BILKENT UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

CS 319 – OBJECT ORIENTED SOFTWARE ENGINEERING

SECTION 03 – GROUP 3L

PROJECT TITLE: Space Impact

DESIGN REPORT
DRAFT

Team Members

Büşra Arabacı
Büşra Oğuzoğlu
Gerard Hysa
Alperen Kaya
Deniz Şen

1. Introduction **4**
1.1 Purpose of the system 4
1.2 Design goals
2. Software Architecture **6**
2.1 Subsystem Decomposition 6
2.2 Hardware/Software Mapping 7
2.3 Architectural Styles 8
2.4 Persistent Data Management 8
2.5 Access Control and Security 8
2.6 Boundary Conditions 8
3. Subsystem Services 9
3.1 Detailed Object Design 9
3.2 User Interface Subsystem 9
3.2.1 Menu Interface 10
3.2.2 MainMenu Class 10
3.2.3 GameOverMenu Class 10
3.2.4 LevelSelectionMenu Class 10
3.2.5 PauseMenu Class 10
3.2.6 SpaceShipControlPanel Class 11
3.3 Game Management Subsystem 11
3.3.1 StateManager Class 11
3.3.2 State Interface 11
3.3.3 MenuState Class 12
3.3.4 LevelScreenState Class 12
3.3.5 GameState Class 12
3.3.6 PauseStateClass 12
3.3.7 GameFinishedState Class 12
3.3.8 StoreState Class 12
3.3.9 AboutUsState Class 13
3.3.10 LevelManager Class 13
3.3.11 InputManager Class 13
3.3.12 GameEngine Class 13
3.4 Game Elements Subsystem 14
3.4.1 GameObject Interface 14
3.4.2 GameMap Class 14
3.4.3 GameLevel Class 14

3.4.4 Spaceship Abstract Class 14
3.4.5 NPC Abstract Class 14
3.4.6 UserSpaceShip Class 14
3.4.7 EnemySpaceship Class 14
3.4.8 BossSpaceship Class 15
3.4.9 Collectable Interface 15
3.4.10 Coin Class 15
3.4.11 Reward Class 15
3.4.12 Buyable Interface 15
3.4.13 PowerUp Class 15
3.4.14 Upgrade Class 15
3.4.15 Bullet Class 15

Glossary 16

1. Introduction

1.1 Purpose of the system

Space Impact is an old arcade game which is based on shooting the enemies and completing levels. In our project, we are recreating this old classical game to make an appealing version of it and adapt it to the new technologies and styles. To reach this goal, we are planning to provide a system that keeps the old and loved features of the game and adds new features that enriches and supports them. Diverse game elements such as enemies, rewards, upgrades and power ups gives the novelty and excitement that the game needs. In addition, our system aims to provide comfortable user interface and glorified visual environment that consummates the overall gaming experience.

1.2 Design goals

In order to decide the design of the overall system and create subsystems effectively, the importance of determining design goals cannot be overestimated. Therefore, before the system design, we examine and discuss non-functional requirements carefully and clarify the design goals. Design goals of our system is described in the following sections.

Ease of use:

The main purpose of our system is to provide an enjoyable gaming experience to user so it is very important to make the gaming environment easy to use and not distracting. Since the complications in game controls, game environment or game logic can easily affect user and cause them to leave the system, we consider this goal important. Our system will allow user to easily move the spaceship with finger and use a well-designed spaceship control panel to use other features of the spaceship. Also, by providing clear menus and easy to access pause menu by lifting finger from screen, our system will ensure easy user interface.

Ease of learning:

Like ease of use goal, ease of learning is important to gain user's interest and give them a good experience. Since the game has tutorials and explanations in it, it will be easier for user to learn the game. Also, in store, all upgrades and power ups will have descriptions, user can easily understand their purpose.

Extendibility:

Since keeping user's attention and always enjoying them is a crucial aim for games, extendibility of the system gains importance. The system will be able to create new levels with different maps, enemy types and difficulties. Also, our system will be suitable for new game elements, new features and functions because it will separate subsystems carefully and creating new functionalities in a subsystem will not affect the overall game.

Portability:

Portability is an important goal for our system because we want to make a new version of a game that will draw attention to users that know and love old version. This requires a platform that can reach wide range of users. We decided to use Android environment which is a widely used platform.

Efficiency:

Since games are dynamic and gaining or losing points occur immediately, they should provide continuous and error-free experience. Our system aims to provide a gaming environment that responds to user's actions quickly and runs smoothly to prevent user from losing points that arise from system errors. In order to achieve this state, our system will create continuous game screen scroll and uninterrupted spaceship control facility. Also, our system will have quick response time to user's actions in order to keep the attention of users.

Trade offs:

Ease Of Use and Ease of Learning vs. Functionality:

The main concern of our system is to entertain user and keep user's attention longer. To achieve this, it is important to make the system both easy to use and easy to learn. Complicated systems tend to decrease the motivation of user to continue the game which contradicts our system's main goals. This means our system will avoid complicated and extra functionalities. We tried to carefully examine the user's actions and avoid unnecessary functions and actions for both user and game.

Performance vs. Memory:

Our system cares creating impressive visual effects and animations which is a high cost for memory. For example, explosion effects, shooting effects and power ups effects are considered important for game and our system will try to make them realistic as much as possible. However, creating these effects may lower the performance which is an important concern for the system. So, we decided to increase the memory not to compromise performance.

2. Software Architecture

2.1 Subsystem Decomposition

In order to see how the system is divided, subsystems are created. Subsystems interact with each other independently so to create a software that can be easily extended, modified and has great performance. In this way the division into subsystems makes it easier even for the team during the implementation stage. Everything will be ordered and have its own place.

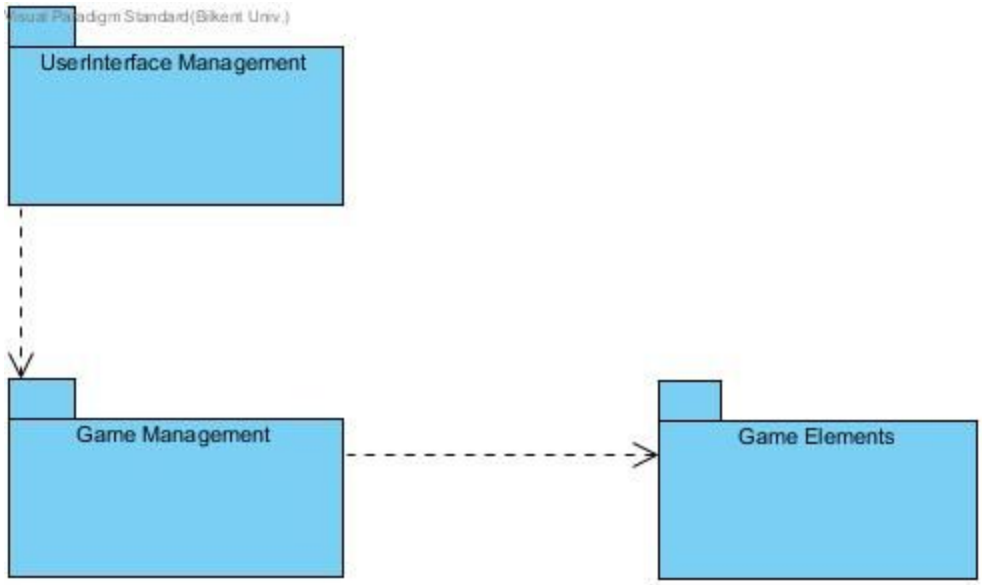


Figure 2.1 Subsystem Decomposition Representation

In Figure 2.1 the subsystem decomposition is represented which consists of three subsystems that interact with each other. User Interface Management interacts with Game Management which is closely related and interacts with the Game Elements. A division of this manner gives the opportunity to locate faster a bug without interfering with the other parts. As it will be seen in the deeper look below the classes that perform tasks related to each other are grouped in a subsystem. In order to implement this software MVC architectural pattern will be used which will give us the need organization of the project. In this way we will have a model, a view and a controller. The controller is the subsystem which will send and get data back and forth from the view and the model so to create an interaction between them.

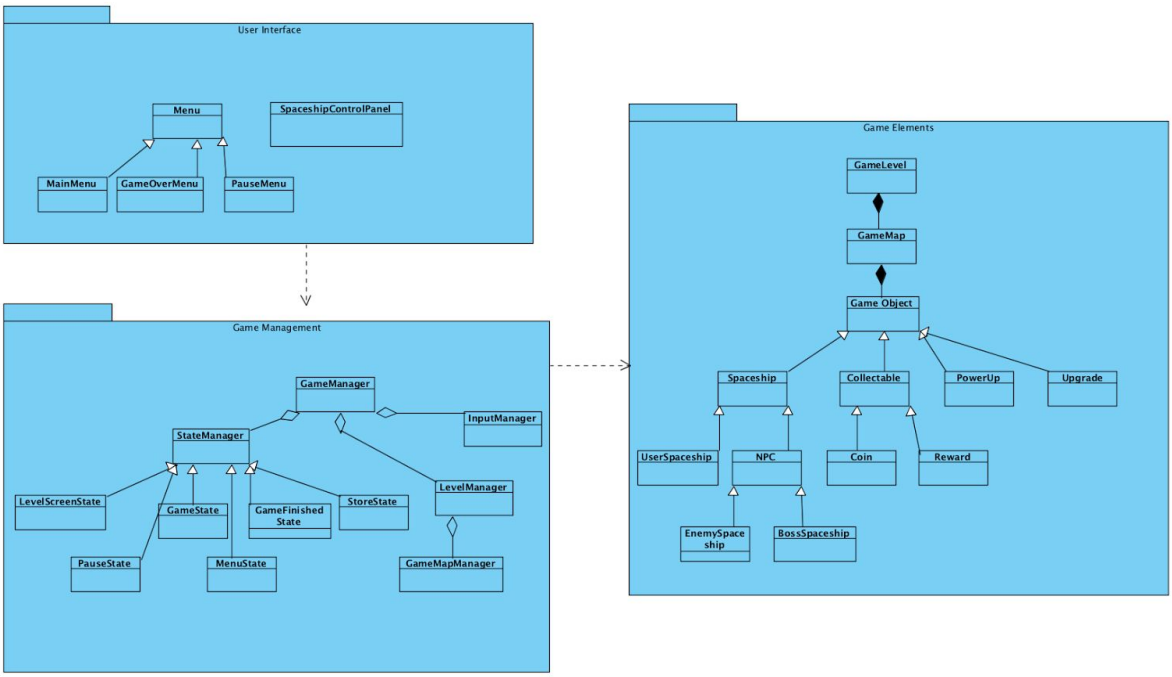


Figure 2.2 Subsystems detailed

In Figure 2.2 a deeper look into the subsystems composition is shown. The User Interface Management will contain the “Menu” class from where the user can select from the different options displayed. “Menu” class has extended classes such as “Pause Menu”, “Main Menu”, “Game Over”. From the User Interface Management different requests from the user are sent to the Game Management subsystem which contains the “GameManager” class which will control all the requests and get all the necessaries from the Game Elements subsystem. In Game Management subsystem all the inputs from the user will be received and the state of the game will be updated. In Game Elements subsystem are located all the needed classes for the objects of the game.

2.2 Hardware/Software Mapping

Our game will be implemented in Java and Android, since it will be a game played in mobile phones. In order to implement this in android we will need all the SDK tools and a Java Runtime Environment. The game will be played in phone, so the phone itself will be the hardware needed to play the game. The player by using touch can control the game so no extra hardware needed for this game. Android has a custom 2D graphics library which will help us to draw our animations. The program will not contain heavy graphics but still we will keep a balance between nice gameplay and usability so almost everyone can play it in their phones with

different android versions. The program will not require any connection to the internet to store, but everything will be stored in the phone's memory

2.3 Architectural Styles

For this game we will use the MVC architectural style which consists of three subsystems that interact with each other, the model, the view and the controller. By using this style the view part which is the User Interface Management is put apart and forms a separate subsystem. The model of the MVC consists of our Game Elements subsystem where all the objects of the game are divided into classes and interact with each other. In order to provide the connection between the model and the view, the controller stands between them sending and receiving requests. The controller part consists of our Game Management subsystem, where the requests from the view are handled from the manager classes and the needed data is sent to the model. MVC is a great choice for games since it provides a smooth work during the implementation and mostly all of the changes that happen in a part do not affect the other parts.

2.4 Persistent Data Management

Space Impact will only use the internal hard drive of the device. The game will not use any kind of cloud or server storage to keep the data. The data will be saved before and after the game time since the saving data on the storage is an expensive process. Saving the data during the game might cause stuttering issues for the game. The game will load the background and figure images, sprites and sounds during the gaming process as expected. These data will be stored in the internal data as well.

2.5 Access Control and Security

Space Impact does not require any kind of authentication, therefore the game does not require any kind of personal data to play the game. For that, the security is not a big concern for this game. Thus, the game data will not be encrypted. The game data will be only accessible through the "Game Management" package, which not only ensures the security of the data, but also make the data more easy to manipulate.

2.6 Boundary Conditions

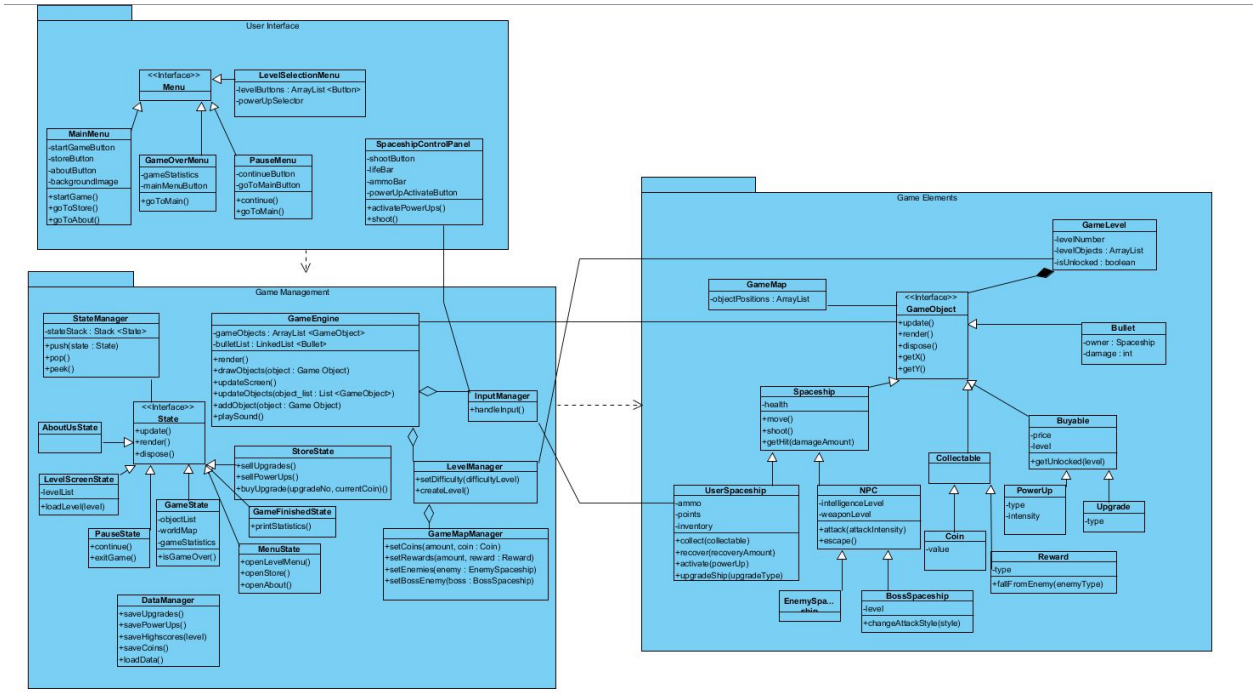
The game will check for a user data file at the start of the program. If such file does not exist, the game will automatically create a new one and put it under usage. If the game ends either by the player completing the level or losing all the health, the game will open the result screen and

in the case of gaining the highest score of the level, the data file will update the highest score of the level.

3. Subsystem Services

3.1 Detailed Object Design

To understand the subsystems, their behaviours and relationships with other subsystems, detailed object diagram is provided. This diagram also shows classes in subsystems which is abstracted in subsystem decomposition part for simplicity. While creating this model, associations and behaviours of classes examined carefully and subsystems from closely related classes.



3.2 User Interface Subsystem

Providing user interface for playing game and using features of the game is an important part of the overall project. User Interface Subsystem is responsible for presenting well-designed, easy-to-use interface to user, collecting actions and informations from user and transmit them to Game Management Subsystem. There are Menu classes to receive user's preferences for application such as starting game or going to store. Also, another important interface is needed when playing game in order to control spaceship which is handled by SpaceshipControlPanel.

This class is in charge of providing interface to user for power-ups management, shooting and checking remaining lifetime, bullets or power-ups.

3.2.1 Menu Interface

This is an interface for the menu classes. Even though now this interface does not have any impact on the current system, however for the future changes, gathering up the classes inside an interface will be useful.

3.2.2 MainMenu Class

This class has the main menu's buttons and actions. The MenuState will render what this class tells to render. It has 3 buttons as attributes and 1 image as a background image. Each button will lead the user at a different screen and state. The background image will be rendered and drawn by the help of the methods that the MenuState contains.

3.2.3 GameOverMenu Class

This class is the user interface class of the GameFinishedState. It appears when the game is over by either victory or defeat. It contains 1 button which makes the game return to main menu. It has an attribute which holds the statistics that were collected by the game. The menu will use these statistics and write them on the screen.

3.2.4 LevelSelectionMenu Class

This menu will appear before the user starts the game. Here, the level selection will be made as well as the starting power-up selection. This menu has an array of buttons all of which indicate the number of a certain level. Also the power-up selector will determine the power-up selection of the user.

3.2.5 PauseMenu Class

This menu appears whenever the user releases their finger from the game screen while the GameState is on. This menu consists of 2 buttons that are used for either going back to the main menu or continuing the game. It has 2 methods that are called depending on the selection of the user. The continueButton will call the continue() method, whereas the goToMainButton will call the goToMain() method.

3.2.6 SpaceShipControlPanel Class

This class is the only class which is not inherited from the Menu interface inside the package. The user interface of the gaming process is constituted by 2 parts. The surface on which the user leads the spaceship is not located in this package. The other part is the dock which contains the control buttons and the indicators of the spaceship. The buttons are shootButton which triggers the shoot() method, and the powerUpActivateButton which calls the activatePowerUp() method. The indicators are the lifeBar which shows the remaining health and the ammoBar which indicates the remaining ammo.

3.3 Game Management Subsystem

Game Management Subsytem is the main place where all actions from user is processed and all game elements are controlled accordingly. Since the task of the subsystem is wide, it includes more classes. The GameManager class combines the StateManager, LevelManager and InputManager classes. Input manager class takes user inputs from UserInterface Subsystem and converts them to actions which can be used to control game and other features of application. LevelManager class is responsible for creating and maintaining levels by deciding the difficulty levels of each level, where coins and rewards are located in levels, which type of enemies come which part of the level and their behaviours according to the level difficulty. The most important and detailed class in this subsystem is StateManager class. It consists of subclasses called LevelScreenState, PauseState, GameState, MenuState, GameFinishedState and StoreState. GameState class is in charge of overall gameplay process.

3.3.1 StateManager Class

The tasks are handled as states in our game. This class manages these states in a stack data structure. For instance, when the game is first loaded, StateManager pushes the MenuState inside the stack. Then, if the user gets in the store, StateManager pushes the StoreState inside the stack. When the user presses the back button, StoreState is popped and MenuState starts being used. As expected, only the state that is on the top of the stack is being used, and the others are not used. When the state on the top is popped, it is disposed by StateManager.

3.3.2 State Interface

This interface gathers 5 states of the game. All the states must have an update() method which updates the current state according to the user inputs, a render() method which draws the state on the screen, and a dispose() method which removes the state from the memory when it is no

longer needed. This interface helps the StateManager a lot in terms of the manipulation of the states.

3.3.3 MenuState Class

This class is the handler of the main menu. It draws the menu on the screen. It has takes commands from the MainMenu class and functions accordingly. It has 3 methods which are similar to the MainMenu class. The difference is that these methods are used to render the menu.

3.3.4 LevelScreenState Class

This state is where the level numbers are shown. It has a levelList which holds the levels and whether they are unlocked or not.Depending on the level selection, the game loads the level and passes the state to the GameState.

3.3.5 GameState Class

This is the main state of the game. The game is loaded in this state. It holds the list of the game objects, a worldMap and the statistics of the game. Its main purpose is to draw the game on the screen. It has a method that indicates whether the game is finished or not. If the game is finished, the game loop is terminated.

3.3.6 PauseStateClass

This state is loaded when the user releases their finger from the screen. While this state is on, the game loop stops counting. When the user decides to stop the game, the StateManager pops the PauseState and the GameState. If the user decides to continue gaming, only the PauseState is popped from the stack.

3.3.7 GameFinishedState Class

This state is pushed on the stack when the game is over. It shows the statistics of the game and the achievements that were achieved during the gameplay.

3.3.8 StoreState Class

This state is used for drawing the store usage. The user can buy upgrades for the spaceship while this state is on the screen. This class has 3 methods. The buyUpgrade(upgradeNo, currentCoin) method takes 2 parameters. The first parameter indicates which upgrade is

requested to be bought and the second parameter takes the coin that the user has currently. If the coin is not enough, the purchase will be unsuccessful. The sellUpgrade() method is called whenever the user wants to sell an upgrade. Similarly, the user can sell their powerups when the sellPowerUps() function.

3.3.9 AboutUsState Class

This state does not contain any method as it has no function. It only shows some information about the developers of the game.

3.3.10 LevelManager Class

LevelManager class is the generator of the level. It has 2 methods which of them is setDifficulty. This method takes the difficulty level as an input which sets the difficulty of the proceeding level. The other method is createLevel method which is the main method of this class. This method creates the level map by the help of the GameMapManager which sends the positions of the objects.

3.3.11 InputManager Class

This class will be responsible for handling the user inputs. It will return different values according to the current user inputs. By these inputs, during the game, some methods will be called, such as move method of the UserSpaceShip object.

3.3.12 GameEngine Class

This class is the main class which will handle the changes on the game map, as well as the changes on the screen. GameEngine is used while GameState is on the top of the StateManager stack. This class gathers changes of every single game element of the game and helps GameState to only call the GameEngine in order to render the game on the screen. GameEngine has 2 attributes, gameObjects and bulletList. The gameObjects is an ArrayList of GameObjects which holds all the game objects that are currently on the map. The bulletList is a linked list of bullet objects. This list is important to easily dispose bullets that have left the game map or hit a target. Without this list, the game will suffer stuttering issues because Bullet objects will be created very often, thus the unused ones must be disposed as soon as possible. The GameEngine will also be responsible of playing the sounds of the game.

3.4 Game Elements Subsystem

3.4.1 GameObject Interface

This interface is very important for the simplicity of the system. Everything that the user sees on the gameplay, must be a GameObject which has an x-coordinate and a y-coordinate. Also, every object should draw itself on the screen with the render() method, update its position with update() method and remove itself from the memory when it is no longer used with dispose() method.

3.4.2 GameMap Class

The map of the game holds the objects along with their positions. Then, they are drawn on the screen by the help of the GameEngine.

3.4.3 GameLevel Class

This class is used to connect the packages Game Management and Game Elements. It will hold the level information and have the list of objects before the game starts. It will have an attribute which holds whether the level is unlocked or not.

3.4.4 Spaceship Abstract Class

The subjects of the game are the spaceships and there can be several types of spaceships. Every spaceship must have some commonalities, which this abstract class puts as obligation. Every spaceship must have an amount of maximum health, have a move() method, have a shoot() method and have a getHit(damageAmount) method.

3.4.5 NPC Abstract Class

This class is for the non-player characters that are the enemy spaceships. These spaceships have a fixed intelligence level and a weapon level to indicate their strength. The escape() method is used when an allie bullet hits the spaceship, so it will not take any damage.

3.4.6 UserSpaceShip Class

This the spaceship which the user controls. It has a limited amount of ammo and an inventory which holds the collected things. It might regain health during the game and activate power-ups.

3.4.7 EnemySpaceship Class

This class does not have a special method to operate because the methods that NPC abstract class obligates are enough.

3.4.8 BossSpaceship Class

This kind of spaceship has some different kinds of shooting styles which can be changed using the changeAttackStyle(style) method. Every boss appear at different levels, so it is indicated by the level attribute of the object.

3.4.9 Collectable Interface

This is an interface which covers game objects that can be collected during the gameplay. It is only needed for easy manipulation purposes.

3.4.10 Coin Class

This object has only 1 attribute that is its value since as the game proceeds, the values of the coins might increase.

3.4.11 Reward Class

Rewards are collectable during the gameplay. Every reward is different from each other, so they have a type which is holded as a property.

3.4.12 Buyable Interface

This interface gathers the objects that can only be bought from the store. They must have a price and a level of upgrade. Their unlocks are done by the getUnlocked(level) method.

3.4.13 PowerUp Class

This object is used during the gameplay but can only be received from the store by purchase. The power-ups have a particular type which is a property of the object.

3.4.14 Upgrade Class

This class indicates the upgrades that can be added to the user's spaceship, for instance bullet strength. The type of the upgrade is a property of the object.

3.4.15 Bullet Class

This object is used to indicate the bullets that come from the spaceships. Each bullet has an owner and a damage amount. These values are hold as properties of each object. Since this

object will be created many times during the game, its manipulation is very important or it might cause some severe stuttering issues.

4. Glossary