# Software Architecture Document
For
# Super Smashteroids

Version 1.0
Written by: Andrew Penhale, John Lian,
Nicolas Martin, Charlie Marokhovsky,
Henry Lu and Omar Abdelkader

# Table of Contents

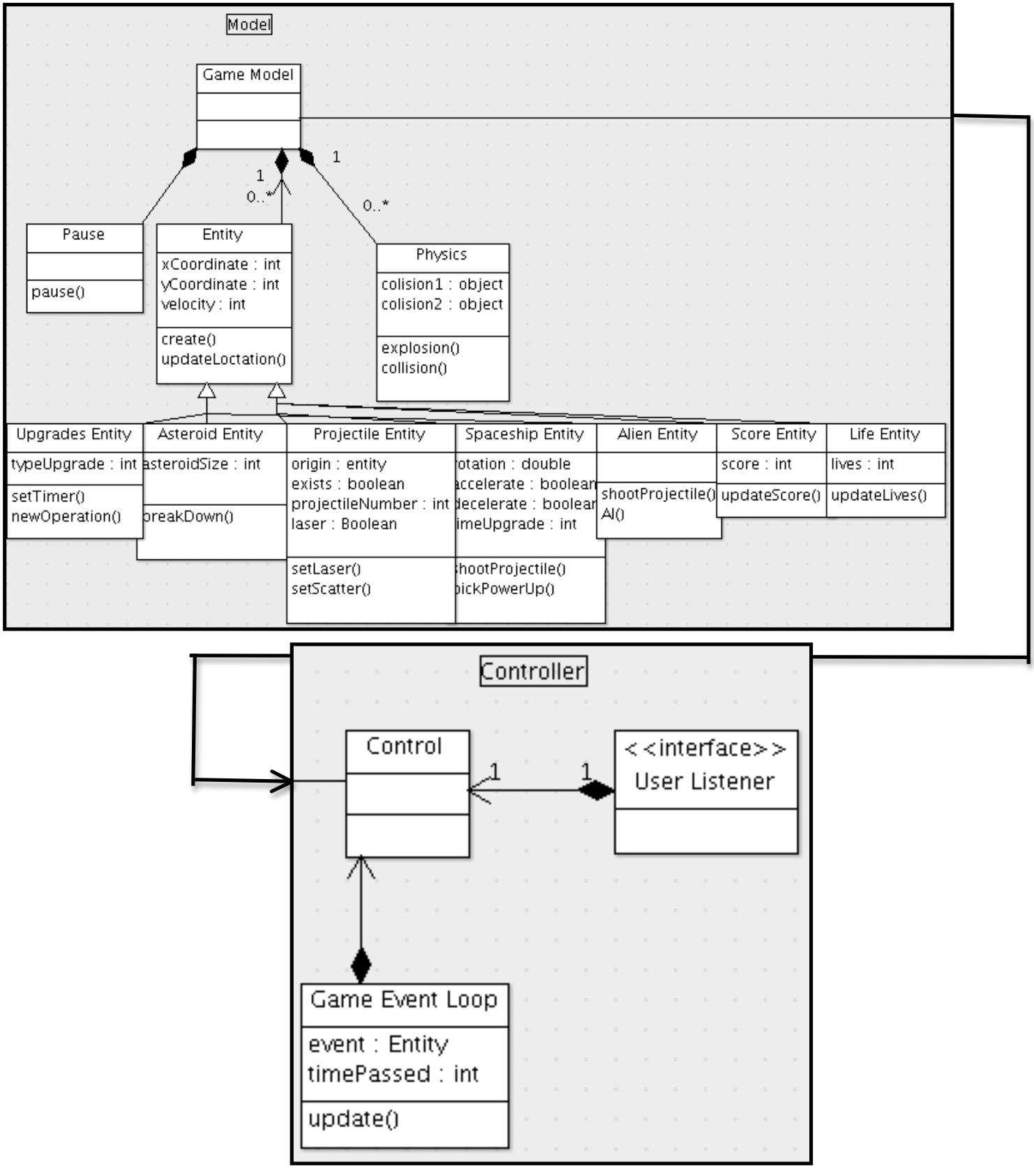
# System Overview

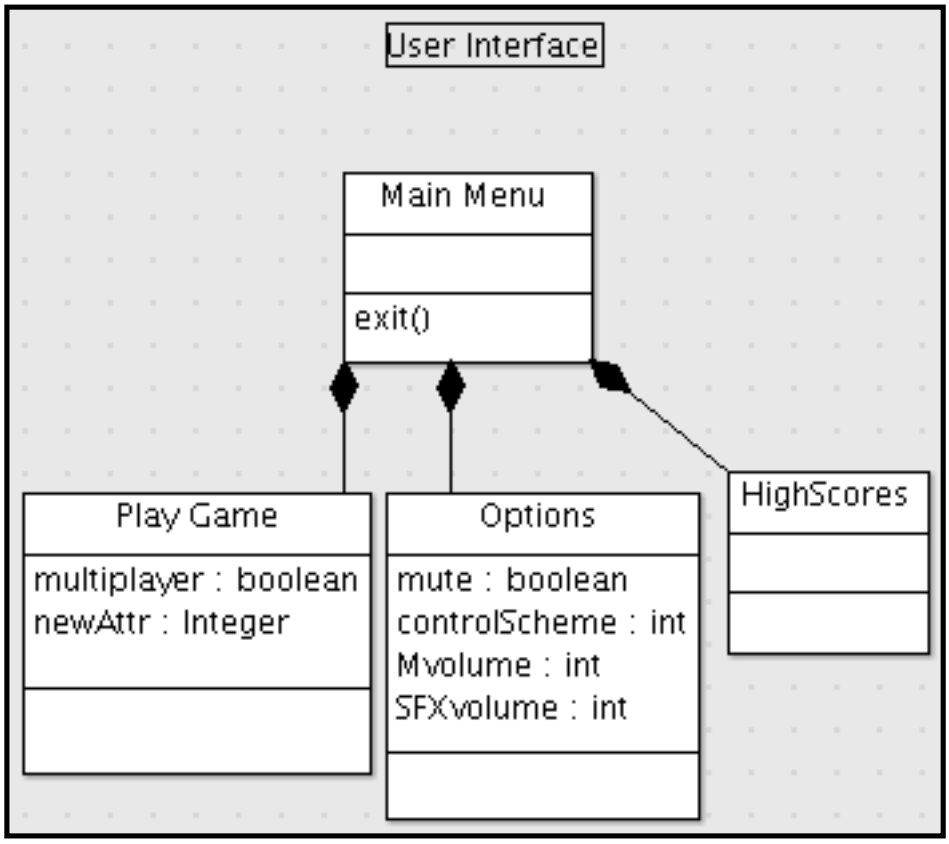
Through research, we have decided the best architecture for our Super Smashteroids game would be the Model-View-Controller (MVC) design. The MVC architecture has been proven to be effective and is applicable to our project. MVC divides the code into 3 main categories: Controller, Model, and View. This design implementation cleanly separates the majority of game logic and data from the graphical view and user input. Specifically, the code that falls under the Model category, such as ship, asteroids, aliens, will perform actions such as maintaining attributes (velocity, position, etc.) and generating new objects. This data is then picked up by the View; classes that will be responsible for rendering the graphical view. The classes that fall under the Controller category will listen to the keyboard input from the user and call for appropriate response from the Model and thus manipulating it. We believe that this design brings good versatility in both game logic and visualization because objects could be added and removed without alteration to a large portion of the system. An additional User Interface subsystem would also be included to handle, as the name suggests, the entire user interface for the software system. The UI is mostly decoupled from the MVC component of the game, because it would have nothing to do with the *gaming* part of the system.

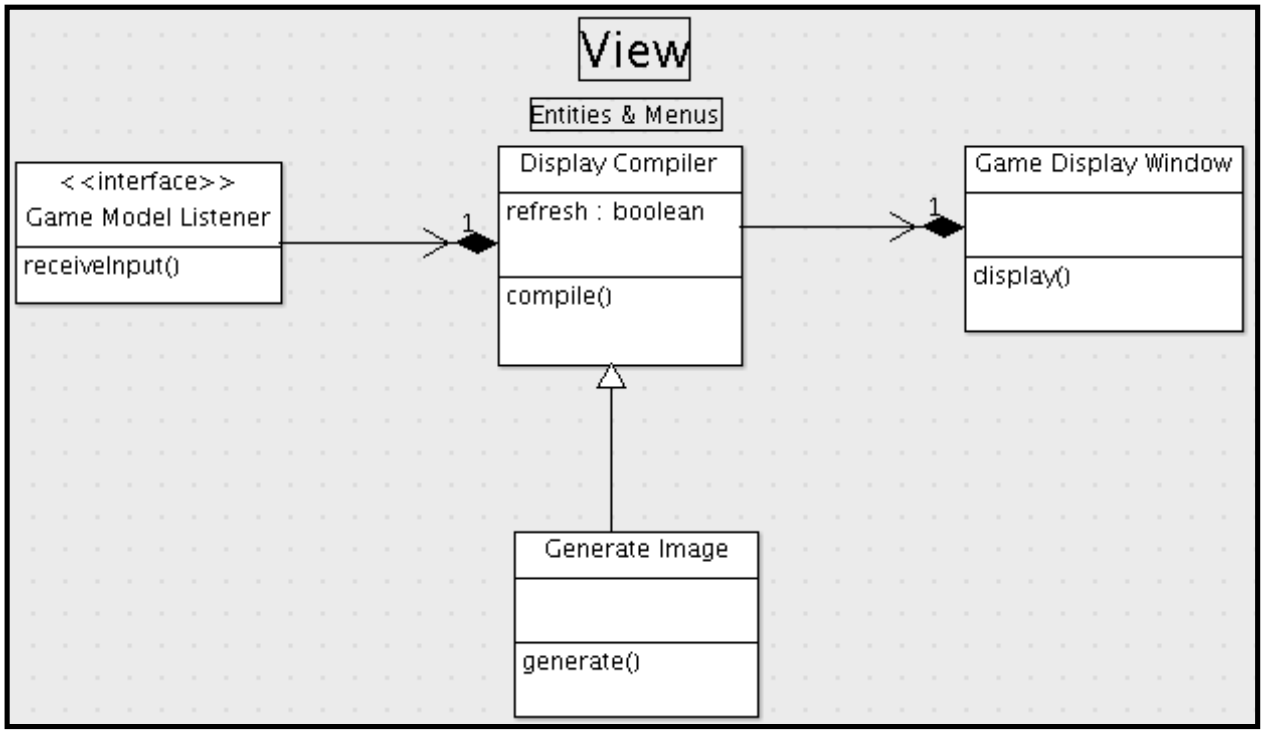
As discussed above, the Game subsystem will be implemented by following the MVC design. In this subsystem, each entity will have it's own class. That is, there will be a ShipEntity class, AsteroidEntity class, ProjectileEntity class, AlienEntity class, and a PowerUpEntity class. In relation to the MVC design, these classes fall under the Model category. There will also be a class used to initiate the game thread; this falls under the Controller category of the MVC design. Finally, there will be a class, or possibly several classes, that will be responsible for repainting the canvas and displaying all the graphics that exist. They are also responsible for updating the game window (or canvas) when changes are made to the positioning of the graphic objects or when the state of these objects is changed. Again, relating this to the MVC design, these classes are considered part of the View category.

The User Interface subsystem is fairly straightforward. There will be a MainMenu class, which will display the very first menu the user comes across when the program is run. Within this class, there will be an exit command or method that will respond to the user's selection of the "Exit" option in the menu and will terminate the program. Additionally, this menu will have 3 submenus all represented by classes. Therefore, there will be a PlayGame class, an Options class, and a HighScores class. Each of these classes will be responsible for displaying a specific submenu if selected by the user.

# Views


Model
Game Model
1
1
0..*
0..*
Pause
pause()
Entity
xCoordinate : int
yCoordinate : int
velocity : int
create()
updateLoctation()
Physics
colision1 : object
colision2 : object
explosion()
collision()
Upgrades Entity
typeUpgrade : int
setTimer()
newOperation()
Asteroid Entity
asteroidSize : int
breakDown()
Projectile Entity
origin : entity
exists : boolean
projectileNumber : int
laser : Boolean
setLaser()
setScatter()
Spaceship Entity
rotation : double
accelerate : boolean
decelerate : boolean
timeUpgrade : int
shootProjectile()
pickPowerUp()
Alien Entity
shootProjectile()
AI()
Score Entity
score : int
updateScore()
Life Entity
lives : int
updateLives()
Controller
Control
1
1
<<interface>>
User Listener
Game Event Loop
event : Entity
timePassed : int
update()


User Interface
Main Menu
exit()
Play Game
multiplayer : boolean
newAttr : Integer
Options
mute : boolean
controlScheme : int
Mvolume : int
SFXvolume : int
HighScores


View
Entities & Menus
<<interface>>
Game Model Listener
receiveInput()
1
Display Compiler
refresh : boolean
compile()
1
Game Display Window
display()
Generate Image
generate()

# Software Subsystems

The system will be broken down into two main subsystems; namely, the Game subsystem, and the User Interface subsystem. As the name implies, the Game subsystem will be the portion of the system that deals with the actual gameplay of the Asteroids game. It will be implemented by following the MVC model. Therefore, this subsystem will be further divided into three *sub-subsystems* that are decomposed in the same manner as normal subsystem decomposition. It will be organized in such a way as to satisfy the MVC pattern. The User Interface subsystem will be the part of the system that consists of the Main Menu and all the options accessible from the Main Menu.

## The Game Subsystem

### Model

The Model is the central hub of information in the software system: it receives controlling instructions, keeps track of data, and provides information for interpretation. Within Model, most of the functionality relies on the Game Model, where all attributes of each object are kept.

#### Main Roles

This Model is essentially the abstract representation of the game. Ideally, it could be reproduced in many different types of graphical representation depending on the interpretation of the information from the Model. It is responsible for

- Housing a list of all entities active in the game;
- Creating entities;
- Destroying, or removing, entities;
- Defining and realizing the coordinate system; and
- Managing other properties of entities.

This Model will also provide references to its listeners (in Views) and notify them whenever needed. Specifically, it flags datums (such as positions, velocity, number of lives the player has, and the current score) and lets the listeners know that the flagged items are updated and needs to be re-rendered. Additionally, the Model listens to the Controller for any user input or event loop updates.

### Entities

An Entity is any *moving* object within the gaming system. An entity would be comprised of its attributes that would describe the object. Each entity

- Has a position2D variable that represents the location of the top left corner of the object in the coordinate system;
- Defines and stores a boundary2D variable that represents the boundary box for collision detection;
- Computes and stores its velocity variable;
- Computes and stores its angleOfRotation variable;
- Is capable of detecting collision with other entities; and
- Stores other attributes pertaining to each entity.

Game Model keeps track of all the entities.

## View

The View obtains information that Model provides and translates it to create a visual representation of the model. Particularly, the Model would pass on data such as position, angle of rotation, and boundary of each entity, and the view would have to rotate, scale, and position each entity's bitmapped sprite image appropriately to create the rendering.

### Listener Interface

The View actively listens to the Model by implementing the GameModelListener interface. When an attribute changes within Model, it is flagged by a flagChanged() function, and calls for a modelChanged() function for View through the interface. This function then informs View to render the updated model using the data.

### Graphical Rendering

When the Model changes and it is asked to redraw the window, View scans all of the read-only data in order to create the visual representation. Since the entire window have to be redrawn every frame, View redraws the flagged items with priority and the rest copied from the last frame. This is done to reduce unnecessary information relay between the modules. Once the drawing has been determined, the View draws the background and calls for the drawEntity() function for each entity. Here

- The position of the entity is mapped onto the DrawBoard;
- The rotation and size is determined in relation to the DrawBoard;
- The shape of the polygon is determined; and
- The polygon is drawn with standard JavaSwing functions appropriately.

It is noted that View has no awareness of any in-game logic.

## Controller

The Controller listens to the user's keyboard input, and provides instructions for the Model to change accordingly.

### Input Listener

The Controller listens to the keyboard for any input. If the user presses the defined *fire* button on the keyboard, the listener would receive a keyboard listener event and would then send instructions to the Model, signaling the need to change.

### Game Event Loop

The Controller also maintains a game event loop that tells the Model to update at a regular interval. Actions and instructions are sent through this event loop. If a keyboard event is received, the Controller includes this information along with how much time has passed since the last iteration of the loop to send to the Model. The Model uses that information to notify each entity to calculate and update their attributes with respect to the amount of time passed. Additionally, the Model is the system that controls when new entities are generated (such as the ProjectileEntity) or not (too little time has passed since the last projectile fired).

### Game Window

The game window is the last stop for information. The graphical data received. Majority of the functions of GUI reside in the GameWindow class. It receives graphical data from View and superimposes any additional information - such as high score, number of lives left, and menu overlay - over the game rendering. It is important to have GUI separate from View because GUI does not pertain to the game system and more of an additional layer for interaction.

# The User Interface Subsystem

### Menu

This subsystem is responsible for all components of the software that occurs when the user is not playing the game. It is responsible for

- The location of items on the menu screen
- Storing the High scores information
- The layout of the different screens (i.e. main menu screen vs. options menu screen)
- The selected control setup (i.e. WASD vs. Arrow Keys)
- The sound settings (both music and SFX)
- Navigation between screens (i.e. from main menu to options menu)

The UI subsystem will offer a menu interface where the user is able to change options, difficulty, start a game, and view high scores.

# Analysis

## Performance

Description: The program should take no more than 3 seconds to show the main menu screen. It should also take no more than 3 seconds for the game to start upon clicking either “Single Player” or “Two Player”. After gameplay has begun, the spaceship should respond to each input from the user almost immediately.
Implementation: The system will be coded in a way that minimizes menu interaction running time and player interaction running time while ensuring all the features are fully implemented. This quality will rely on the Game Event Loop consistently updating the display graphics( based on user input and game mechanics) without any sort of delay.

## Reliability

Description: This software should never lag or crash on its own accord. Any program errors should occur only as a result of the operating system on which it runs.
Implementation: We will perform extensive bug testing to remove all bugs from our game before it is released.

## Availability

Description: This software should be free to run on any computer with Java installed.
Implementation: We will be writing our software in Java; therefore it should be guaranteed to run on any system that has the most up-to-date version of Java installed.

## Securities & Privacy

Description: This software should not collect nor distribute any data or information from the user or from the computer on which it is installed.
Implementation: Our software code will not include any directions for the game to collect data from the user. We will not be able to distribute information either, since it will not be connected in any way to the internet.

## Maintainability

Description: This software should keep an up to date and accurate version history. Any bug fixes or new releases should be implemented on entirely separate software releases as opposed to patch updates.
Implementation: The software will be re-released as a whole system whenever updates are made to the system.

## Safety

Description: This software should warn the user of the risks associated with epileptic players playing video games. It should also advise the player to take frequent breaks, even though they might find it difficult to stop.

Implementation: When the game starts up, a splash screen will appear displaying the required safety instructions. It will fade away after 5 seconds.

## Training-Related Requirements

Description: This software should not require any training to be able to play the game. The instructions should be displayed on the main screen and the features of the game should be learned as the user plays through the game.
Implementation: The main menu screen will display the instructions and controls next to the menu buttons. This will allow the user to learn how to play the game before the game starts. Additionally, the difficulty levels begin at easy and increase as the game progresses so the user should also be able to learn while playing the game.

## Packaging Requirements

Description: The software should be packaged with proper documentation in some form of text file.
Implementation: A ReadMe text file will be written and made available for distribution alongside the software system.

## Features likely to change:

The performance of the system may change as we progress in the coding of our software.

- This is relevant to all systems in our software.

The system may be changed to display in full screen as opposed to a fixed-resolution window.

- The user interface subsystem and graphical rendering division will be the targets of any changes that have to be made for this feature to be implemented.

The system may be changed slightly to increase the number of users playing simultaneously.

- This will mostly affect the controller and model divisions of the Game subsystem to accommodate the increase in players and inputs. The system will also need to render two different user controlled ships.

The collision mechanism of the asteroids may also change. When an asteroid comes into contact with another asteroid, asteroids may be changed to ricochet off each other instead of passing through one another.

- Changes here would mostly affect the Game (model) subsystem to perform the necessary calculations and store information.

Power-Ups and bombs may be added to the game, allowing the player to upgrade their ship.

- Adding these will affect the Game subsystem. under the controller (adding user inputs), the view (graphics) and the model (to store information about power up entities) divisions.

The in-game pause menu may be altered to add a save game function.

- The Game and UI subsystems would be affected to implement this.

The alien flying saucer types may be changed to allow for several different types, creating more varied gameplay for the user.

- Subsystems needing to be altered would be the Game subsystem.

Background music may be added to the game.

- The Game subsystem would handle this addition.

A scoring system is likely to be added to our game, for shooting aliens and asteroids and grabbing power-ups.

- Would affect the Game subsystem (model and game windows divisions).

A difficulty setting will likely be implemented into the game

- This would fall under the domain of the Game and UI subsystem. UI would take input from user, but Game would implement the required difficulty in game.

# Design Rationale

## User Controls:

- The user will only be able to use the keyboard to navigate the menu. Because the game itself only requires the use of the keyboard, navigating the menu via keyboard keeps the controls consistent throughout the program.

## Gameplay:

- Alien AI will not travel beyond the edge of the screen and thus will not wrap around. This decision was made in order to simplify the game for the user, making the aliens easier to track.
- Asteroids and aliens passover each other rather than ricocheting off each other. Due to the inability to judge the implementation difficulty of making asteroids and aliens ricochet off one another, the simpler implementation was chosen. This is a design aspect that we wish to change once we understand the implementation.
- In the game design, the way difficulty will affect gameplay is as follows:
    - On Easy mode, there is a set number of large asteroids that can appear on the screen at once (i.e. 5 large asteroids on screen at a time). At regular intervals of time, the speed of the asteroids will increase. The game continues until the user has no lives left.

- As difficulty increases (Normal or Hard), the number of large asteroids that can appear on the screen at once is increased compared to the Easy difficulty (i.e 6 for Normal and 9 for Hard at any given time). The initial speed of the asteroids is also greater than the speed in the lower difficulty modes. As with all difficulty modes, at regular intervals of time, the speed of asteroids will increase. The game continues until the user has no lives left.

*Note: Asteroid numbers used above are for clarification purposes and do not reflect how the game will actually be implemented.

# Workload Breakdown

Work will be divided into three main groups:

- 2 people for UI subsystem and testing
- 2 people for collision physics
- 2 people for View and Controller parts of the Game subsystem

Our work division will be flexible, and people may be moved around to better fit the current situation. Once something is completed, that person will move around to help another team with their section. All teams will be constantly sharing information and collaborating on ideas to ensure that the code remains uniform.