Bilkent University

Department of Computer Engineering

Object Oriented Software Engineering Project

CS 319 Project: Split-Field

Analysis Report

Ali İlteriş TABAK, Efe TUNÇGENÇ, Gizem ÇAYLAK

Course Instructor: Hüseyin Özgür TAN

Progress

Oct 28, 2015

Table of Contents

1. Introduction

"Split-Field" is a platform game which is consisted of different number of levels. The game success depends on player's quick time responses, strategic capabilities and enemy tracking. Split-Field's main goal is to push players' skills to the limits while aiming them to have fun. In Split-field there are different number of levels. In each level there is a major enemy and a few minor enemies with it. Our main character is a spaceship named Line-Rider. It can move on the outer boundaries of the field. On the map, there are also power-up boxes, which give different variety of enhancement to the Line-Rider. These power-up boxes are not mandatory, but optional. The player is able to finish the game without even using them. Using them is not affecting the score: there is no penalty. But a clever player is welcomed to use them to get higher scores in the game.

Main purpose in a level is to cut off an area while not getting touched by the major or minor enemies. To do that, Line-Rider should be directed into the field from "safe" boundary lines. After completing and reserving a portion of the field, it should return to safe boundary lines before contacting an enemy in the field. In the field, Line-Rider becomes vulnerable to enemy contact. After completing a portion-cutting operation, that field would be removed from the original field and remaining boundary becomes a "safe" boundary for our Line-Rider to move in. Player moves to next level if %80 of the field is captured.

Game is originally based upon a 90's DOS game named "Volfied", which is also based upon a Commodore64 game named "Qix" made by Taito Company [1].

Split-Field will be a PC game written in Java, aiming to use the object oriented model programming. It should run on every kind of operating system since Java supports all of them.

2. Requirement Analysis

2.1. Overview

As previously stated above, Split-field is a platform-action game. Game is consisted of finishing all the levels in the game without losing all of lives given at the start. Total number of levels will be determined in implementation stage of our project. The data of level numbers will be held out of program code. It will be held in a text file. Thus after its implementation, we would be able to change its design aspects without even touching to our source code.

In our game, player will be welcomed with the main menu. After selecting "Play Game" in the menu, the user is asked if they want to start a new one or continue from a previously saved game. If user selects new game, if user selects new game, level 1 will commence right away.

In every level, our main character (Line-Rider) will start from center of bottom line of the field. It will have three lives given from start. There is also a shield surrounding the Line-Rider and its capacity is designed to last in 900 seconds. This amount is illustrated in the left corner of screen and it will decrease as time flies. If remaining shield capacity reaches zero, Line-Rider will be vulnerable to all enemies whether it is on a main line or not.

While capturing an area, Line-Rider is also vulnerable to enemies. Player must make decisions between getting caught and losing a life or getting a bigger chunk of the field.

2.1.1. Control

Players are able to control our main character via using move keys on the keyboard; up arrow, down arrow, left arrow and right arrow. Additionally, users are not allowed to go into field while they are on main lines. If user decides go into field for split operation, they must press and hold the spacebar to enter into it. After entering, holding down spacebar is not necessary to continue moving.

2.1.2. Line-Rider

Line-Rider is our spaceship, which is controlled by the Player. It has got a shield, which lasts 900 seconds before finishing the level.

Line-Rider is also capable of splitting the field in any shape player desired it to be. Its movement capability allows turning 90 degrees left or right. So any concave or convex rectangular shape could be cut from the field.

Figure 1: Playable character "Line-Rider" [2]

2.1.3. Enemies

There are different kind of enemies in every level; major enemy and minor enemies. In each level enemies vary in standards of characteristic, design and speed. After Line-Rider enters the field, the become slightly faster than their previous patrolling speed.

Figure 2: Some of major enemies from different levels [2]

2.1.4. Power-ups

There are 4 types of power-ups in every level to help player win the level. Placements of power-ups differ for every level. These power-ups are wrapped in a power-up box before they are acquired. Splitting the area, they exist, and cutting them from original field does require them. After requiring, they are explained with a logo where they were when they were a box. Here is the list of all four power-ups:

: Clear all minor enemies from field.

: Line-Rider moves with more speed for a short time.

: Pause shield decrement for a few seconds.

: Line-Rider is able to shoot laser if spacebar is pressed. Lasers destroy only minor enemies, not major ones.

2.2. Functional Requirements

2.2.1. New Game

The player can choose "New Game" option from the play game menu to start a game from very beginning. The default parameters determined by the system will be used to construct the levels and the player will start from the first level of the default levels.

2.2.2. Play Game

In each level, the in-game character controlled by the player will be located to middle of the bottom line of the game field and, enemies and power-ups will be distributed randomly to the game field by the system. The attributes of the enemies and power-ups will vary each level. Enemies will be able to move in the game field and the control of the player character will be provided by the "arrow" keys, but power-ups will be steady throughout the game. When the player is at the edges of the game field, the enemy will not be able to harm the player, but in the game field if enemy contacts with the player the health of the player will decrease, the line player create will be destroyed and the player will be located to the beginning point of the line. The purpose of the game is cutting the 80% of the game field before the health of the player, which can be damaged by enemies, ends. To cut an area, initially the player must press on the "space" key and then draw the lines of cutting

area by "arrow" keys. The cutting line should start and end on any edge of the game field, and the other points of the cutting line should contact with the game field. Since these cutting rules create 2 areas with different sizes separated by lines, the small area will be considered as the cutting area. If there are enemies in the cutting area, the enemies will be destroyed and the player will gain score for each enemy and, if there are power-ups in the cutting area, the player will be able to use them. The player will repeat cutting the area procedure until 20% of the game field remains. If the health of the player ends before 20% of the game field is cut, the game will over and a game over message will disappear in the screen. When the 80% of the game field is cut and the health of the player is not consumed, the player will proceed to the next level which is predetermined by the system. Until the default levels are over, the system will upload next level with different backgrounds, enemies and power-ups. At the beginning of each level, the health of the player will be recharged. The same cutting and playing rules will be applied at each level but the velocity, damage and health magnitude of enemies and, the properties of power-ups will vary as well as the numbers of these game objects. If the player level up all levels by in each level cutting 80% of the game field without consuming its health, he/she will win the game and if he/she score up in high score list, the system will prompt the player to enter his/her name to update high score list.

2.2.3. Load Game

The player will be able to continue from the last saved level in addition to starting to a new game. Within the play game menu, the player can load its progress by clicking on the "Load Game" button, if there is a saved level before and resume the level.

2.2.4. Pause Game

There will be a pause icon in the corner of the game screen to pause the game. The player can pause the game any time during gameplay by clicking on that icon. The player can continue to the game, save its progress, return to the main menu or exit the game from the pause menu.

2.2.5. Save Game

When the player wants to save its progress, the system will allow user to save the last played level. Within the pause menu, the player can save its progress by clicking on the "Save Game" button in order to progress from the level he/she left off as it was detailed in above "Load Game" section.

2.2.6. Exit Game

From the main menu or the pause menu, the player can exit the game by clicking on the "Exit Game" button. Since the game will not provide automatic saving, if the player will not save his/her progress from pause menu manually, he/she will not be able to load the last level he/she played.

2.2.7. Show High Scores

The highest 10 score and their holders will be stored in descending order by the system. The player will be able to see high score table by clicking on the "Show High Scores" button from main menu. If any player has never played the game before, the system will show "No High Score" message to player.

2.2.8. Show Help

The player can obtain information about the game from the main menu. This information will contain player control, the features of the power-ups and other main rules about the game to learn how to play the game.

2.3. Non-Functional Requirements

- **Performance**
  - No other game engine will be used to design the game other than Java GUI itself, so not using complex graphics will reduce the response time of the system.
  - The minimum frame rate must be twenty frames per second for the game run smoother and faster.
  - As the Split-Field requires no internet connection to save or load data, the response time of storage operations will be more time efficient comparing to cloud storage.

- **Usability**
  - The game must have a consistent and user- friendly interface which will provide familiar control environment to the user.
  - The player can exit the game from any menu he/she encountered through the game.

- **Clarity and Smoothness**
  - To provide a satisfactory user experience, the game must have a clear and smooth graphics which will be considered as one of the priorities while designing the game.

2.4. Constraints

- The project will be written using Java programming language.

- GitHub, web-based hosting system, will be used to host project codes.

- The group member will use Eclipse development environment for the implementation of the project.

- Graphic objects such as player character and enemies will be designed using Adobe® Photoshop CS6.

2.5. Scenarios

Use cases will be derived from the following scenarios:

1. **Scenario Name:** playsAndFinishesGame

   Participating actor instances: bob:Player

   Flow of events:

   1) Bob opens the game.

   2) Bob selects "Play game".

   3) Bob chooses one of two options which are starting from first level and continuing from last saved level.

   4) Bob cuts game field in each level until he cuts the %80 of the area and win the game or the enemy touches three times to player during the game and enemy defeats Bob.

   5) Game finishes and Bob is asked to write a name if his score is in top ten high scores.

   6) Bob exits.

2. Scenario Name: <u>saveTheGame</u>

Participating actor instances: <u>bob:Player</u>

Flow of events:

1) Bob opens the game.

2) Bob selects "Play game".

3) Bob chooses one of two options which are starting from first level and continuing from last saved level.

4) Bob pauses the game and sees an in-game menu.

5) Bob selects "Save".

3. Scenario Name: <u>informationAboutGameOrHowToPlayGame</u>

Participating actor instances: <u>bob:Player</u>

Flow of events:

1) Bob opens the game.

2) Bob selects "Help".

4. Scenario Name: <u>lookExistingHighScoresAndTheirHolders</u>

Participating actor instances: <u>bob:Player</u>

Flow of events:

1) Bob opens the game.

2) Bob selects "High Scores".

2.6. Use Case Models

The possible user interactions and use cases of the player with the system is briefly displayed in figure 1 as a use case diagram. Individual use cases will also be described textually after the diagram.
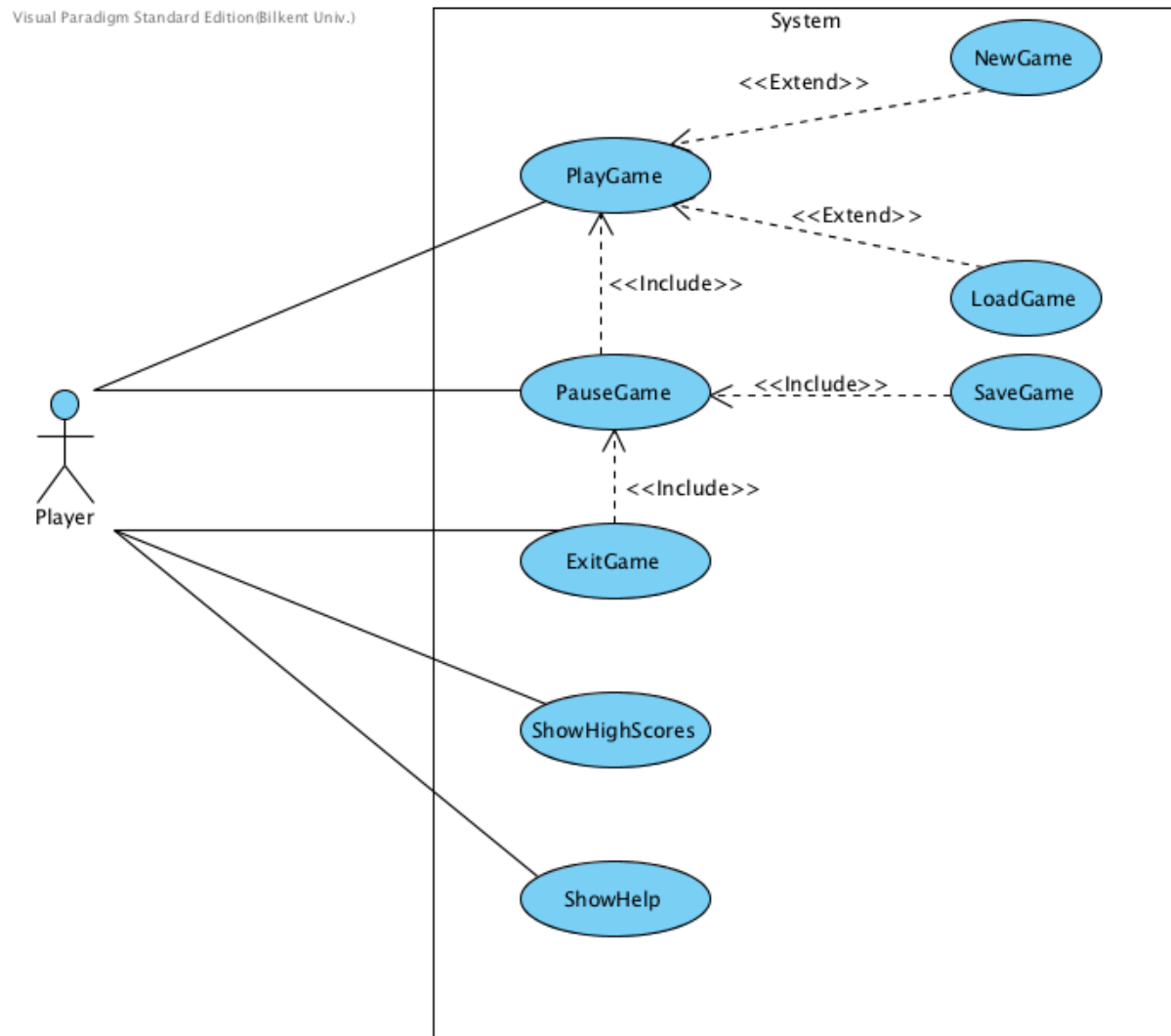

Figure 3. Use Case Diagram

2.6.1. Use Case Descriptions

Use case #1

Use case name: Play game

Participating actors: Player

Pre-condition: The player has already entered main menu.

Post-condition: The system directs player to a new screen that contains new game and load game buttons.

Entry condition: The player clicks the "Play Game" button.

Flow of events:

1. The system displays a menu screen that contains new game and load game buttons.

Exit condition:

- The player chooses to return to main menu.

Exceptions:

- The player returns to main menu.
    - The player clicks on the return button from menu.

<u>Use case #2</u>

Use case name: New game

Participating actors: Player

Pre-condition: Player has already entered the play game menu.

Post-condition: The system shows the high score table.

Entry condition: The player clicks on the new game button.

Flow of events:

1. The system constructs the levels which are determined in "Levels.txt" file by the system.
2. The player encounters with the level.
3. The player starts from middle of the bottom line of the game field.
4. The player cuts the game field until 80% of it is cut.
5. The system directs player into the next level.
    a. Steps 3, 4, 5, 6 are repeated until the game is over.
6. The system displays the score of the player. If player's score is in the top ten scores, the system asks player to write his/her name.
7. The player writes his/her name.
8. The system saves the score and the name, and does modification on the high score table.
9. The system shows the high score table.

Exit condition:

- The player has finished all the levels successfully.
- The player has lost its lives throughout the game before finishing the all the levels successfully.
- The player has clicked pause button and choose to return to the main menu.

Exceptions:

- The player loses all its lives throughout the game before finishing the all the levels successfully.
    1. The player starts with the level.
    2. During the area cutting operation, an enemy touches the player.
    3. The system decrements the number of player's lives.
        a. Steps 1, 2 and 3 are repeated until player loses all of his/her lives.
    4. The system shows a game over message to player.

- The player can pause the game at any time and return to the main menu.

Quality Requirements: At any point during the flow of events, this use case can include the PauseGame use case.

Use case #3

      Use case name: Load game

      Participating actors: Player

      Pre-condition: The player has already entered play menu and there is already a saved state.

      Post-condition: The player continues playing the last saved level.

      Entry condition: The player clicks on the load game button.

      Flow of events:

        1. The system constructs the last saved level and rest of the higher levels from "LastSavedLevel.txt" file.
        2. The player continues playing the last saved level.

      Exit condition:

        ▪ The player starts to play the game from the last saved level.

      Exceptions:

        ▪ If there is no saved state, the player encounters with the first level of the predetermined level set by the system.

Use case #4

Use case name: Pause Game

Participating actors: Player

Pre-condition: Player has already been playing the game.

Post-condition: -

Entry condition: The player clicks on the pause button during the gameplay.

Flow of events:

1. The system displays the in-game menu.
2. The player chooses one of the options in-game menu by clicking buttons which are "Continue", "Save Game", "Return to Main Menu" and "Exit".

Exit condition:

- The player returns to the main menu by clicking on the "Return to Main Menu" button.
- The player returns to the gameplay by clicking on the "Continue" button.

Exceptions: -

Use case #5

Use case name: Save Game

Participating actors: Player

Pre-condition:

1. The player has already been playing the game.
2. The player has clicked on the "Pause" button to be directed in-game menu by the system.

Post-condition: The system displays "Saved Successfully!" to the player.

Entry condition: The player clicks on the "Save" button from in-game menu.

Flow of events:

1. The system modifies the **"**LastSavedLevel.txt**"** file which contains the number of the last saved level.
2. The system displays the message that the game is saved successfully without interrupting the game.

Exit condition: The player clicks on the "Continue" or "Return to Main Menu" buttons from the in-game menu after the game is successfully saved.

Exceptions: None

Use case #6

Use case name: Exit Game

      Participating actors: Player

      Pre-condition: The player has already entered the main menu or in-game menu by clicking on the "Pause" button during gameplay.

      Post-condition: The player exits the game.

      Entry condition: The player clicks on the "Exit" button.

      Flow of events:

1. The system prompts "Are you sure?" and waits for a click from the player on "Yes" or "No" buttons.
2. Player clicks on "Yes".
3. The system closes the main frame and terminates itself.

      Exit condition: None.

      Exceptions: The player clicks on the "No" button and exists the game.

Use case #7

       Use case name: View High Scores

       Participating actors: Player

       Pre-condition: Player has already entered main menu.

       Post-condition: -

       Entry condition: The player clicks on the "High Scores" button that exists in the main menu.

       Flow of events:

          1. The system reads the "HighScores.txt" file which contains the high scores and their holders.
          2. The system displays the high scores in descending order with their holders.

       Exit condition:

          ▪ The player returns to the main menu by clicking on the "Return" button.

       Exceptions:

          ▪ If any player has never played the game before, the system cannot display any high score and its holder. Thus, the system shows "No High Score" message to player.

Use case #8

Use case name: View Help

    Participating actors: Player

    Pre-condition: The player has already entered main menu.

    Post-condition: The system shows a screen that has information about the game and instructions.

    Entry condition: The player clicks on the "Help" button.

    Flow of events:

        1. The system displays information about the game and instructions.

    Exit condition:

        ▪ The player returns to the main menu by clicking on the "Return" button.

    Exceptions: None.

2.7. User Interface

2.7.1. Navigational Path
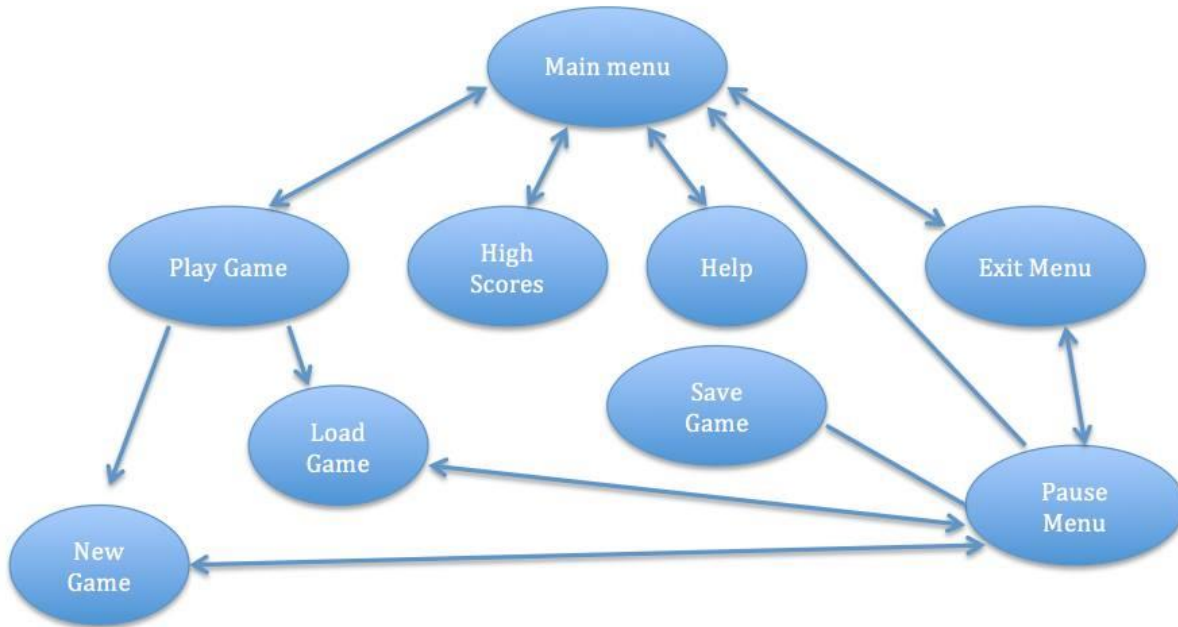
Figure 4. The navigational paths between the game menus

Since when the player chooses the save game option, the system will display a pop-up message and the user will have no control over navigation between pause menu and save game, the non-directional edge is used to indicate the relation between nodes. When the user has control over navigation, two-directional edges are used to indicate the relations between nodes.

2.7.2. Game Menu Mock-ups

The menu designs in this section are not final and their design may change at the final version of the project.

2.7.2.1.   Main Menu

Figure 5. The mock-up of the main menu

Figure 6. The mock-up of the high scores menu.
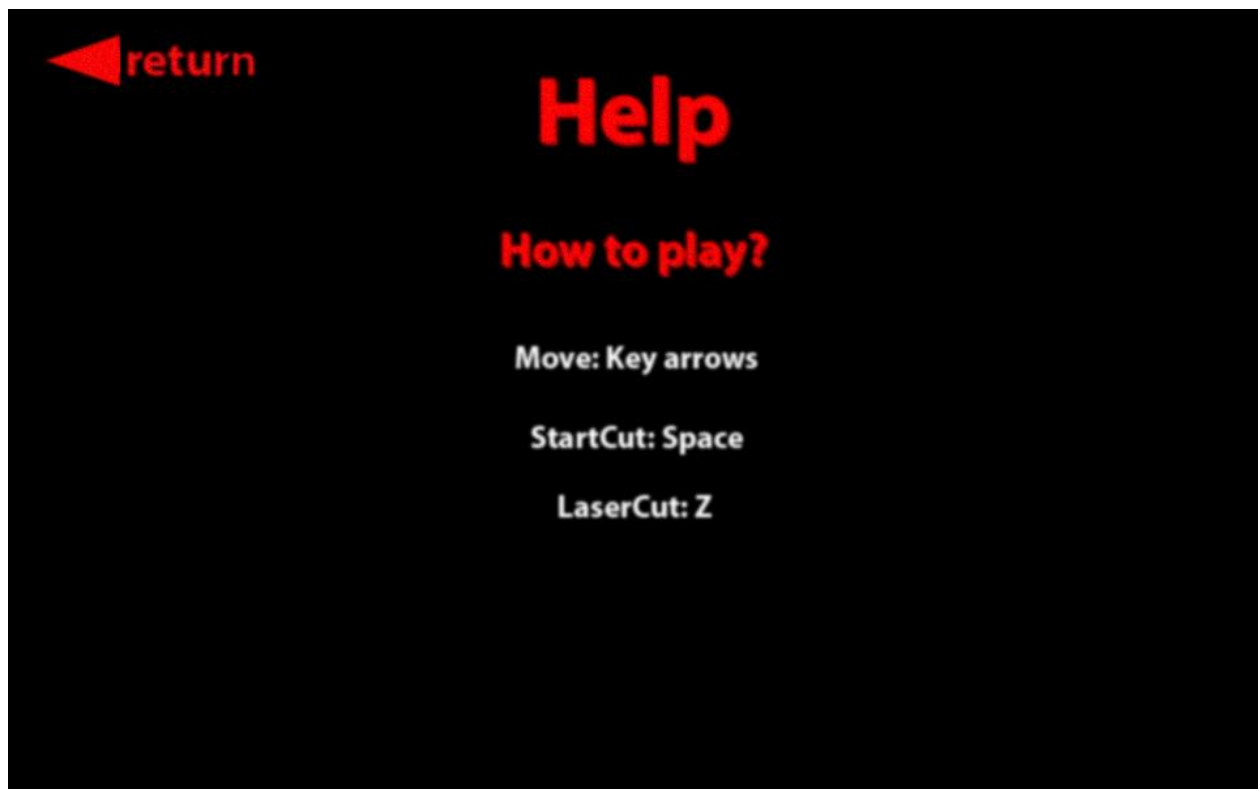
Figure 7. The mock-up of the help menu

2.7.2.4.   Play Game Menu
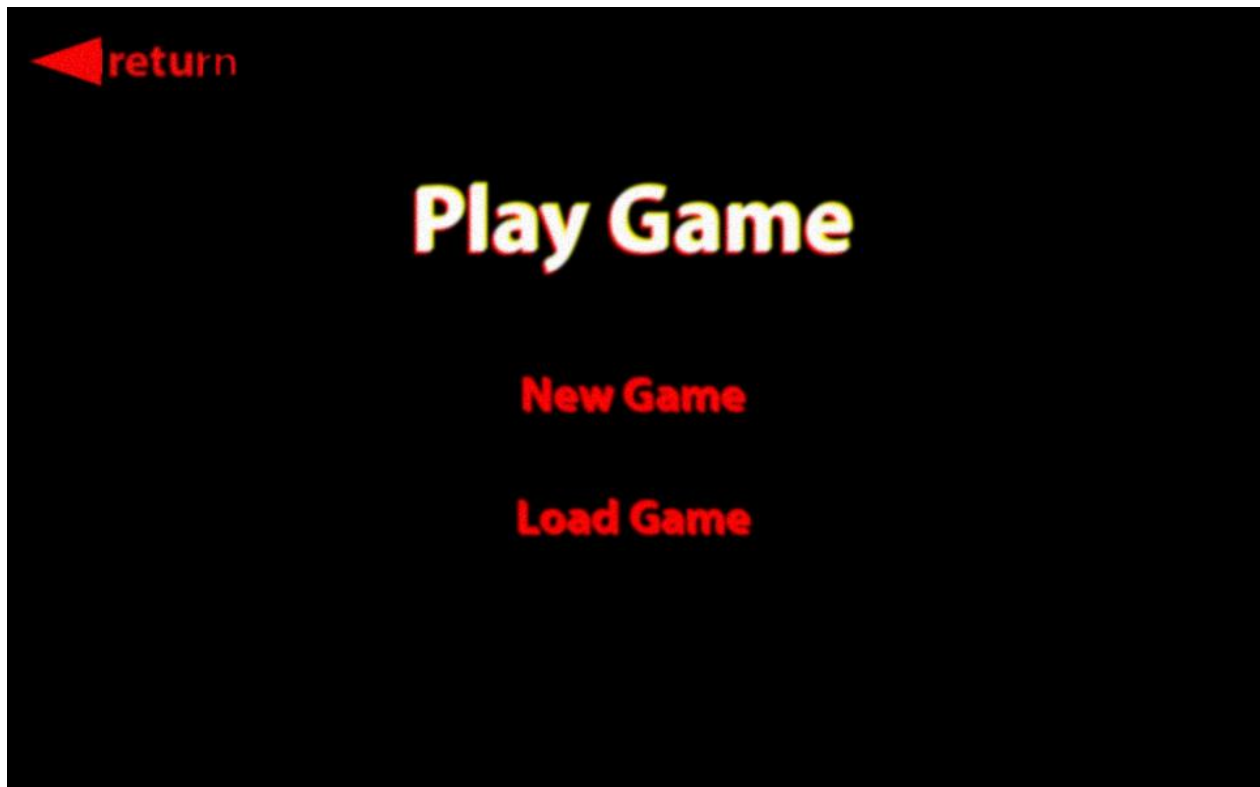
Figure 8. The mock-up of the play game menu

2.7.2.5.   In-game Menu

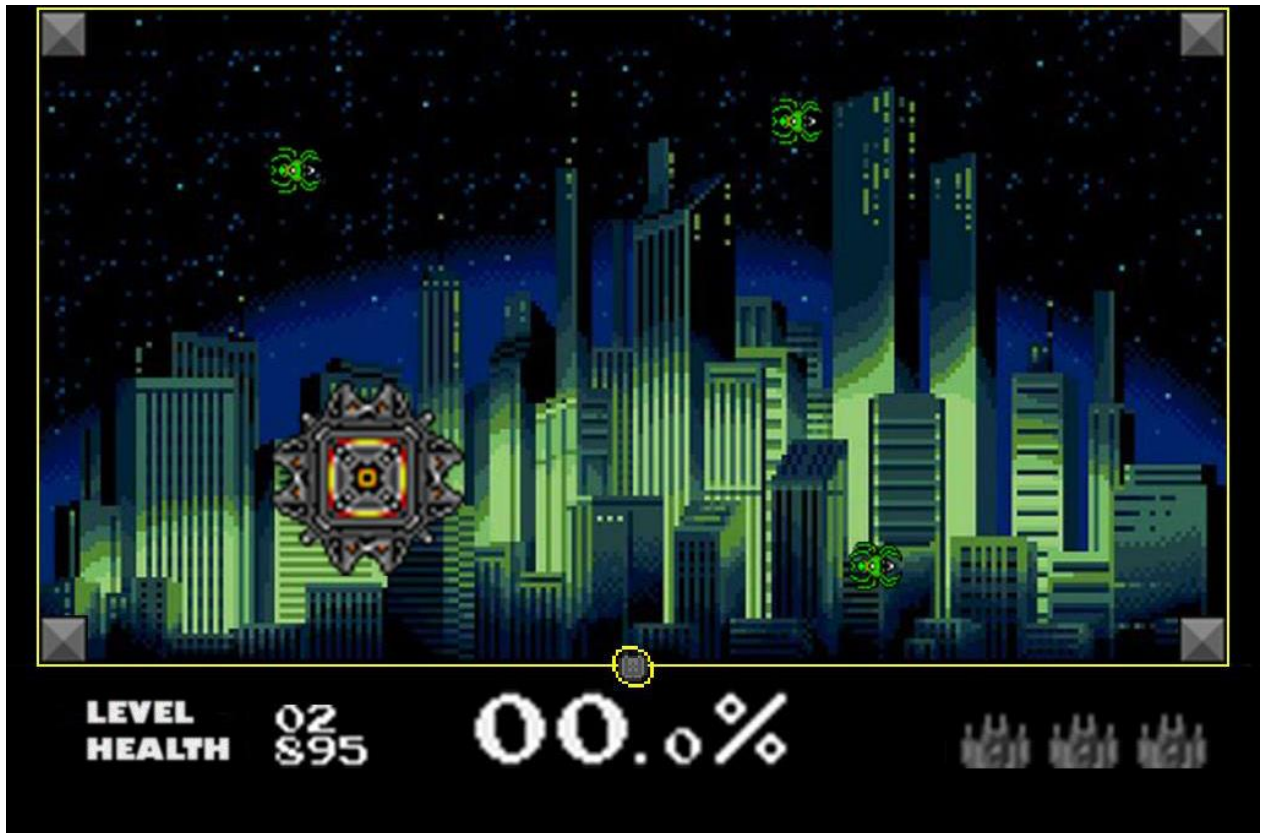Figure 9. The mock-up of the play game menu

2.7.3. In Game Graphics

Figure 10. The mock-up of the game field.

2.7.4. Characters

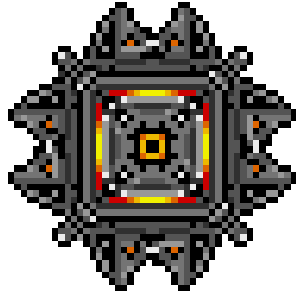#### 2.7.4.1. Line-Rider
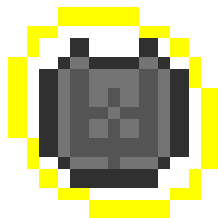
Figure 11. LineRider

- Level 1 Major Enemy   Level 1 Minor Enemy
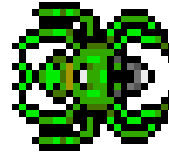
Figure 12. Level 1 Major Enemy   Figure 13. Level 1 Minor Enemy

- Level 2 Major Enemy   Level 2 Minor Enemy
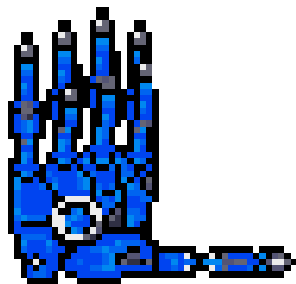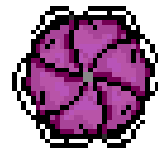

Figure 14. Level 2 Major Enemy   Figure 15. Level 2 Minor Enemy

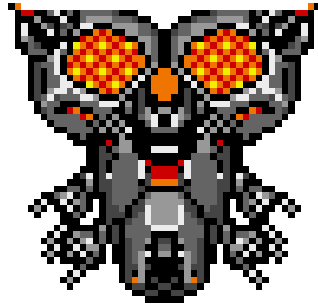- Level 3 Major Enemy   Level 3 Minor Enemy

Figure 16. Level 3 Major Enemy

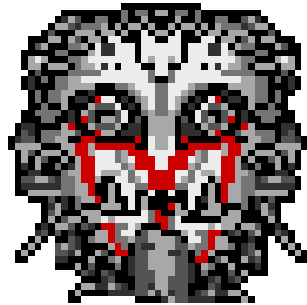Figure 17. Level 3 Minor Enemy

- Level 4 Major Enemy
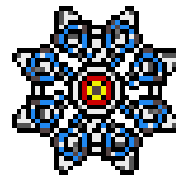
Level 4 Minor Enemy

Figure 18. Level 4 Major Enemy

Figure 19. Level 4 Minor Enemy

- Level 5 Major Enemy

Level 5 Minor Enemy

Figure 20. Level 5 Major Enemy

Figure 21. Level 5 Minor Enemy

2.7.4.3. Power-Ups

- Power-up box
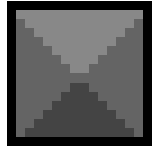
- : Clear all minor enemies from field.

- : Line-Rider moves with more speed for a short time.

- : Pause shield decrement for a few seconds.

- : Line-Rider is able to shoot laser if spacebar is pressed. Lasers destroy only minor enemies, not major ones.

3. Analysis
3.1. Object Model
3.1.1. Domain Lexicon

- **Split-Field:** An arcade game. It is the name of our game.

- **LineRider:** A player character whose actions are controlled by the player.

- **Health:** An attribute assigned to entities that indicates its state in combat.

- **Enemy:** A computer character whose actions are controlled by the system algorithm.

- **Power-ups:** Objects that add extra abilities to the game character.

- **GUI:** Guided User Interface. It is the interface that enables interaction between the user and the system.

- **MVC:** Model/View/Controller. It is an architectural style that is used for designing systems and subsystems.

- **JVM:** Java Virtual Machine which is an abstract computing machine that enables a computer to run a Java program.

3.1.2. Class Diagram

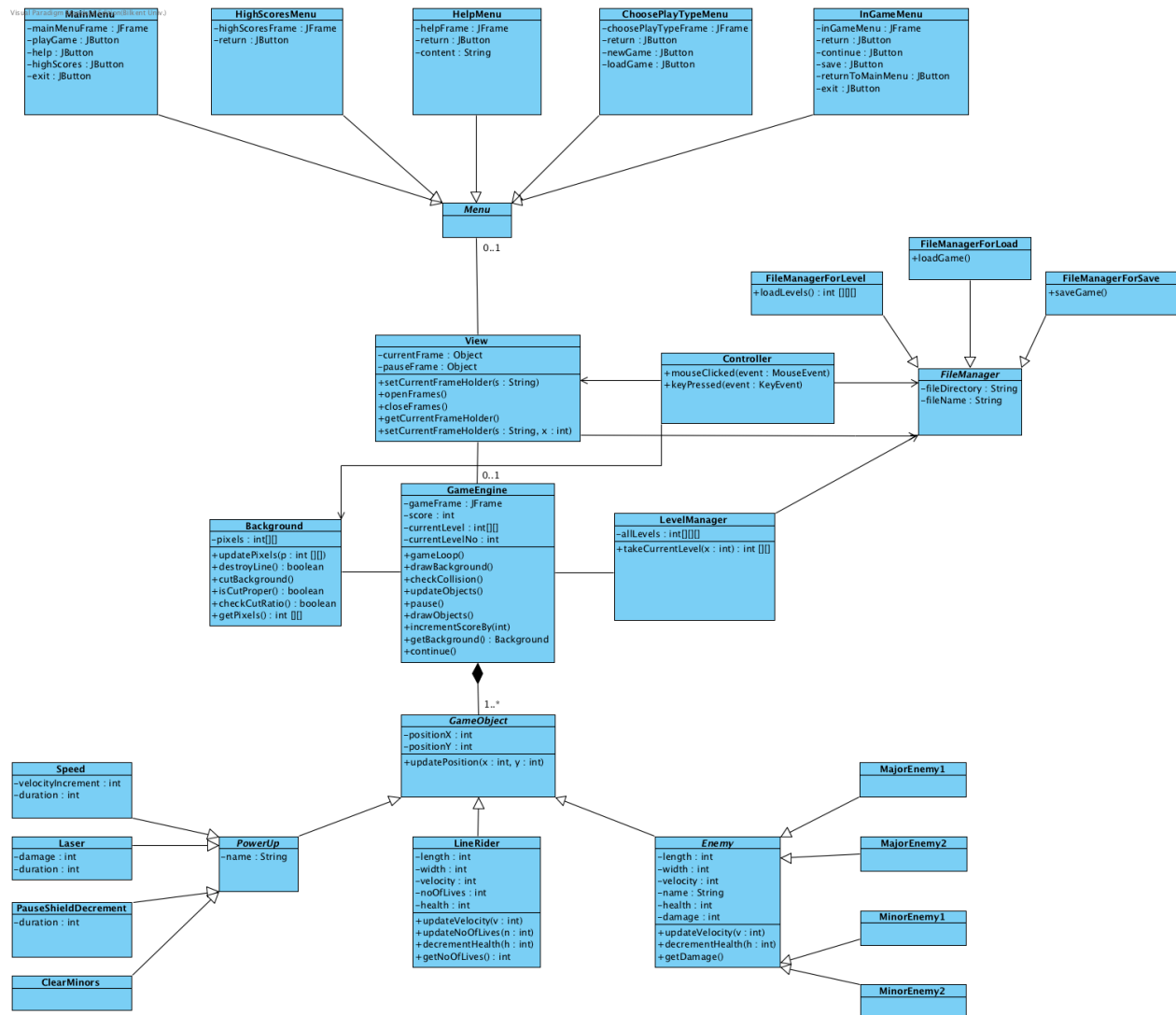Figure 22. Class diagram

Class diagram of "Split-Field" is illustrated above the system comprises of 26 classes whose functionalities are as follows:

Menu is an abstract class which consists of general menu attributes and operations. The following classes extend this class for carrying on necessary menu operations and forming GUI of the menus.

· **MainMenu**

· **HighScoresMenu**

· **HelpMenu**

· **ChoosePlayTypeMenu**

· **InGameMenu**

View class is similar to the view of MVC design pattern but it does not draw models as it is done in MVC. View manages the objects that draw the required frames like menus and gameplay (that is done in GameEngine class). It opens, closes and switches the previously stated objects.

Controller class is similar to the controller of MVC design pattern. It holds one View instance and controls the menu or gameplay object, which are drawers of frames and are in View instance, with inputs from the player. For example, player opens the game and sees the main menu. When it clicks a button, "mouseEvent" function is called and required operation is done on the View instance (e.g. changing the current frame with "setCurrentFrameHolder" function of View class). Or, when View object holds a GameEngine object, player gives the input from the keyboard. "keyPressed" function of Controller class is called and this updates the two-dimensional integer array in the BackGround class. In other words, controller holds an View instance as the viewer and holds an Background instance to change the state of background of the game. Updated background is redrawn by the

GameEngine object of the View class. In addition, Controller class will save the last level played by the player if player pauses the game and clicks on the save button.

FileManager is an abstract class which consists of general file manager attributes and operations. The following classes extend this class for carrying on necessary file management operations.

- **FileManagerForLevel:** This class is used by GameEngine class to load existing levels from "Levels.txt" file which holds many levels with their content (e.g. types of enemies, number of enemy for each type).

- **FileManagerForSaveLoad:** This class is used by the Controller class when player pauses the game and clicks on the save button in in-game menu. "saveGame" function saves the last level played by the player in "LastSavedLevel.txt" file.
  The other function of this class, "loadGame" is called when player clicks on the load game button in menu where the player chooses playing new game or continuing game from last saver's level.

Background class is used by GameEngine class to locate player in gameplay as LineRider object and other game objects. Gamefield is represented as two-dimensional integer array in this class. Different integers represent the different game objects, the split area and borders. Two-dimensional integer array is updated by Controller class when player directs LineRider from keyboard. Updating two-dimensional integer array is done with "updatePixels" function. GameEngine uses "isCutProper" function to control the line drawn by LineRider is proper for cutting. Then it calls "cutBackground" to split cut area from background. If an enemy touches the line drawn by LineRider when player has not finished drawing, GameEngine calls the "destroyLine" function. Finally, being successful at level is checked by GameEngine via "checkCutRatio" function.

GameEngine class is the most important class of the game in which all the game objects are declared according to the information of levels taken from the "Levels.txt" with the function "takeCurrentLevel" of LevelManager class. Graphics are drawn and redrawn with "drawObjects" and "drawBackground" functions according to the information taken from game objects and background. States of game objects are updated in this class with "updateObjects" function in which update functions of all game objects are called. It also, conducts the game in a loop called "gameLoop" that is basically makes the game go on as long as end conditions, which are cutting backgrounds of all levels successfully or killing certain time during the game, are not reached.

LevelManager class is used by GameEngine to obtain the level informations according to the current level. For example, when player pass the Level 2, GameEngine calls the "takeCurrentLevel" function of LevelManager which returns a two-dimensional integer array.

GameObject abstract class represents all of the items seen by the player. It holds the coordinates of the items as the x-position and y-position. Update of this coordinates are done in the "gameLoop" function of the GameEngine class.

- **LineRider** class
- **PowerUp** abstract class
- **Enemy** abstract class

inherit and expand the GameObject class. Required functions and attributes are added to these classes.

- **Speed**

- **Laser**

- **PauseShieldDecrement**

- **ClearMinors**

classes inherit the PowerUp abstract class. They represent the power ups of the LineRider in the gameplay.

- **MinorEnemy1**

- **MinorEnemy2**

- **MajorEnemy1**

- **MajorEnemy2**

classes inherit the Enemy abstract class. They represent the enemies in the gameplay. They are different from each other with their appearance, velocity, health and damage.

3.2. Dynamic Models
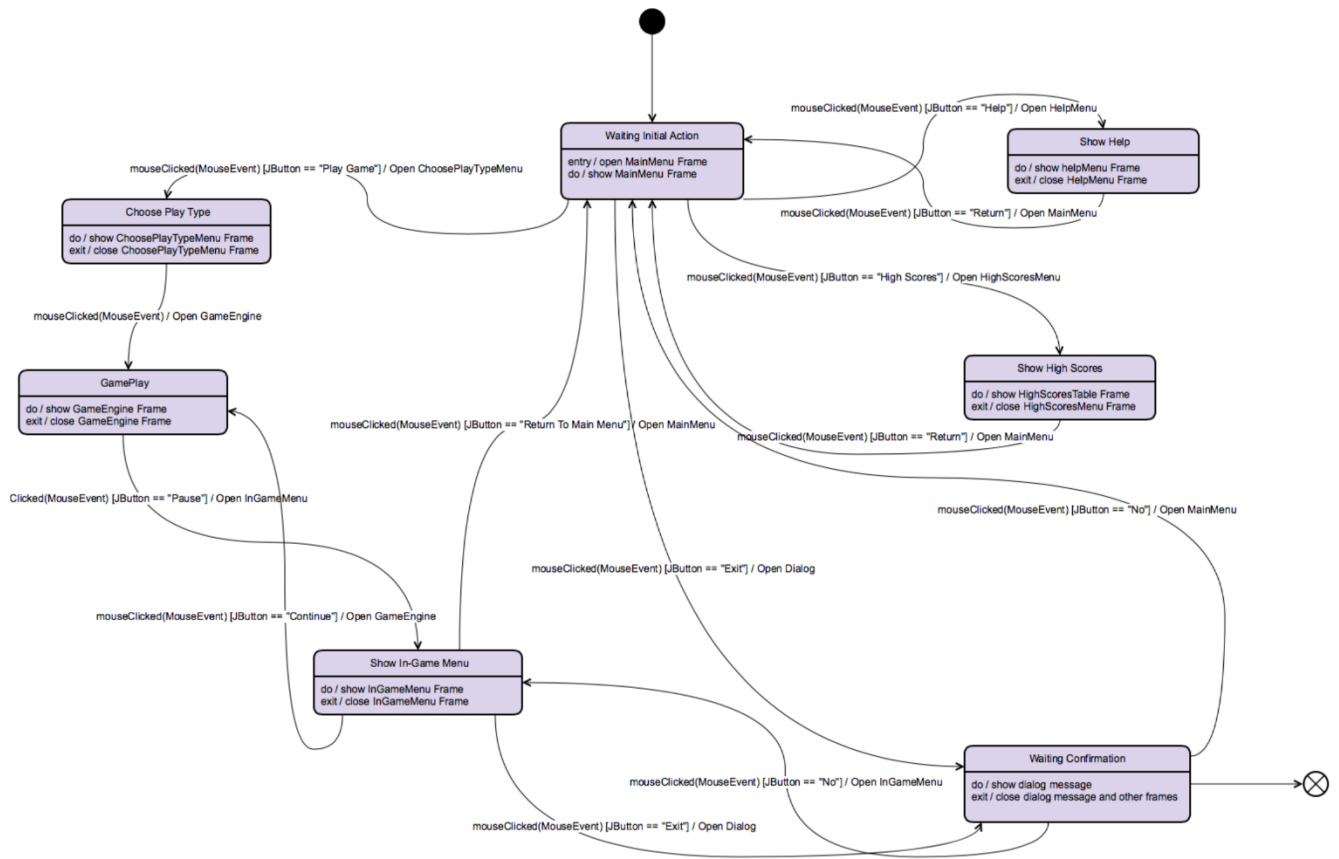### 3.2.1. State Chart

Figure 23. State chart diagram

3.2.2. Sequence Diagram

After the player runs the game, the main menu will appear on the screen and when the player clicked on the "Play Game" button, two options which are "New Game" and "Load Game" will appear on a new frame. If the player chooses to play new game the below scenario will be applied.

3.2.2.1. New Game

Scenario: The player chooses the "New Game" from play menu. After that, the mouseClicked() function of Controller object will be called and according to the button clicked on by the player, setCurrentFrameHolder() function of View object of Controller will be called with a parameter which indicates the choice of player which will be "New Game" in this context. setCurrentFrameHolder() will first call the closeFrames() function of View class then initialize a new GameEngine object and assign it as its currentFrame. The constructor of GameEngine will first assign the integer parameter to "currentLevelNo" variable which indicates the level, which will be 1 in this context because player chose "New Game", player wants to start, then use takeCurrentLevel() function of LevelManager object of itself. LevelManager will have already had all levels stored as three-dimensional integer array. takeCurrentLevel will return the allLevels[currentLevel][][] array. Then GameEngine constructor will declare objects with level information taken from the two-dimensional array. A Background object will be declared in the GameEngine constructor. Created GameObject objects will be located on the background with the updatePixels() function of Background object of GameEngine. Controller calls the getBackground() function of GameEngine object which will be taken with the getCurrentFrameHolder() function of View object of Controller. Gameplay will be ready for player. Controller takes the keyboard input from the user with

keyPressed() function and updates the location of player(or LineRider) in the two-dimensional integer array with the updatePixels() of Background object of Controller. gameLoop() function will go on until the end conditions will be reached. This function will call the drawObjects() and drawBackground() functions of GameEngine with the help of two-dimensional integer array taken from the Background object. GameEngine will control the appropriateness of cuts with isCutProper() function of Background object and if it is proper, GameEngine will call the cutBackground() function of Background. Also, in the same function, collision of enemy and player will be checked with checkCollision() function of itself. If a collision occurs, GameEngine will decrement player's health with decrementHealth() function of LineRider. In each turn of game loop, checkCutRatio() of the Background will be called to continue to loop(or game). This scenario's sequence diagram can be seen in Figure 5.

Figure 24. New game sequence diagram

3.2.2.2. Load Game

Scenario: The player chooses the "Load Game" from play menu. After that, the mouseClicked() function of Controller object will be called and because clicked on button will be "Load Game", Controller will call the loadGame() function of FileManagerForSaveLoad object to get the last saved level. setCurrentFrameHolder() function of View object of Controller will be called with two parameters which indicate the choice of player, which will be "Load Game" and the level desired to start in this context. setCurrentFrameHolder() will first call the closeFrames function of View class then initialize a new GameEngine object and assign it as its currentFrame. The constructor of GameEngine will first assign the integer parameter to "currentLevelNo" variable which indicates the level player wants to start, then use takeCurrentLevel() function with an integer parameter which will indicate the desired number of level to take from LevelManager object. LevelManager will have already had all levels stored as three-dimensional integer array. takeCurrentLevel() function will return the two-dimensional integer array. Then GameEngine constructor will declare objects with level information taken from the two-dimensional array. A Background object will be declared in the GameEngine constructor. Created GameObject objects will be located on the background with the updatePixels() function of Background object of GameEngine. Controller calls the getBackground() function of GameEngine object which will be taken with the getCurrentFrameHolder() function of View object of Controller. Gameplay will be ready for player. Controller takes the keyboard input from the user with keyPressed() function and updates the location of player(or LineRider) in the two-dimensional integer array with the updatePixels() of Background object of Controller. gameLoop() function will go on until the end conditions will be reached. This function will call the drawObjects() and drawBackground() functions of GameEngine with the help of two-dimensional integer array

taken from the Background object. GameEngine will control the appropriateness of cuts with isCutProper() function of Background object and if it is proper, GameEngine will call the cutBackground() function of Background. Also, in the same function, collision of enemy and player will be checked with checkCollision() function of itself. If a collision occurs, GameEngine will decrement player's health with decrementHealth() function of LineRider. In each turn of game loop, checkCutRatio() of the Background will be called to continue to loop(or game). This scenario's sequence diagram can be seen in Figure 5.

Figure 25. Load game sequence diagram

3.2.2.3.   Pause & Save Game

Scenario: As long as game loop continues to be executed and the player can control the flow of the game, the player can click on the pause button that will be located top right corner of the screen to interrupt flow of the game. When the button is clicked, the mouseClicked() function of Controller object will be called and because clicked on button will be "In-game menu". The setCurrentFrameHolder() function of the View object will set the current frame as inGameMenu frame. Then the gameEngine frame will be closed by closeFrames() function of the View object. Then the View object will call pause() function and the game will pause. The View object will initiate InGameMenu object and in-game menu which consists of 4 options, which are "Save", "Load", "Return to Main Menu" and "Exit", will appear on the screen. If the player wants to save his game, he will click on the "Save" button and the mouseClicked() function of Controller object will be called. Then the Controller object will call saveGame() function of the FileManagerForSave class. The saveGame() function will modify LastSavedLevel.txt file by writing the current level that the user is playing. When the game has been saved this function will display a "Successfully Saved" message on the screen and the player will continue to see in-game menu. If he clicks on the "Continue" button, the mouseClicked() function of Controller object will be called and the inGameMenu frame will be destroyed by closeFrames() function of View object, so in-game menu will disappear. Then by setCurrentFrameHolder() function of the View object, the current frame will be gameFrame and the player will continue to the game from where he left. If he clicks on the "Return to Main Menu" button, the mouseClicked() function of Controller object will be called and the current frame will be closed by closeFrame() function of the View object. Then by setCurrentFrameHolder() function of the View object, the current frame will be mainMenuFrame. Then the View object will initiate MainMenu object. If he clicks on the "Exit"

45

button, the mouseClicked() function of Controller object will be called and Controller object will call java.lang.System.exit() method that terminates the currently running JVM. This scenario's sequence diagram can be seen in Figure 6.
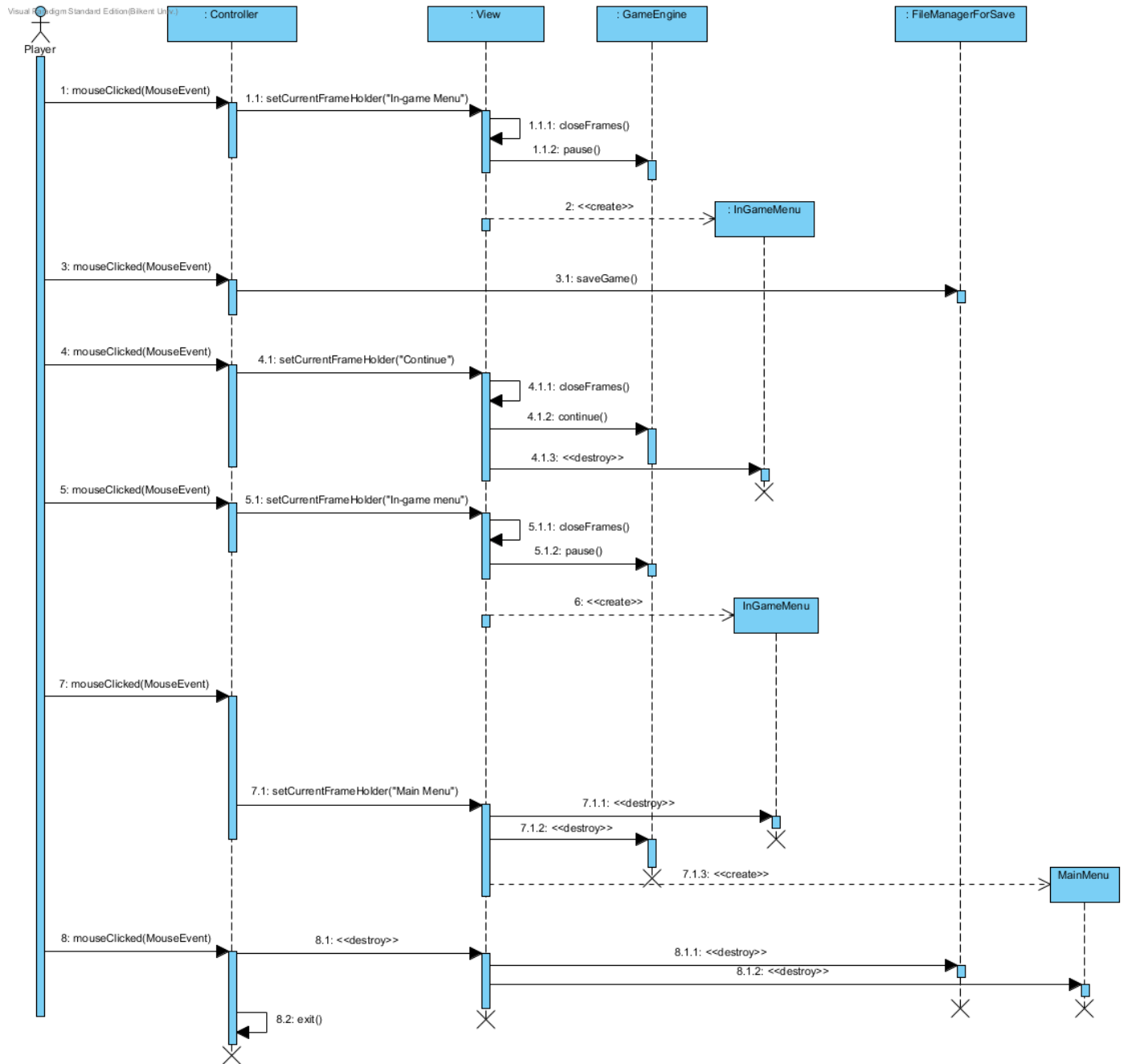
Figure 26. Pause & Save Game sequence diagram
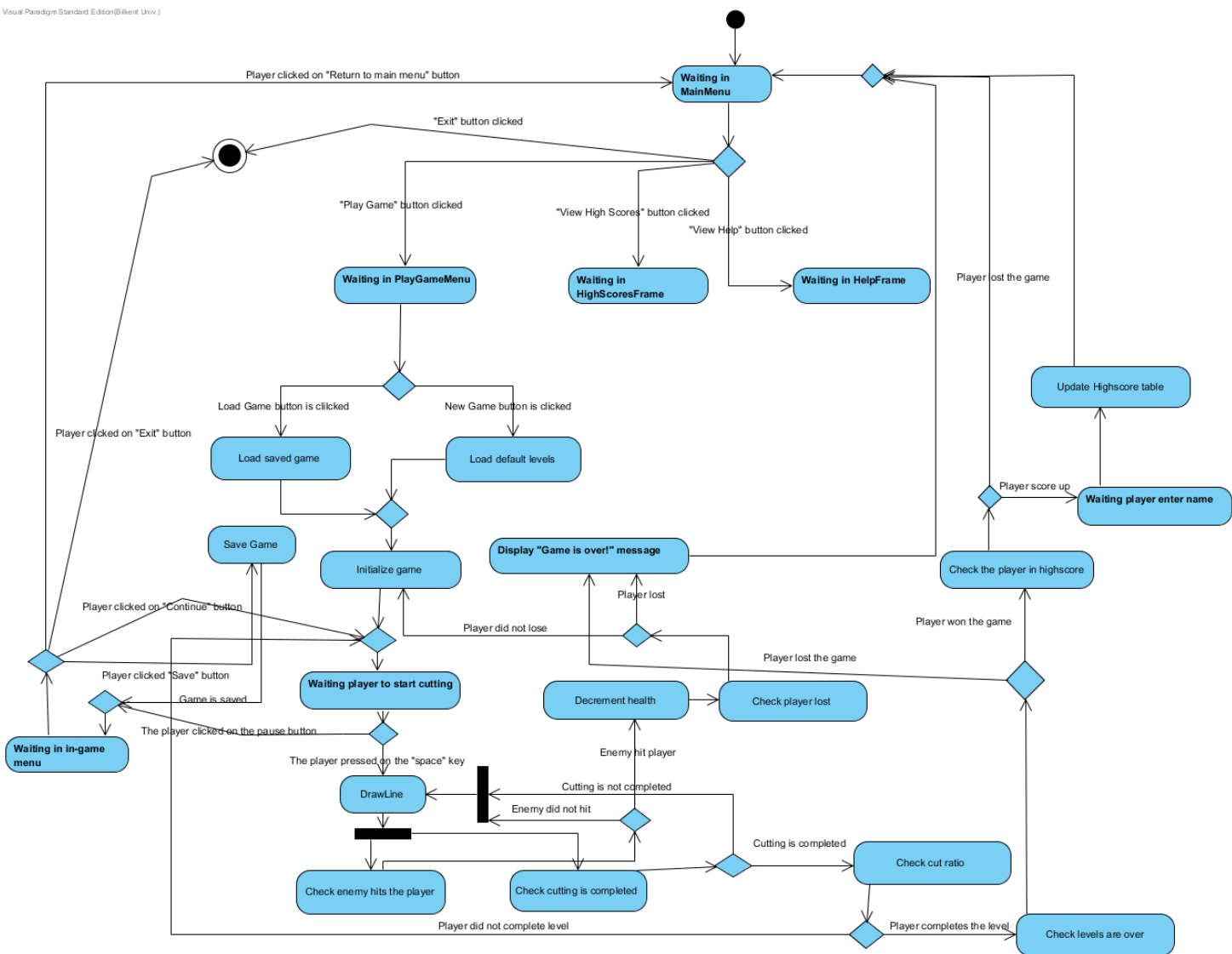
3.2.3. Activity Diagram

Figure 27. Activity Diagram

4. Conclusion

In this report, we introduce and analyzed our project the game "Split-Field" for CS319 course. Our report consists of the main parts which are requirements elicitation and system model design. The purpose of this analysis report is facilitate the implementation and design part of our project. With the guidance of this report, our implementation time and the possible bug number will decrease since we created possible scenarios that the player can role.

In the first part requirements elicitation, initially, we determined functional and non-functional requirements to identify the purpose of the system. Then we tried to examine all possible scenarios that player could perform during the lifecycle of the game which helped us to determine what is inside and what is outside of the system. After defining the system with requirements elicitation part, we started to determine system model design.

The second part of this report which is system model design consists of object models and dynamic models. This part contains the design of all classes and contents, methods and functions of these classes and the interactions and relations between these classes. The class diagram which contains classes, functions and relations between them will form the structure of implementation of our project, so it will reduce the possible problems that may occur because of lack of hierarchy and provide a way to reduce the complexity of the system by determining classes and their tasks.

Other than the class diagram, this report contains diagrams that visualize the design stage of the system such as use case diagrams, sequence diagrams, state chart diagrams and activity diagrams. These diagrams will help us to reduce the complexity of the system by separating the system to parts and analyzing these parts separately and also analyzing relations between these

separate parts. Thus this separate and conquer approach of diagrams will help us to understand the system effectively.

Finally, this report also contains the navigational path diagram and the user interface design of our project. We tried to keep the GUI of the project consistent and simple as possible to provide a usable and friendly interface that the player can have easy and familiar control on the game. While creating the navigational path diagram, we based on our use case diagram. Designing the user interface in this report will provide us to create a better user experience for the player as well as helping creation of distinction between lifecycles of the system.

To sum up, we acknowledged the fact that the analysis reports form the basis of projects by providing consistent structure for a solid software design. Thus, we tried to analyze this project as detailed as possible within this report with the expectation that this report will be guidelines for the implementation and design of the future steps of the project.

References:

[1]: http://www.page-online.nl/volfied/index.php

[2]: http://www.page-online.nl/volfied/index.php?c=thegame