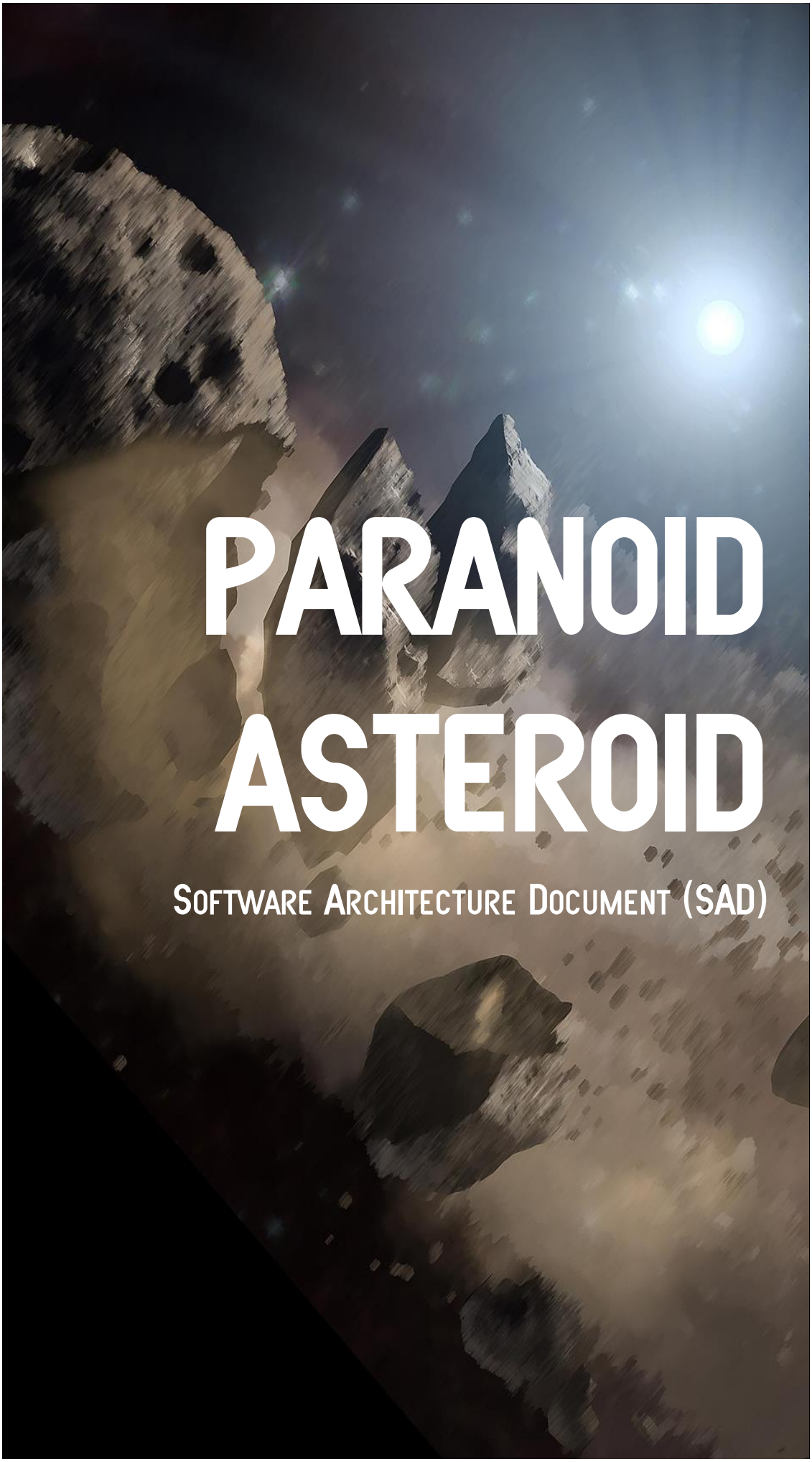

PARANOID ASTEROID

Software Architecture Document (SAD)

ECSE 321
Introduction to Software Engineering

Group 8:
Alex Bourdon
Alexander Coco
Payom Meshgin
Daniel Ranga
Jad Sayegh
Yi Qing Xiao

Client
Prof. Haibo Zeng

Revision No.: 1

Date: Friday, March 1, 2013

Table of Contents

INTRODUCTION

1.1 PURPOSE

This document is the Software Architecture Document for the design project of the first release of the Paranoid Asteroid video gaming system and provides an overview of the software architecture and its different modules, as well as the rationale behind its conception in relation to the system requirements.

1.2 SCOPE

The scope Software Architecture Document is a high level depiction of the Paranoid Asteroid game system which is designed to facilitate the game play of the arcade game Asteroids on modern household personal computers.

1.3 DEFINITIONS, ACRONYMS, ABBREVIATIONS

- *Asteroids*: original name of the Atari space game.
- playing field : area with wrap-around edges where game sprites move (star field, outer space)
- PC, spaceship: player animated sprite, controlled via the keyboard
- Asteroid: inanimate game sprite, travels on the field at constant velocity. varies in size (velocity depends on size; smaller, faster)
- Alien, Flying saucer: non-playable character which exists as two different versions, Large Alien and Small Alien, which travel on the field at constant velocity and fire projectiles which cause the PC to lose a life on impact. Exists in 2 formats, large and small (behaviour differs)
- Hyperspeed: player action. When activated (via keypress), spaceship disappears from current location and reappears at a random space in the playing field.
- SRS: Software Requirements Specification document, which specifies the functionalities of the system and its operation constraints.
- GUI: Graphical User Interface. Consists of all the menus and buttons with the user interacts.

2. System Overview

The game is separated into different parts: a windowing system, an in-game system, and a data handling system.

2.1 Window System

- The "Highscores" window will generate the score table and allow the user to check high scores (at end of a game or out of game) and add a high score (when a game has ended in high score). It appears on screen when the player clicks on the "High Score" button while in the "main menu" window is active or when the player ends a game session by losing all his lives.
- The "Options" window allows the user to choose certain preferences. Preferences include key setup, enabling and disabling sound and power ups and difficulty levels. The information will be passed down to the data handler, which will save the information and make it available to the rest of the game.
- The "Pause" window will appear only while playing when the player presses the pause key.
- The "Instructions" and "Credits" windows are made visible when the buttons with the same name are clicked from the "main menu" window, displaying text information about the game and the developers.
- The "Game" window is the window in which the in-game activity happens.

2.2 The Ingame

After the "menu" comes the "Game" class, which essentially serves as a game launcher and as a level watcher. Upon request of one of the three game types in the main "menu", the game will generate a gamefield appropriately. This class also keeps track of the in-game level the player is presently at, and sends command to refill the gamefield with the appropriate amount of entities every time the field is emptied by the player (level cleared by player).

The "gamefield" is the main class which controls the overall game session. It is the one generating or eliminating game entities as the game progress. It possesses a list of all the in-game entities, which is why it is also responsible for object collisions.

Basically, any in game related elements are gathered in this class, like scores, number of lives the player still has, the level the player is presently at, the power-ups the player presently possess, etc.

The "gamefield" is also the main class which feeds the "renderer" class with the position and type of each entities, allowing the graphic display to appropriately describe them graphically.

All entities inherit from the super class "Entity", which gives all of them variables for their position, orientation, and status (live/death). Each specific entity type will then have its own respective class to allow entities of that type to customize variables that they need. For example, only the player(s)'s ship need a keylistener object.

The entity types are: Space Ship, Large Aliens, Small Aliens, Large aliens, Bullets, Large Asteroid, Small Asteroid, Power Ups, Explosion.

Note that the Explosion class is simply to facilitate the rendering by providing to the rendering class the necessary information for shrapnel's movements.

The user can interact with the game through the keylistener, which is divided into two classes.

The keylistener is associated to all user related classes (non entity classes, except the space ship entity class). The function of a specific key pressed will be determined within the specific class in which the key is associated with. This class merely alerts that a key has been pressed, and tells to the class it belongs what key has been pressed. Thus, result of key press depends on which class is presently active.

2.3 Data Handling

Some external data must be stored after the game application has ended so that some of the previous state of the game may be preserved. Therefore, data such as key configurations, the high score table and save game data are stored in external files on the user's hard disk. However, this data is only written to a file when necessary, that is, in the case of exiting the game. While the game is running, a copy of the data is stored in memory. A number of data handling classes (HighscoreDataHandler, the SaveDataHandler and the OptionsDataHandler) are responsible for the global access and writing of the data, which are made available to all other classes in the application.

3. System Views

3.1 User View

The application window, or game window, is initially displayed upon the game being launched. The main menu window, generated above the game window

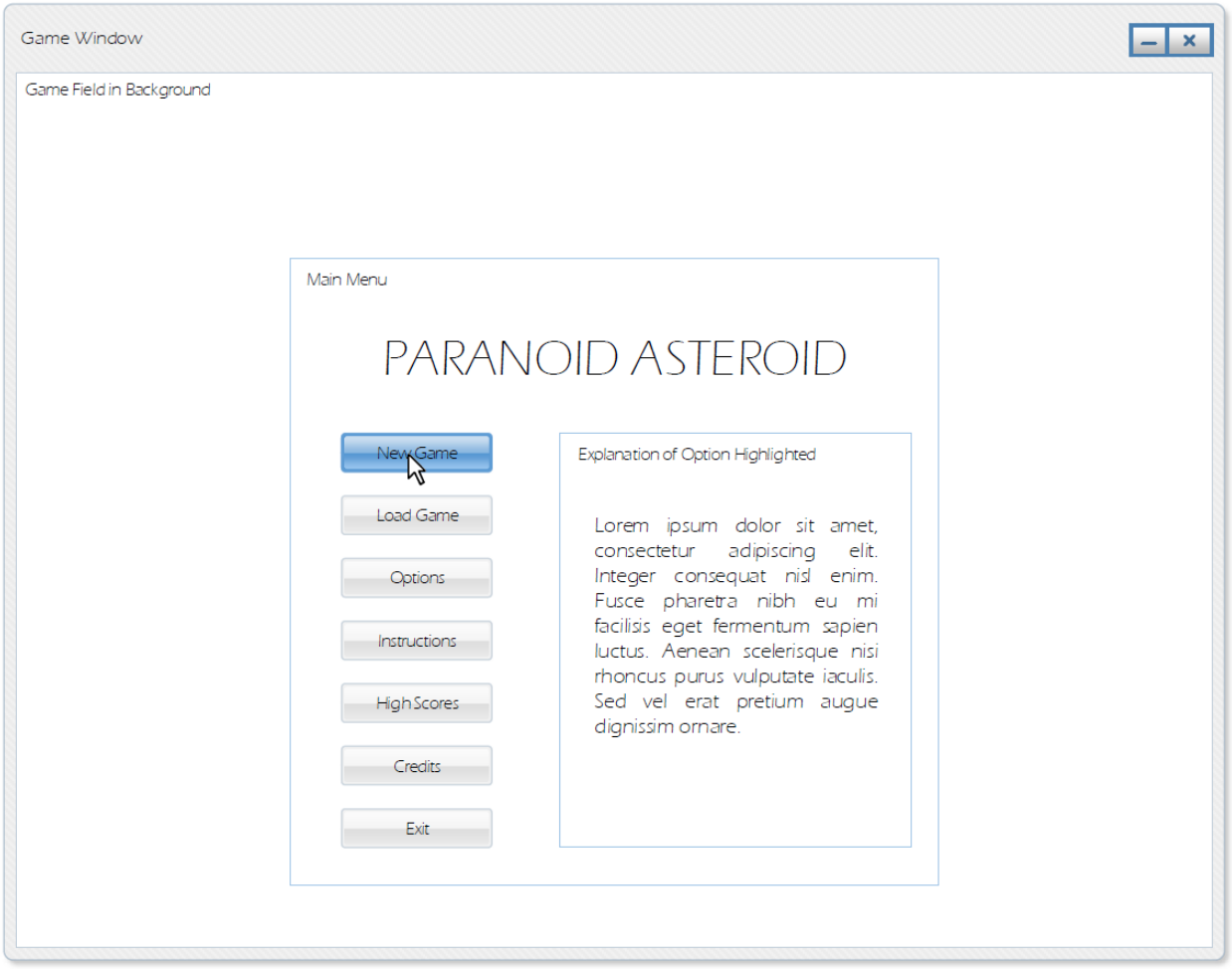

Figure 1: Main Menu graphical Representation (Tentative)

3.2 Window System and Data Handling

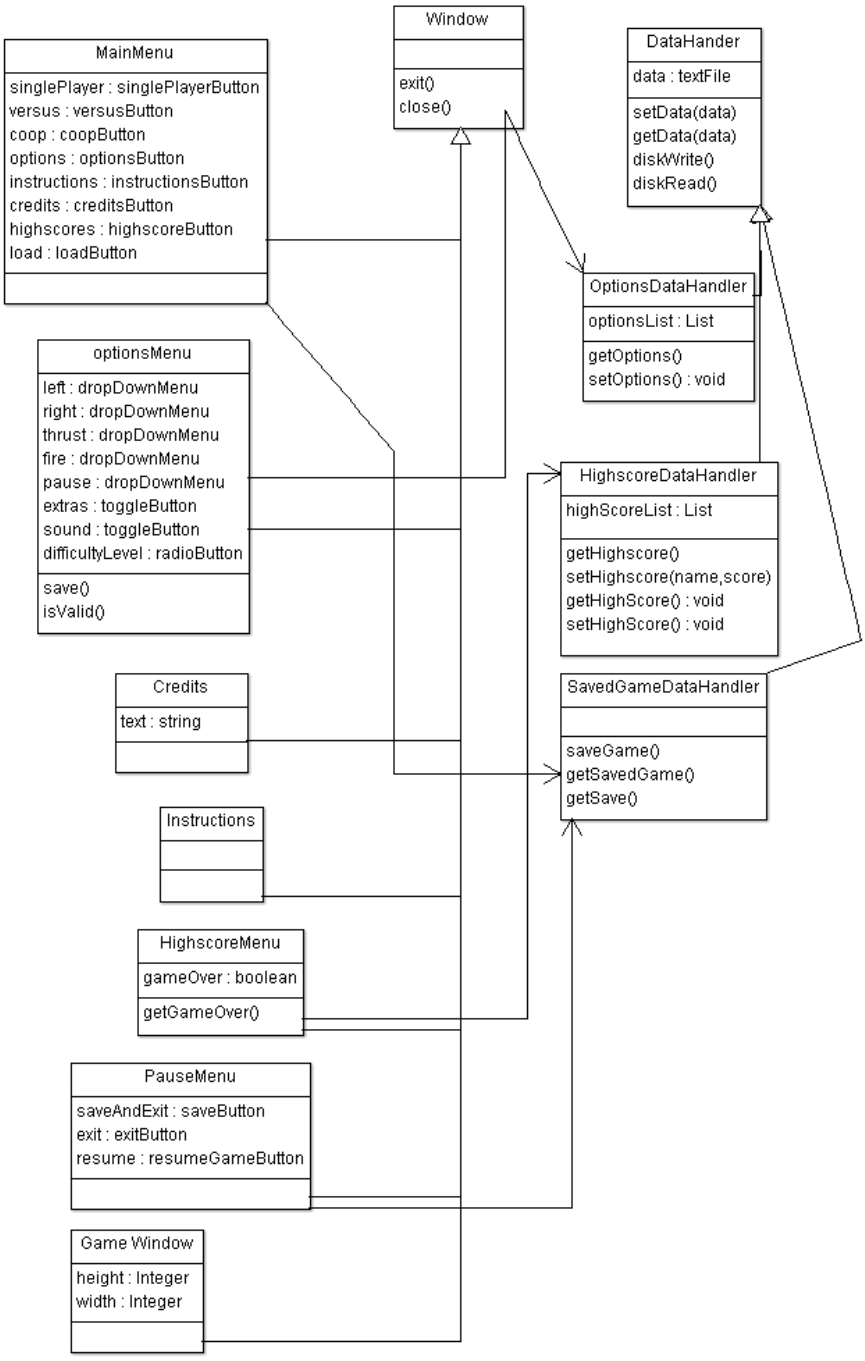

Figure 2: Menu System diagram

The above class diagram describes the system which is responsible for controlling the peripheral of the main game. Anything not directly related to the actual gameplay are described here, such as the main menu, the high score board, the saved files handling classes, etc.

3.3 INGAME

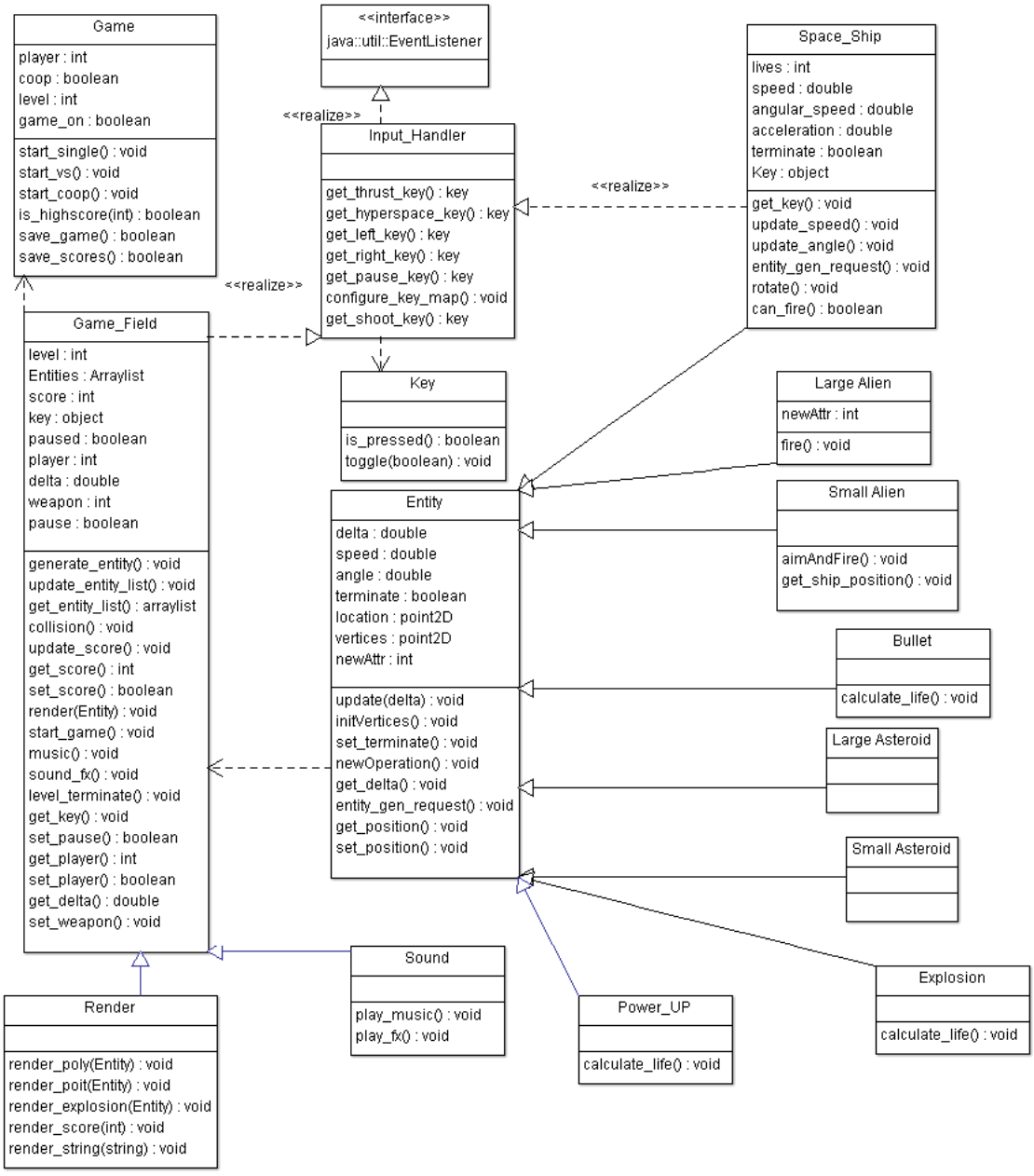

Figure 3: Game System Diagram

The above is a class diagram describing all the specific classes related to the actual gameplay of the game: in game objects such as asteroids, player's ship, as well as the game field in which the game proceeds.

NOTE: not displayed here are the dependencies of some of the classes of this diagram to the classes mentioned in the menu class diagrams. Most notably:

- Game class depends on the Game Window class
- Game class depends on the High Score class
- Game class depends on the Pause menu class
- Input_Handler class depends on the Data and Pause Menu classes

4. Software Units

4.1 GUI

4.1.1 Overview

The menu subsystem is composed of a collection of Menu subclasses which implement the JWindow interface and extend the Menu superclass. This interface imposes the implementation of close() and exit() methods, the behavior of which is explained below:

- exit() saves the state of the system then calls the close() method of all windows.
- close() removes the window from display, returning the user to the underlying window.

The different Menu subclasses which make up the menu subsystem are explored below:

4.1.2 Modules

4.1.2.1 MainMenu

As stated in the system overview, the MainMenu is the root of the Menu subsystem. From it the user navigates by clicking the various buttons displayed which either launch the game, or instantiates the appropriate Menu object. A table of the buttons available to the user, the classes which implement them, and the action executed on click can be seen below:

Button Label	Corresponding Button subclass	Action
Solo	SoloButton	Launches the game in single player mode
Versus	VersusButton	Launches the game in competitive play mode
Co-op	CoopButton	Launches the game in co-op mode
Options	OptionsButton	Instantiates an Options menu object
Highscores	HSButton	Instantiates an Highscores menu object
Load Game	LoadButton	Loads previously stored game and resumes it
Credits	CreditsButton	Instantiates an Credits menu object
Instructions	InstructionsButton	Instantiates an Instructions menu object

The MainMenu class does not have any methods apart from those implemented in the Menu superclass and does not require any inputs, nor does have any outputs aside from the instantiated classes. The game is launched when the Button object calls one of the Game.start(), depending on the chosen game mode then calls the close() method all Menu classes.

4.1.2.2 OptionsMenu

This class acts as an interface through which the user can configure the settings and the controls with which they want the game to be played. A combination of drop down menus and checkboxes display the current configuration and allow the user to choose valid configurations. The user can either cancel or save the changes they have made, with either decision closing the Options window. When the configuration is saved, Options sends the configuration to the DataHandler object.

The drop down menus provide the user with a list of valid keyboard keys that can be chosen to control the playable character's rotation (left or right), it's thrust and it's firing. Two separate menu sets exist to accommodate two players. Checkboxes allow the user to disable/enable music or extra features which were not present in the original Asteroids arcade game. Upon instantiation and initial display, the Options menu's drop down menus and checkboxes displays the last configuration saved. Finally, buttons allow the user to either save the options configuration, allowing the changes to be applied to the gameplay thereafter or to reset the options to their default configuration. This is achieved as explained below:

The method save() is called when the 'OK' button is clicked. The methods first calls the isValid() method to check that the configuration is valid and no key are assigned more than one function. The OptionsDataHandler.setOptions(int[]) method in the OptionsDataHandler class is then called, and the values contained in the dropDownMenu objects and the CheckBox objects are passed as an int array. OptionsMenu requires upon instantiation the last saved configuration set by the user as input. It's constructor therefore calls the OptionsDataHandler.getOptions() method in the OptionsDataHandler class.

The method OptionsDataHandler.setDefault() is called when the 'Reset' button is clicked, followed by the close() method of OptionsMenu.

Since the class outputs the configuration in an int array, the outputted array formatting must match the formatting expected by the setOptions() method.

4.1.2.3 InstructionsMenu

The InstructionsMenu class displays a collection of static text and images which constitute a gameplay tutorial for the user. It does not execute any behavior, nor does it have any inputs/outputs.

4.1.2.4 CreditsMenu

The CreditsMenu class displays information about the development teams as well as company acknowledgments and copyright information in the form of static text. It does not execute any behavior, nor does it have any inputs/outputs.

4.1.2.5 HighScoreMenu

The HighscoreMenu class is the interface that displays of the top 10 scores achieved in the game and initiates its updating with newer highscores. The entries are kept in a member variable list, the nodes of which contain the score, death count, date of achievement and the user's entered name. Upon instantiation, the past highscores are provided by the HighscoreDataHandler using the HighscoreDataHandler.gethighscores() method, and are displayed to the screen.

The close() method can be called by Button labeled "OK".

A flag member variable gameOver is set to true when the HighscoreMenu is instantiated at the end of a game, meaning the user has attained a highscore, which is passed to the constructor. The user is then prompted to enter a name and the highscore is inserted in the ordered list, and the last node removed. This also passes the list as a parameter to the HighscoreDataHandler in the HighscoreDataHandler.setHighscores(list<Highscore>).

The formatting of the list being passed to and from the HighscoreDataHandler must match the format expected by the HighscoreMenu.

4.2 Data Handling

4.2.1 Overview

The Data Handler module is composed of the HighscoreDataHandler, the SaveDataHandler and the OptionsDataHandler which are all subclasses of the DataHandler abstract class. Each class is instantiated at launch time and loads the data from its assigned file, or failing that, from a default data constant stored in the class. The subclasses also implement their own diskWrite(data), diskRead(), getData() and setData(data) methods. These methods are explained below:

- diskWrite(data) Overwrites the assigned memory file with the data after encoding.
- diskRead() Reads the assigned memory file and assigns its value to data after decoding. If the file is not valid, a default value is assigned.
- getData() Returns a list containing the current state of the data.
- setData(data) Sets data as the DataHandler's current data and writes it to memory using diskWrite(data)

4.2.2 Modules

4.2.2.1 OptionsDataHandler

The OptionsDataHandler stores in a list the current options configuration. It writes to a text file which can be edited directly by the user. It implements the additional setDefault() method:

- setDefault() sets the options configuration to its default state.

4.2.2.2 HighscoreDataHandler

The HighscoreDataHandler stores in a list the current highscores. It writes to a binary file with a simple encryption.

4.2.2.3 SaveDataHandler

The SaveDataHandler stores in a list the signatures of the saved games. These only include the game score, save date and game mode. The signatures are stored along with the each saved games data in a binary file with a simple encryption. It also implements the saveGame(i) and getSave(i) methods explained below:

- saveGame(i) Overwrites the saved game signature in index i of the list and overwrites the corresponding saved game data in the memory file with the current game data, using the diskWrite(data) and diskRead() methods.
- getSave(i) Reads the memory file for the game data corresponding to the saved game signature in index i and returns it.

The data lists used by each DataHandler must be formatted to match the lists used by the Game class and the Menu classes. The DataHandler must also format the data it writes to memory, allowing the data to be reused by the class when reading the file.

4.3 In-game

4.3.1 Game

Once a *Game* object is generated by the *Menu* it keeps track of levels, two player games and saves salient information such as high scores and last level completed between the creation of new *Games* and *Game_Fields*. This subsystem makes the creation of levels and coordination two players modular.

4.3.2 Game_Field

This class is the main node for game play. It Generates and terminates levels, it holds level information and uses this to It generate e*ntities* either at the onset of a new *Game_Field* or circumstantially; like Bullets when called on by a s*hip* and debris when objects are destroyed. It keeps a list of live Entities and tracks collisions to determine termination or the addition of lives. The live list of *entities* is sent by *Game_Field* to be rendered by *Render*. The *Game_Field* keeps track of the

score. In addition to coordinating the rendering of polygons it also coordinates the rendering of strings; the score, the beginning and end of a level, and Game Over. The game sound is coordinated by this class, it is done by a call to the Sound class. *Game_Field* also coordinates pausing, by implementing *input_handler*.

4.3.3 Input_Handler

This class implement *KeyListener* and generates *Key* objects when a key is pressed. *Key* objects are then used by *Game-Field* for pausing and *Space_Ship* for bullets and manoeuvering. It is a sigelton object. The mapping of the control keys can be configured in the *Menu* options.

4.3.4 Key

Key objects are created by the *Input_Handler* and used by *Game-Field* for pausing and the *Space_Ship* for bullets and manoeuvering.

4.3.5 Render

Render implements the graphics libraries to render the game *entities* and *strings*. It is either passed vertices, a point or a string. And has four methods to deal with these parameters. Vertices always render a polygon, ship, asteroid or powerup. A point can be a bullet or an explosion there will be a method for each. Finally a String can be, the score, lives or messages which book end a game or level, such as, start level x, level x finished and game over.

4.3.6 Sound

Called by the *Game_Field* class to render sound, it generates the music in a constant loop and some sound FX. It will use the Java sound libraries to do this.

4.3.7 Entity (Abstract Super Class)

All objects on the game screen except *strings* extend the *Entity* abstract super class. Most importantly they extend the ability to update their position and generate the vertices for their polygon with the knowledge of time since the next update its speed at spawn and it heading. All *entities* use the same coordinate system. They extend the ability to request the creation of any *Entity* such as a Bullet. An *Entity* also keeps track of time it has been alive and can self-destruct.

4.3.8 Classes that Extend Entity

4.3.8.1 Space_Ship

Using *Input_Handler* it updates its speed and angular speed and then its position and vertices. From the *Input_Handler* it also receives commands to shoot Bullets. This, however generates a requests sent to the *Game_field* to generate a Bullet *entity* with the current heading of the *Space_Ship*. The time between firing is also controlled in this class.

4.3.8.2 Large Asteroid

Asteroids are moving polygons, they update their new position with knowledge of the time elapsed since the last update and the speed and initial trajectory since spawn.

4.3.8.3 Small Asteroid

See Large Asteroid.

4.3.8.4 Large_Alien

Update like an *Asteroids*. They have the added feature that they can shoot *Bullets*. They do this by sending an *Entity* creation request to the *Game_Field* with a heading identical to theirs.

4.3.8.5 Small_Alien

Update like an *Asteroids*, they can shoot *Bullets*, however these bullets are aimed in the general direction of the *Space_Ship*. They do this by sending an *Entity* creation request to the *Game_Field* with a heading in the vicinity of the *Space_Ship*.

4.3.8.6 Bullets

These *entities* are points not polygons, but their position *update* process is the same. Their unique properties are that they are generated by a request made by an *Entity* and that they expire after a set time. They do this by checking the time elapsed since spawn and set a terminate boolean when their time has expired.

4.3.8.7 Explosion

Explosions are *entities* created by the *Game_field* when a polygon is terminated. Like *bullets* they are points and *update* like a *bullet*. The animation of an *explosion* is dealt with in *Render*. Like *bullets* they also expire using time elapsed since spawn and a Boolean *terminate*.

4.3.8.8 Power_Up

A *Power_UP*, is a simple moving polygon with a single heading like an *Asteroid*, however it has an expiration time on it. After a certain time has elapsed it sets its *terminate* Boolean to true and is no longer rendered.

5. Analysis

5.1 SRS Summary

Security requirements such as encoding to prevent tampering of saved games and high scores are implemented in the write methods in the Data Handling interface. The graphical interface of each element is described under each of the particular windows, however layout cannot be described outright, an example of potential main menu layout can be seen in Figure 1.

Timing requirements outlined in the requirements document are difficult to portray in architecture. However, the requirements are testable and may be modified to reflect the actual activity.

Sound effects and user controls are documented in their own classes.

The chosen architecture provides flexibility and reuse by implementing interfaces and class hierarchy. These provide methods to control what all classes under a specific interface must or can do.

5.2 Traceability Matrix

Please consult the appendix to map requirement IDs to their corresponding requirements.

5.2.1 Mode 1: Menu System

Requirement ID	Module 1: GUI	Module 2: Data Handling	Module 3: In-game

1.2.1	Method : load(), close() in LoadMenu	Method: getSavedGames(), getSave(), diskRead() in SaveDataHandler	Method: start() in Game
1.2.2	Method: close() in MainMenu	Method: getOptions() in OpitonsDataHandler	Method: start() in Game
1.2.3	Method: close() in MainMenu	Method: getOptions() in OpitonsDataHandler	Method: start() in Game
1.2.4	Method: close() in MainMenu	Method: getOptions() in OpitonsDataHandler	Method: start() in Game
1.2.5	Variable: HighScoreButton in MainMenu		
1.2.6	Variable: InstructionsButton, OptionsButton in MainMenu		
1.2.7	Method: exit() in MainMenu, OptionsMenu, HighscoreMenu, LoadMenu, Credits Menu, InstructionsMenu		
1.2.8	Variable: CreditsButton in MainMenu		

5.2.2 Mode 2: In-Game

Requirement ID	Module 1: GUI	Module 2: Data Handling	Module 3: In-game
2.1.1			- get_key() in Space_Ship - get_thrust_key(), get_hyperspace_key(), get_left_key(), get_right_key(), get_shoot_key() in Input_Handler - is_pressed() in Key
2.1.2			- update_entity_list() in Game_field - update_speed(), update_angle(), entity_gen_request() in Space_Ship

2.1.3			- entity_gen_request() in Space_Ship - get_entity_list() in Game_field - Collision() in Game_Field
2.1.4			- Get_Position() in Space_ship - Collision() in Game_field
2.1.5			- generate_entity(), update_entity_list() in Game_Field
2.1.6			- generate_entity(), update_entity_list() in Game_Field
2.1.7			- get_delta(), generate_entity(), get_entity_list(), update_entity_list() in Game_Field
2.1.8			- get_entity_list() in Game_field
2.1.9			- get_left_key(), get_right_key() in Input_Handler - is_pressed() in Key - get_key(), update_angle(), rotate() in Space_ship
2.1.10			- get_thrust_key() in Input_Handler - is_pressed() in Key - get_key(), update_speed() in Space_ship
2.1.11			- get_space_key() in Input_Handler - is_pressed() in Key - update_position() in Space_Ship
2.1.12			- update_speed() in Space_Ship
2.1.13			Variable: speed in Entity
2.1.14			Variable: anguler_speed in Space_Ship

2.1.15			- update_speed() in Space_Ship
2.1.16			- Variable: speed in Bullet - update_speed() in Space_Ship
2.1.17			- render() in Game_Field -get_position() in Entity
2.1.18			- render() in Game_Field -get_position() in Entity
2.1.19			- render() in Game_Field -get_position() in Entity
2.1.20			- can_fire() in Space_Ship - fire() in Large Alien - aimAndFire() in Small Alien
2.1.21			Variable: angle method: entity_gen_request() in class: Entity,
2.1.22			Method: collision(), in Game_Field
2.1.23			Variable: delta; method: can_fire() in Space_Ship
2.1.24			Variable speed; method: generate_entity(), in Game_Field
2.1.25			Variable: delta; terminate; method: calculate_life() in Bullet
2.1.26			Variable: terminate; method: collision() in Game_Field
2.1.27			Method: collision(); in Game_Field
2.1.28			Method: collision(); in Game_Field
2.1.29			Variable: speed; Method: generate_entity(); in Game_Field

2.1.30			Variable: angle; method: entity_gen_request() in Large_Alien
2.1.31			Variables: position, angle; method: get_ship_position(), aim(), entity_gen_request(), in small alien
2.1.32			Variable: delta; method: can_fire() in Small/Large Alien
2.1.33			Variable: terminate, lives; method: collision() in Game_Field
2.1.34			Variable: terminate; method: collision() in Game_Field
2.1.35			method: collision() and generate_entity() in Game_Field
2.1.36			Variable: score, method: set_score()
2.1.37			Variable: terminate; method: collision()in Game_Field
2.1.38			method: collision() and generate_entity() in Game_Field
2.1.39			Variables: terminate, delta; method: calculate_life() in Power_Up
2.1.40			Variable: lives, weapon, Method: collision(), set_weapon() in Game_field & Space_Ship
2.1.41			Variable: weapon; method: set_weapon(); in Game_Field
2.1.42			Variable: score; method: set_score() in Game_Field()
2.1.43			Variable: game_on, level ;

			method:terminate_level; start_single() in Game; Game_Field
2.1.44			Variable: game_on, level ; method:terminate_level; start_single(), generate_entity() in Game; Game_Field
2.1.45			Variable: level, speed method: generate_entity(), in Game_Field & Entity
2.1.46			Variable: pause; method: set_pause(); in Game_Field

5.2.3 Mode 3: High Score Screen

Requirement ID	Module 1: GUI	Module 2: Data Handling	Module 3: In-game
3.0.1	Variable: highscoreList in HighscoreMenu	Method: getHighscores() in HighscoreDataHandler	
3.0.2	Interface: JWindow		
3.2.1	Variable: gameOver, highscoreList Method: getGameOver() in HighscoreMenu	Method: getHighscores(), setHighscore() in HighscoreDataHandler	
3.2.2		Method: setHighscore() in HighscoreDataHandler	
3.2.3	Method: close() in HighScoreMenu		
3.2.4	Variable: highscoreList in HighscoreMenu	Variable: data in HighscoreDataHandler	

5.2.4 Mode 4: Game Instructions and Preferences

Requirement ID	Module 1: GUI	Module 2: Data Handling	Module 3: In-game
4.0.1	Variable: picList in InstructionMenu		
4.0.2		Method: diskRead() in SaveDataHandler	

4.0.3	Variable: optionsList, dropDownMenuLeft1, dropDownMenuLeft2, dropDownMenuRight1, dropDownMenuRight2 dropDownMenuThrust1 dropDownMenuThrust2 dropDownMenuFire1 dropDownMenuFire2 Method: save() in OptionsMenu	Method: getOptions(), setOptions() in OptionsDataHandler	
4.0.4	Interface: JWindow		
4.2.1	(implemented by JavaSwing API)		
4.2.2	Variable: dropDownMenuLeft1, dropDownMenuLeft2, dropDownMenuRight1, dropDownMenuRight2 dropDownMenuThrust1 dropDownMenuThrust2 dropDownMenuFire1 dropDownMenuFire2		
4.2.3	Method: close() in OptionosMenu	Method: diskWrite(), setOptions() in OptionsDataHandler	
4.2.4	Variable: dropDownMenuLeft1, dropDownMenuLeft2, dropDownMenuRight1, dropDownMenuRight2 dropDownMenuThrust1 dropDownMenuThrust2 dropDownMenuFire1 dropDownMenuFire2		
4.2.5	Method: isValid() in OptionsMenu		
4.2.6		Method: setOptions(), diskRead() in OptionsDataHandler	
4.2.7	Method: close() in OptionsMenu	Method: setDefault() in OptionsDataHandler	

4.2.8	Method: save() in OptionsMenu	Method: setOptions(), diskWrite() in OptionsDataHandler	

5.2.5 Mode 5: Credits

Requirement ID	Module 1: GUI	Module 2: Data Handling	Module 3: In-game
5.0.1	Variable: credits in CreditsMenu		
5.2.1	Method: close() in CreditsMenu		
5.2.2	Variable: credits in CreditsMenu		

5.2.6 Mode 6: Pause Menu

Requirement ID	Module 1: GUI	Module 2: Data Handling	Module 3: In-game
6.0.1	- exit() in Pause Menu	- saveGame() in SavedGameDataHandler	- set_pause() in Game_Field
6.0.2	- Game Window - Pause Menu		
6.2.1	- exit() in Pause Menu		- set_pause() in Game_Field
6.2.2	- close() in Pause Menu - open() in Window	- save() in optionsMenu - saveGame() in SavedGameDataHandler - setData(), diskWrite() in DataHandler	
6.2.3	- close() in Game Window - open() in MainMenu		

6. Design Rationale

6.1 Module 1: GUI

The Menu system was made to implement JWindow to allow the user to close a specific menu dialogue without having to exit the game, the menu system or the application. For example if a user were to pause the game during play time, the pause menu would appear on top of the game field, without removing the later. The user could then close the pause menu and continue play without

having to reload the game. Likewise, when accessing a submenu from the main menu, the user can close the menu without having to reload the main menu, which improves performance. The user also has the option of exiting at any point in the application, which is one of the requirements.

The rooting of the Menu system at the MainMenu class allows the user to access each submenu with only one click, improving user navigation speed. With the same rational, the different modes of play (versus, solo and co-op) can be accessed directly from the Main Menu and the user does not need to change anything in the options to do so. This improves the speed and ease with which the user can launch the game.

The layout of the OptionMenu was done with the rational of centralizing all user-adjustable settings apart from the game mode, to minimize the user's need of navigating the menu. Drop down menus were used for their ability to provide the user with all valid key choices.

The separation of the Menu system from the Data Handler system improves modularity by separating the front end user interface from the backend memory writing/reading. This also reduces the reliance of the Game system on the menu system for launch settings as the options and saved games are passed through the Data Handler system. Exceptions to this are the adding of new highscores and the starting of the Game system, which depends on the user choosing a game mode. Default data values are hardcoded in the DataHandler classes to ensure proper functionality should the memory files be rendered invalid and allows the game to run without their existence.

6.2 Module 2: Data Handling

The LoadDataHandler class only passes the signatures of the saved games to avoid the need to load all saved games from memory at launch time and boost performance. Only when the user chooses to load a game is the corresponding game data loaded from memory.

6.3 Module 3: In-Game

The game has been designed with minimal coupling in mind. All elements of the game will have their own class that will be responsible for them. For example, the ship, aliens, and asteroids will all be separate types.

This separation, or decoupling, encourages good practice and reduces the risk of small changes damaging large portions of the game. The best example of this is the logical versus graphical parts of the game. These two parts will be completely separate. The game's entities will be responsible for maintaining their state while a separate class will be responsible for drawing them on the screen.

Game entities will inherit from a common Entity type that proves the basis by which all entities can maintain a state on the playing field. A renderer will be able to determine what type of Entity needs to be drawn and will be responsible for drawing the Entity. Similarly, there will be components that are responsible for the sound, configuration, and user input.

7. Workload Breakdown

The work has been divided into two main parts: the game, and all that wraps around the game (menus, data handling and graphical interface). Two teams of three were been assigned to these tasks:

- Alex Coco, Alex Bourdon, Yi Qing Xiao assigned to game development [team 1].
- Payom Meshgin, Jad Sayegh, Daniel Ranga assigned to rest (graphics, menus, data handling) [team 2].

7.1 Team 1: Game Development

Alex Coco is the team leader and is responsible for the basic foundation of the game engine, while Yi Qing Xiao and Alex Bourdon will work on details and testing.

Alex Coco will make the base classes with their variables and methods while more precise tasks will be given to the other two such as object collision tweaking, entities, field wrap around conditions, ship fire rate, differentiating the players' shots in the case of coop play, alien generating conditions, etc.

Game testing will be mostly done by Yi Qing (game engine), Alex Bourdon (game engine), and Payom Meshgin (game feel).

7.2 Team 2: Menu

All three members will work on the menu system. Anything related to the menu such as the menu links (generation of sub-menus) to the sub classes such as the high score board, the initialization of

the in-game, how to transition between each menu classes, what are the conditions for changing pages, where the save files will be placed, in what format will they be saved, .etc.

Payom Meshgin will work on the graphics representation of the menu items: which pages are open graphically in which class, which pages are to be closed when transitioning between classes, update displayed variable values when changing key allocations, decide whether frame is fixed or not, how to dampen one page display without dampening another one, etc.

Payom Meshgin and Daniel Ranga will work on data handling. Testing the data handling will be initially tested by Daniel.

APPENDIX A: REQUIREMENTS (FROM SRS DOCUMENT)

I. MODE I: MENU SYSTEM

I.0 GENERAL FUNCTIONALITY

1.0.1 The menu system provides a way for the user to access other parts of the game as well as change settings for the game. It is either a separate window or a menu bar embedded into the gameplay window **(essential, easy)**.

I.1 EXTERNAL INTERFACES

1.1.1 The main way to interact with the menu will be with the display and the mouse **(essential, easy)**.

1.1.2 The menu window includes the following buttons : 'Load Game', 'New Single Player Game', 'New Two-Player Contest', 'New Coop Game', 'High Score', 'Game Instructions and Preferences', 'Credits', 'Exit' **(essential, easy)**.

1.1.3 All windows (Menu, High Score, Gameplay, etc.) will include an exit button, in the top right corner, marked as an 'X' which will close the game, cancelling any activities that are occurring **(essential, medium)**.

1.1.4 When the user clicks on a button in the menu, the game will respond appropriately and display the correct sub-menu such as an options page or help dialog **(essential, medium)**.

I.2 FUNCTIONAL REQUIREMENTS

1.2.1 Clicking the 'Load Game' button will close the main menu window, open the gameplay window and load a previously stored game **(essential, medium)**.

1.2.2 Clicking the 'New Single Player Game' button will close the main menu window, open the playing field window and will start a new single player game **(essential, medium)**.

1.2.3 Clicking the 'New Coop Game' button will close the main menu window, open the game play window and will start a new game in which two players play together on the same game play field **(desirable, hard)**.

1.2.4 Clicking the 'New Two-Player Contest' button will close the main menu window, open the game play window and will start a new game in which two players play one after the other **(essential, hard)**.

1.2.5 Clicking the 'High Score' button will open the High Score window (leaving the main menu open) **(essential, medium)**.

1.2.6 Clicking the 'Game Instructions and Preferences' button will open the Control/How-To-Play window (leaving the main menu open) **(desirable, medium)**.

1.2.7 Clicking the 'Exit' button will close the program **(essential, medium)**.

1.2.8 Clicking the 'Credits' button will open the Credits window (leaving the main menu open) **(desirable, medium)**.

1.3 Performance

1.3.1 Starting the game (i.e. loading the Main menu screen) will take up to 1 second from the moment the executable starts running **(essential, medium)**.

1.3.2 The "hitbox" of any button will correspond to its displayed size **(essential, medium).**

1.3.3 Transitions from the Main menu to any other non-gameplay window will be executed within 500ms **(essential, medium)**.

1.3.4 Transitions from the Main menu to any new game gameplay window will be executed within 1 second **(essential, medium)**.

1.3.5 Transitions from the Main menu to loading a saved game to the gameplay window will be executed within 1.5 second **(essential, medium)**.

1.3.6 Exiting the game will not take more than 5 seconds **(essential, medium)**.

2. Mode 2: In-Game

This section discusses the actual gameplay component of the game. These requirements are specific to the time between the game has been started and the player has lost or ended the game. These

requirements involve the player, the player's ship in the game, and the enemy units in the game **(asteroids and aliens)**.

2.0 External Interfaces

2.0.1 During the game, the user will interact with the system with their keyboard **(essential, medium)**.

2.0.2 Almost all feedback to the user will be delivered via the display **(essential, medium)**.

2.0.3 Some additional feedback may be present in the form of sound effects **(desirable, difficult)**.

2.1 Functional Requirements

2.1.1 A player will be able to interact with his or her ship using the keyboard **(essential, medium)**.

2.1.2 At each spawn of a new ship, which occurs at the beginning of every level (or after the PC has lost a life and still has some remaining), the playable character is instantiated in the middle of the playing field, pointing upward **(essential, easy)**.

2.1.3 If there is an obstacle near the center of the field (any objects near the center of the game field), the spawn of a new ship is delayed until there is no object within a radius of the center **(essential, medium)**.

2.1.4 Exception to above: in coop, partner's ship is not considered an obstacle. **(desirable, medium)**

2.1.5 At the beginning of every level, between four to six large asteroids are randomly initialized at the edges of the playing field **(essential, medium)**.

2.1.6 Aliens are instantiated at the left or right edges of the playing field **(essential, medium)**.

2.1.7 Aliens appear at random intervals of time, not exceeding 10 seconds **(essential, medium)**.

2.1.8 Only one alien may be present on the field at any time **(essential, medium)**.

2.1.9 They will be able to turn their ship clockwise or counter-clockwise using two different keys on the keyboard **(essential, medium)**.

2.1.10 The ship can accelerate in the direction the ship is presently facing by holding a key **(essential, medium)**.

2.1.11 The ship is able to enter "hyperdrive" which will allow the ship to disappear and reappear at a random location on the screen using a key on the keyboard **(desirable, medium)**.

2.1.12 The ship may be subject to inertia: the ship will continue to move in the direction it was moving while it slows down **(optional, difficult)**.

2.1.13 All other objects (shots, aliens, asteroids) travel at a constant speed **(essential, easy)**.

2.1.14 Ship turning speed is constant **(essential, easy)**.

2.1.15 The player's ship can accelerate until it reaches some terminal velocity **(essential, easy).**

2.1.16 The terminal speed of player's ship must be below the bullet speed **(essential, easy)**.

2.1.17 The ship, asteroids and shots will be able to wrap around the screen. In other words, if they pass the edge of the screen they will continue moving in the same direction but will appear at the opposite edge **(desirable, difficult)**.

2.1.18 When wrapping around at the vertical limits (right and left), objects maintain vertical axis position **(desirable, difficult)**.

2.1.19 When wrapping around at the horizontal limits (up and down), objects maintain horizontal axis position **(desirable, difficult)**.

2.1.20 The player's ship and the aliens will be able to fire shots **(essential, medium)**.

2.1.21 The player's ship can only fire shots in the direction they are facing **(essential, easy)**.

2.1.22 The player's ship is destroyed upon any form of collision with the following exceptions: Collision with a power-up; Collision with partner in coop mode; Collision with partner's bullet in coop mode **(desirable, difficult)**.

2.1.23 The player can only have a maximum of 4 shots fired appearing on the field at all time. **(essential, medium)**

2.1.24 All bullets have the same constant speed. **(essential, medium)**

2.1.25 The shots fired have a limited time to live in the game and will disappear after this time has expired **(essential, medium)**.

2.1.26 When the shots collide with an enemy unit the shots will disappear **(essential, easy)**.

2.1.27 Shots disappear on collision except when the object they collide with is another bullet or a power-up object **(essential, medium)**.

2.1.28 The objects' hitboxes will be rectangular and must correspond as closely as possible to the minimal size possible which still fully encapsulate the objects' graphically represented images **(essential, medium)**.

2.1.29 Aliens move at a constant speed and at a random direction (maintaining that direction, thus moving linearly) and change direction at a constant frequency **(essential, hard)**.

2.1.30 Big aliens shoot at random directions **(essential, medium)**.

2.1.31 Small aliens shoot in the general direction of the player's ship

2.1.32 Aliens shoot at a firing rate of 1shot/3sec **(essential, medium)**.

2.1.33 When aliens interact with a shot, it will either die or lose hit points until they reach zero, then die **(essential, medium)**.

2.1.34 If an asteroid collides with anything except another asteroid or a power-up, it will be deleted **(essential, difficult)**.

2.1.35 If a deleted asteroid has a non-minimal size, a number of smaller asteroids of same size with different velocities (values depend on the velocity and size of the original) will be generated nearby the last position of the destroyed asteroid. **(essential, medium)**.

2.1.36 Every time the player's ship fires a shot and hits an enemy, it will gain points (the amount is to be determined) **(essential, easy)**.

2.1.37 When an Alien collides with anything except a power up, it is destroyed **(essential, difficult)**.

2.1.38 When an enemy is destroyed, there is a small chance (10%) that a power up is dropped **(optional, medium)**.

2.1.39 Power-up drops remain static on the field and are destroyed when their lifespan of 4 sec is over or when a player's ship collide with it **(optional, medium)**.

2.1.40 When the player's ship collides with the power up, the ship will gain points, receive an ability or upgraded weapon depending on what power up was gained **(optional, difficult)**.

2.1.41 Any previously acquired weapon bonus will be replaced with the newly acquired one **(optional, difficult)**.

2.1.42 If the player is able to clear the screen of all the asteroids, then they receive extra points **(essential, easy).**

2.1.43 If the player is able to clear the screen of all the asteroids, a new level begins, while maintaining the position and momentum of the ship. **(essential, easy).**

2.1.44 The next level starts with a new set of asteroids, which spawn a minimal distance away from the present position of the ship **(desirable, medium)**.

2.1.45 As the player reaches a higher level, asteroids move at a gradually higher speed (10% more per level) and small aliens have a greater chance to appear than large aliens (10% more) **(desirable, medium)**.

2.1.46 Clicking the pause button will cause the game to freeze and the pause menu to open **(desirable, medium)**.

2.2 PERFORMANCE

2.2.1 The player's remaining number of lives cannot exceed 10 lives **(desirable, medium)**.

2.2.2 The player's score is stored in a 32-bit unsigned integer **(desirable, easy)**.

2.2.3 However improbable it may be, if the player's score exceeds the maximum bound of un unsigned integer, the score remains static at that maximum value **(desirable, easy)**

2.2.4 The game will run on one machine only. Up to two users may be supported but it will not be over a network. Cooperative play will use the same keyboard for two players **(optional, medium)**.

2.2.5 There will be no writing of files during the gameplay **(optional, medium)**.

2.2.6 The game is expected to update and render over 10 times per second, ideally at a variable frame rate and controlled update count **(desirable, medium)**. As such, the responsiveness for the gameplay may depend in part on the hardware the user is running the game on.

2.2.7 The game will target a large number of standard hardware and should run on most modern household computers **(desirable, medium)**.

3. Mode 3: High Score Screen

3.0 General Functionality

3.0.1 The high score screen allows the player to view the highest scores achieved in the game on that machine. The window will contain the 10 best scores and the associated text to the score, and saves made during this game session **(desirable, easy)**.

3.0.2 While high score board is open, main menu display will not disappear, but dampened or blackened to allow focus on the score board **(desirable, easy)**.

3.1 External Interfaces

3.1.1 The high score screen will be an extension of the menu system. This screen will interface with the display and mouse. The screen will also include an 'OK' button **(desirable, easy)**.

3.1.2 File system access will be required in order to store the score information **(desirable, easy)**.

3.2 Functional Requirements

3.2.1 After a game, once the High Score screen is open, if the player's score is better than the last high score, the player's score is inserted below the score immediately larger in the scoreboard. Scores below the current score are shifted down. Scores beyond the 10th score in the scoreboards are removed from the list **(essential, medium)**.

3.2.2 Once the score shift occurs the player is permitted to add a name next to the score. The score can be composed of no more than 32 printable ASCII characters **(essential, medium)**.

3.2.3 Clicking the 'OK' button will close the High Score screen and return focus to the Main menu window **(essential, medium)**.

3.2.4 The High Score screen will also have a field displaying the number of games played and lost lives **(essential, medium)**.

3.3 Performance

3.3.1 All stored saved game information is to be saved in binary form as to prevent unauthorized modification. **(desirable, easy)**

3.3.2 Returning to the Main menu (via the 'OK' button) will be executed within 500ms **(desirable, easy)**.

4. Mode 4: Game Instructions and Preferences

4.0 General Functionality

4.0.1 The Game Instructions and Preferences screen allows the player to view the different game sprites that appear in the game and understand their interactions with the spaceship, and show the effects of the different controls **(desirable, hard)**.

4.0.2 The first time the game is launched the Game Instructions and Preferences screen will appear, to help the user understand the game dynamics **(desirable, easy)**.

4.0.3 In the settings menu, the user will be allowed to change the controls for the game **(desirable, medium)**. In this situation, keyboard interaction will be required as well as the mouse and display.

4.0.4 While the Game Instructions and Preferences screen is open, main menu display will not disappear, but dampened or blackened to allow focus on the game instruction and preferences screen **(essential, medium)**.

4.1 External Interfaces

4.1.1 The screen will interface with the display, keyboard and mouse **(essential, easy)**.

4.1.2 The screen will include static text describing the game, images illustrating game elements and sprites, and labeled field where the user can choose game controls (for both users). The screen will also include an 'OK' button **(desired, medium)**.

4.1.3 The screen will include controls for volume and enabling and disabling features **(optional, medium)**.

4.2 Functional Requirements

4.2.1 Options are displayed on a list, from up to down, with in-game control at top section, sound check on the bottom **(desirable, easy)**.

4.2.2 Player 1's options should be separated from player 2's options, though both can be manipulated by the same person **(desirable, easy)**.

4.2.3 Clicking the 'OK' button will save all changes made, close the Game Instructions and Preferences screen and return focus to the Main menu window **(essential, medium)**

4.2.4 Any of the controls can be changed to any keyboard key **(desirable, hard)**.

4.2.5 If a key is already assigned to a control, no other control can be assigned to that key **(desirable, medium)**.

4.2.6 The game begins with a set configuration of controls for both players **(essential, easy)**.

4.2.7 A default button resets all key assignments to default. **(desirable, easy)**

4.2.8 Settings are saved when exiting the screen **(essential, easy)**.

4.3 Performance

4.3.1 All stored preference information is to be saved in binary form as to prevent unauthorized modification. **(desirable, easy)**

4.3.2 Returning to the Main menu (via the 'OK' button) will be executed within 500ms **(desirable, easy)**.

5. Mode 5: Credits

5.0 General Functionality

5.0.1 The Credits screen contains static text, it contains developer names, company logo and copyright information **(desired, easy)**.

5.1 External Interfaces

5.1.1 The Credits screen interacts with the screen and the mouse, it has only an 'OK' button **(essential, easy)**.

5.2 Functional Requirements

5.2.1 Clicking the 'OK' button will close the Credits screen and return focus to the Main menu window **(essential, medium)**.

5.2.2 Components are listed from top to down with the producer's name and position **(essential, easy)**.

5.3 Performance

5.3.1 Returning to the Main menu (via the 'OK' button) will be executed within 500ms **(desirable, easy)**.

6. Mode 6: Pause Menu

6.0 General Functionality

6.0.1 In the Pause Menu the game is stopped and may either be resumed or saved **(desirable, easy)**.

6.0.2 While pause menu is open, the game display will not disappear, but will be dampened or blackened to allow focus on the pause menu **(desirable, medium)**.

6.1 External Interfaces

6.1.1 The Pause menu interacts with the screen and the mouse, it has a 'Resume Game' button and a 'Save and Exit' button **(essential, easy)**.

6.2 Functional Requirements

6.2.1 Clicking the 'Resume Game' button will close the Pause menu and return focus to the In-Game screen, continuing the game **(essential, easy)**.

6.2.2 Clicking the 'Save' button will save the game, close the In-Game window and return focus to the Main menu **(essential, easy)**.

6.2.3 Clicking the 'Exit' button will close the In-Game window and return focus to the Main menu **(essential, easy)**.

6.3 Performance

6.3.1 Returning to gameplay will executed within 500ms **(essential, easy)**.

6.3.2 Saving the game and returning to the Main menu will be executed within 1 second **(essential, easy)**.

6.3.3 Pausing cannot be done more than once per 5 second to prevent abuse of pause ability **(desirable, medium)**.