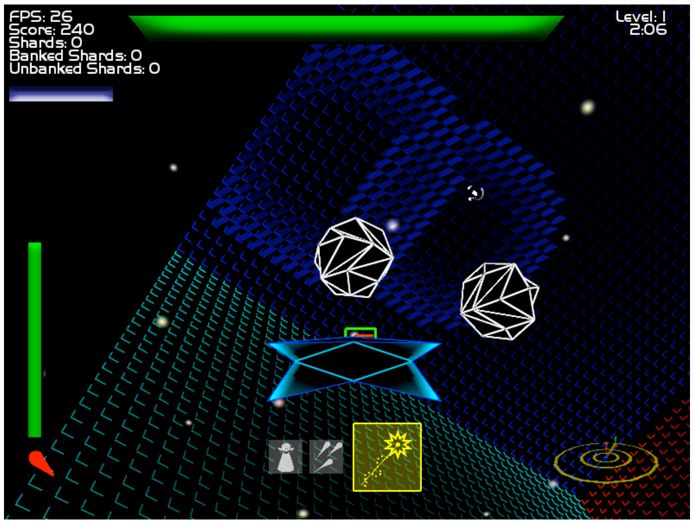
## Algorithm Details

### Cube Walls



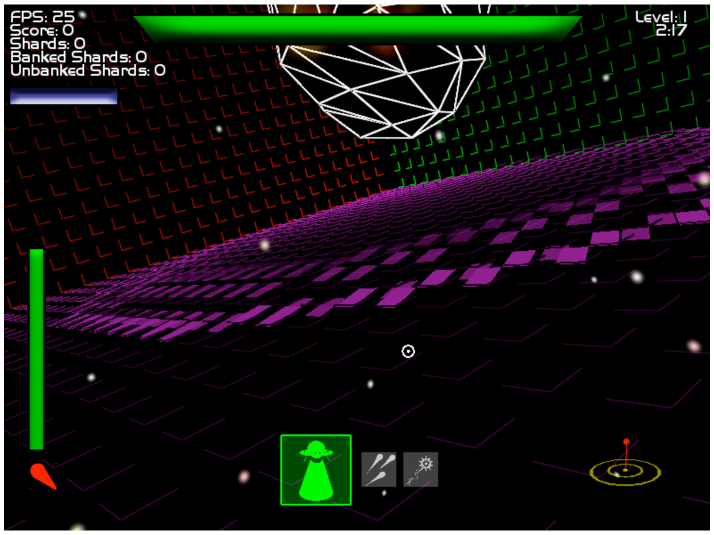
One of the most obvious and visually appealing graphical effects is the ripple effect on the walls of the cube. This is triggered whenever an object bounces off the sides of the cube. Large asteroids cause a large ripple, while smaller objects cause smaller ripples. The effect is achieved by drawing parallelograms the same color as the wall in a circle originating where the object hit.

Each cell of the wall is its own object, called a GlowSquare. When nothing has hit the wall, none of the GlowSquares draw. As soon as an impact is detected, the object that impacted is reflected and the ripple effect is triggered. The wall does the actual collision detection, so when something hits it, the wall finds the closest GlowSquare to the object and uses that as the center of the ripple.

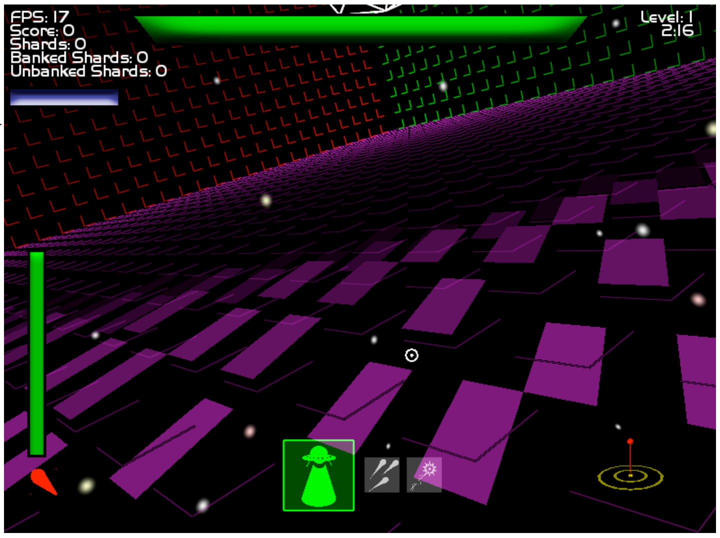


Each GlowSquare keeps a priority queue of times sorted from earliest to latest. These times represent when the GlowSquare's animation should begin. When the wall detects a collision, it first finds all affected GlowSquares. The wall adds the current time to the priority queue of the center GlowSquare. GlowSquares that are further from the center have the current time delayed proportional to their distances from the center before the time is added to their queues.

When drawing, the GlowSquares use the latest time in the priority queue that has not yet passed. By using a constant animation length, the GlowSquare can calculate how much of the animation has passed from 0, meaning the animation just started, to 1, meaning the animation is just finishing. This value is called amountComplete. The GlowSquare uses this value to set its position and opacity. When amountComplete is 0, the GlowSquare is half transparent (alpha 0.5). This fades linearly to fully transparent (alpha 0), at which point the GlowSquare is no longer drawn.



The position of the GlowSquare is also modified during the ripple effect. The GlowSquare is translated along its normal by an amount proportional to the sin(amountComplete * PI). As amountComplete goes from 0 to 1, the GlowSquare drops behind the wall, moves back in front of the wall, and then fades out as it settles back to its original position.



This ripple effect has been the single largest source of positive feedback when showing people *Asteroid Blaster*. Unfortunately it also is one of the most computationally intensive parts of the game. Several profiles have show it using over 40% of CPU time. However, the effect is very pretty, so it’s CPU time well-spent.

In its first iteration, the GlowSquares only stored a single time, as opposed to a priority queue of times. This worked fine when only a single ripple happened at a time, but when two ripples would overlap, the second would cancel the first in an ugly way. Adding the priority queue ensures that one ripple does not overwrite the times from another.

## Collision Detection

Many of the commonly used spacial data structures, such as oct-trees and binary space partition trees, seem to be well-suited for large, mostly static scenes, like terrain. *Asteroid Blaster* presents a less traditional challenge of handling collisions in a world with at most one or two hundred objects, all of which are moving (there are only six walls, all of which are axis-aligned, so this algorithm does not include collision detection for walls). Most of the time, however, the objects are small and the world is sparsely populated.

To handle this sparse population, all collideable objects are stored in a single list. Each object must specify its minimum and maximum values in the X dimension. Every frame, after object positions are updated but before collisions are tested, the list of objects is sorted by the objects' minimum X values. When testing for collisions, the program starts at the element with the smallest mimimum X value. It checks if the next item in the list has a minimum X value less than the current item's maximum X value, signifying a potential collision. If so, that item will be tested again more thoroughly, as in a sphere-ray intersection or a sphere-cone intersection. The next item is then checked, and so on, until one item's minimum X is greater than the first item's maximum X, indicating no more collisions are possible with the first item. Next, the second item is checked. Only items further in the list must be checked, since previous items will already have been checked.

Each step of this algorithm can be thought of as taking a slice of the world along the Y-Z plane containing just enough space to hold the object to be checked. Only other objects in that slice must be checked. This has a worst case scenario of being O(n^2). However, in this application, it is exceedingly rare for every object to occupy the same slice of the world. If it did happen, it would only be for a small number of frames, since every object is moving. Furthermore, having all objects exist in the same slice is only likely to happen when there are a small number of objects, in which case the test would not cause a performance hit anyway.

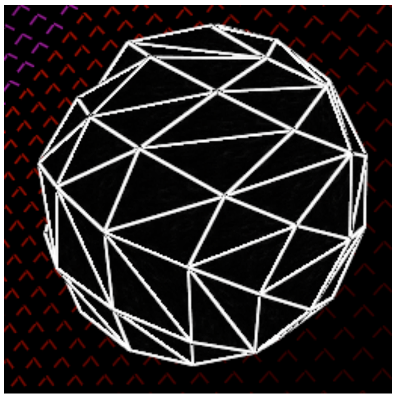
Each collideable object has a collision object that defines the shape and parameters of the collision area. For example, each asteroid has a CollisionSphere centered at the asteroid’s center with a radius equal to the asteroid’s radius. This allows various objects with different interfaces to have a consistent way of checking for collisions with each other. Each collision object has a method that accepts each other type of collision object and returns a bool indicating whether or not the objects collide.

Once collisions are detected, they are handled separately. The first version of collision handling used a virtual method on each collideable object called handleCollision that accepts a pointer to the other object. This worked fine during the initial stages of development, but quickly became unwieldy. The effects of a collision could come from either of the collided objects, and the order of the handleCollision calls could not be guaranteed. This was replaced with the current collision handling system. There are now template functions that handle each type of collision outside of either of the collided objects. For example, when an asteroid and a ship collide, the function handleCollision<Ship, Asteroid>(a, b) is called. Centralizing all collision handling had made the code far more maintainable. If some new effect is to be applied whenever a ship gets hit with a shot, seven functions -- one for each type of shot -- must be changed in the same file. In the past, the effect could be applied in any of 10 places (the ship, each of the shots, and some parent classes of shots) -- a maintainability nightmare!

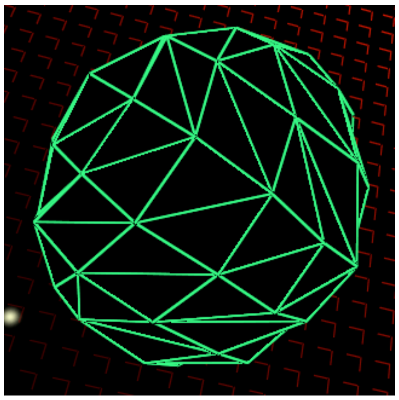
## Asteroid Explosions

When the ship explodes in the original *Asteroids*, it breaks from a triangle into a few lines that drift apart for a few seconds. That effect inspired the explosion effect of the asteroids in *Asteroid Blaster*.

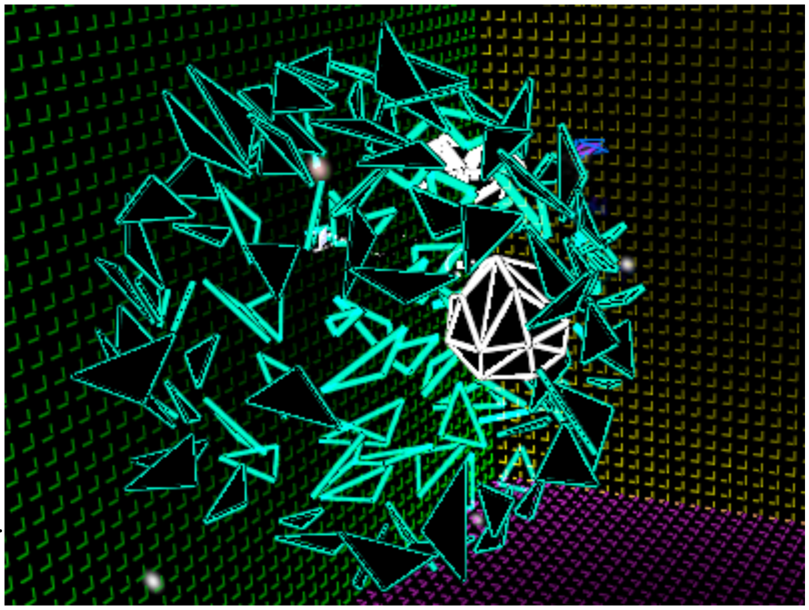
When an asteroid spawns, its lines are white. This indicates that its health is at maximum. As it is hit more, the colors change toward a fully saturated color, indicating lower health.



When the asteroid's health reaches zero, it explodes. If the asteroid is big enough (radius > 2 world units), it leaves two smaller asteroids in its place. When asteroids explode, each of the triangles turns into its own object in 3D with its own trajectory and rotation. This gives the appearance of the shattering like glass.



Each asteroid has a randomly generated triangle mesh. When the asteroid explodes, each vertex on each face is converted to world space from object space. Then the center point is calculated for each face. The coordinates for the vertices are then made relative to the centerpoint for the face. A random axis of rotation is generated, with the initial angle set to 0. Faces are given an initial velocity based on the velocity and rotation of the asteroid. If an asteroid is spinning fast, faces near the equator will fly farther than those near the poles. Faces from asteroids are not collideable objects, although they do bounce off the walls, generating ripples. Since they don't collide with anything, and since asteroids are randomly generated on the client side in network games, faces never need to be sent over the network.



## Results

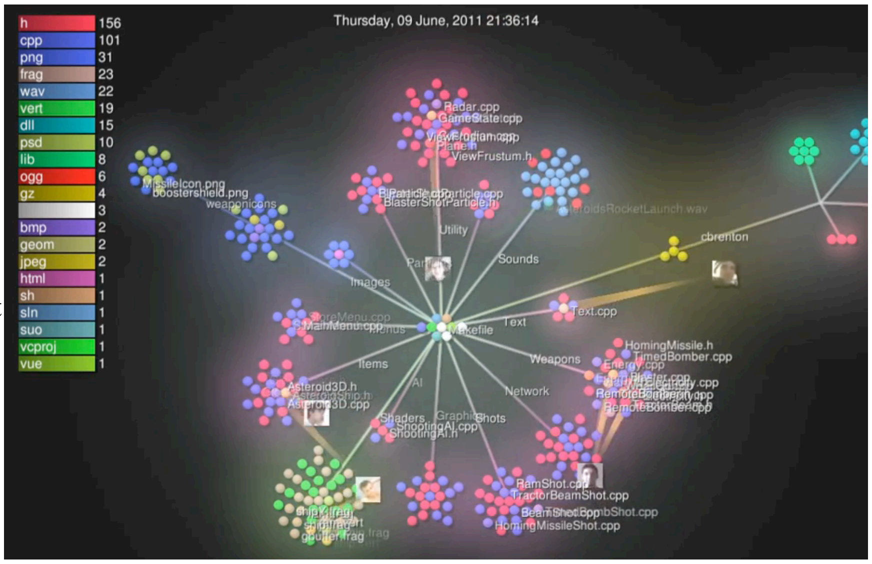
*Asteroid Blaster* ended up being ranked first in playability by other students in the class. The game's graphics have an arcade feel that fits well with the style of gameplay. There are enough levels that a player could reasonably play for over an hour, and the game would still offer an increasing challenge 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. 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 modified 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 initially tested *Asteroid Blaster*, they would continue to respawn infinitely three seconds after dying. Players complained about feeling like there was no point to playing, since the game did not end, even when 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 made the game significantly more satisfying.

We attempted to add networking in the last several weeks of the project. Unfortunately, this ended up being a very difficult task. At its best, players could use two of the weapons, and the tractor beam caused a segmentation fault. Before turning in the project, we decided to deactivate the networking code, since it was unstable and incomplete.

We also attempted to implement a deferred shading engine. We made a lot of progress, but in the end, the we were unable to fully implement light geometry. This was partially due to a paucity of available resources online and partially due to time constraints. At this time, deferred shading is an emerging technology that has only recently become popular in the game industry.



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. A link to this video will be available on the project's web page.

As a team-based school project, *Asteroid Blaster* was somewhat unique. This project allowed our team to decide what challenges to accept and to what degree: we set our own goals and decided how ambitious to be. This gave us an enormous amount of freedom to experiment with the graphics and gameplay.

## Future Work

Like many software projects, *Asteroid Blaster* could always benefit from more features. Unfortunately, time is limited, especially in school. We could add a lot of usability improvements, like remappable player input and autodetection of screen resolution. However, this project has been an exercise in managing priorities. Our team has done an enormous amount of work in the time we were given, but we still were not able to finish every feature we wanted.

More weapons would extend the time that the player continues to encounter new game material. There were a few weapons that we started working on, but never finished. Others, we started thinking about, but never worked on. For example, we started a weapon called the lawnmower. This ended up looking like lobster claws coming out of the ship, which unfortunately did not make any sense, so we cut it from the game.

Another big feature that we would have liked to implement is multiple game modes, especially for multiplayer. At best, multiplayer only supports free-for-all deathmatch. It would not have been unreasonable to support cooperative play with some simple modifications. We also discussed adding other gametypes, such as one we called “Shard Soccer,” where the goal is to use weapons to push a shard into the opponent’s goal.

Adding droppable powerups would make gameplay more interesting. Asteroids could, for example, have a random chance of dropping a health powerup or a temporary speed boost. These kinds of things can add variety to the game. It would also be fun to be able to grab these with the tractor beam, since the tractor beam is currently only used for grabbing shards.

Another thing we would like to add is multiple types of asteroids. It would be interesting to have some sort of indestructable asteroid that could be pushed into other players by shots.

Finally, there are plenty of ship upgrades and items that could be added to the store. The more store items available, the more choices players have when constructing their ship. Already, there are a few interesting things to buy. But with more upgrades, the player has more control over the balance of engines and shields.

## Conclusion

Our group had a friendly rivalry with another group in the class, *Project Lume*. We teased each other in class, but we shared a mutual respect for each other's work. This rivalry motivated us to put our best effort into the game, and that effort shows in both *Asteroid Blaster* and *Project Lume.*

We spent a lot of time working on networking and deferred shading, which we never finished. If we had made them work, they would have been huge wins, but we didn't end up having enough time. If I had been able to realize that sooner, I could have reassigned the people working on those features and gotten more done in other areas. I'm glad this was a school project, since canceling a feature like networking so late into the development of a production game is not feasible. If we had given up early, we would have always wondered if we could have pulled it off.

Over the last quarter, I learned that with time and effort, I really can make something awesome. I realized that the key to good code is a mixture of self-confidence, persistence, and honest self-evaluation. I learned how far I can push people I'm managing and what I can expect from them in a given amount of time. I learned not to wait until the end of a project to add features as big as networking or deferred shading, and I learned not to expect a single programmer to be able to do it all. I reinforced my knowledge of the value of pair programming. I learned how difficult it can be to integrate third-party code, but that usually, it is still less work than writing the code yourself. I learned that sometimes, it's better to cut your losses than pour a lot of resources into unlikely features.

Our game turned out great. I got to have my vision realized, which was intensely satisfying. It was a lot of work, but the whole experience was a blast.

## References / Bibliography

This includes several of libraries that we used in the creation of *Asteroid Blaster*, as well as games that inspired us.

### Libraries

SDL http://www.libsdl.org/

SDL Mixer http://www.libsdl.org/projects/SDL_mixer/

SDL Image http://www.libsdl.org/projects/SDL_image/

Boost http://www.boost.org/

OpenGL http://www.opengl.org/

GLEW http://glew.sourceforge.net/

### Games

Quake III Arena http://www.quake.com/games/quake/quake3-arena/

Asterax http://musegames.com/news/games-we-loved/asterax/

Geometry Wars http://store.steampowered.com/app/8400

3D Asteroids http://www.grassgames.com/asteroids/