CS 319 - Object-Oriented Software Engineering

System Design Report

Animal Uprising

<u>Group 3-A</u>

Bora Ecer

Ata Gün Öğün

Albjon Gjuzi

Tanay Akgül

1. Introduction

Animal Uprising is a different variation of the basic Castle Wars game, in which the main goal is destroying the enemy castle with the soldiers that the player summons with in-game currency, food. The game will be a desktop app and will be controlled with keyboard.

2. Overview

2.1 Game Play

Animal Uprising is a strategy/adventure game in which the player will be able to control a hero rather than having a castle. During the game, in order to destroy the enemy castle, the player will have to carve a path to the enemy castle by defeating the enemy soldiers. The player can summon ally soldiers in order to have a bigger striking force towards enemies soldiers. Once the player and allied soldiers reach to the enemy castle, they can attack it to tear it down. The player hero will be able to cast both defensive and offensive skills which either help the allies or harm the enemies. The game has five different difficulty levels and in each level, the player has to face more and harder types of enemy soldiers. Upon finishing each level, and destroying all enemies soldiers the player will gain coin which will allow the player upgrade skills or soldiers. The player will be able to pause the game and continue whenever they want to. Also, the player will be able to replay completed levels to gain more coins. There will also be an option to reset the whole game from scratch in order to set a higher score. The game will be played with the keyboard and mouse, specific buttons will be assigned for moving the hero, summoning different soldiers and casting skills. The mouse will be used only on the shop and the main screen.

2.2 Hero

The hero is the main character and the only character the player is able to control.. Since, the hero will be the one who is summoning soldiers and casting skills, it should have the in-game currency, food and mana. At first, when the game is initially started, both food and mana will be zero. Also, they will have upper limits so the player cannot reach high amounts of mana and food.

- HP: 1000
- Mana upper limit: 100
- Food upper limit: 150

2.3 Allies

There will be four different soldiers which can be summoned by the player. All of the soldiers will have a certain role, and in total there will be three different roles: melee, ranged and tank.

2.3.1 Melees

- Dog: the basic unit, it has low health and low damage but the summoning it requires less food.
  - HP: 75
  - Damage: 15
  - Food required to summon: 20
- Bear: compared to the dog, bear has more HP and more damage but summoning it requires more food.
  - HP: 150

- Damage: 50
- Food required to summon: 50

2.3.2 Ranged

- Monkey, is the only ranged unit of the game, it has less health but it offers ranged attack, so if the frontline is strong enough it can provide more damage.

  - HP: 50
  - Damage: 20
  - Food required to summon: 20

2.3.3 Tank

- Tortoise: the only tank unit, the tortoise does not offer any damage output but it can absorb the damage done by the enemies.

  - HP: 800
  - Damage: 0
  - Food required to summon: 100

2.4 Enemies

In the game, there will be three different types of enemy soldiers.

2.4.1 Infantry

Infantry is the basic enemy type which will be summoned by the enemy castle occasionally.

- HP: 100,
- Damage: 20

2.4.2 Knight

The knights is the leading officers of the infantries, compared to the Infantry, the Knight is much rarer, and strong. Together with the Infantries, they form the main army of the enemy.

- HP: 200
- Damage: 25

2.4.3 Crusader

The crusader is the most rare enemy type, they only emerge from the castle in high difficulty levels.

- HP: 400
- Damage: 50

2.5 Skills

The hero will have four different spells. Each spell can be casted if the player has enough mana required to cast. These spells will have two different types: Offensive spells and defensive spells.

2.5.1 Offensive Spells

- Raven Strike: The hero calls upon a pack of ravens to attack the enemies. It hits the first enemy that the ravens find.
  - Required Mana: 20
  - Range: 30px
  - Damage: 30
- Hail: the hero summons a hail of rocks to hit the enemies in a certain area
  - Mana: 40

  - Range: 30px
  - Radius: 20px
  - Damage: 20 to each enemy in the radius

2.5.2 Defensive Spells

- Heal: Heals the nearby allies, in a certain area centered at the hero.
  - Required Mana: 40
  - Radius: 20px
  - Heal amount: 50HP
- Speed Buff: Increases movement speed of the nearby allies, in a certain area centered at the hero.
  - Mana: 50
  - Radius: 20px
  - Unit Speed Increase: %20
  - Duration: 5 seconds

2.6 Upgrades

After completing a round successfully or defeating an enemy soldier, the player will gain certain amount of coins. With coins the player can choose to upgrade skills and soldiers.

2.6.1 Skills

- There will be some types of skills with which our hero can attack the enemy soldiers.
- Offensive Spells: Increase damage, decrease mana requirement amount.
- Defensive Spells: Increase heal or movement speed, decrease mana requirement.

2.6.2 Soldier Upgrades

The soldier upgrades will increase the damage done and the health of the soldier which is chosen to be upgraded.

3. Functional Requirements

3.1 Play Game

This is our main functional requirement since it consists of the main screen of the application which the player will be controlling the main hero and trying to conquer as many enemy castles as possible. Also, through the main menu, the player will be able to open the shop, so it is a sub functional requirement for Play Game.

3.1.1 Shop

The Shop will contain the detailed information about the soldiers and skills, the player will be able to upgrade these with using the in-game currency, gold.

3.2 How To Play

In case the player has never played this type of game, he/she can simply open how to play in order to understand the main idea of how the game is actually played, and what are its components. We are thinking of a video tutorial, but in case that will be very difficult to implement we will simply insert a mockup of the game at the moment it is being played.

3.3 Settings

Initially, the player will be provided a default game setting. From the settings menu, the player will be able to change them according to his own wishes. These settings will be:

- Sound Level
- Sound On/Off

3.4 Restart Level

The player can restart the level whenever he wants after pausing it for different reasons

3.5 Toggle Sound

The player can toggle the sound effects on or off depending on his preferences. The sound effects are activated at different stages of the game such as killing an enemy or winning/losing the round(level).

3.6 Toggle Music

The player can toggle the music sounds on or off depending on his preferences too. The music is simply to make the game less boring but it can be turned off at anytime from settings menu.

4. Nonfunctional Requirements

4.1 Performance

In order to provide a high-quality game play, we are planning to add extra features like, music and sound to our system which essentially will not decrease the performance of our game, but instead increase the satisfaction gained from the game. Also, during the implementation, we will try to find the most efficient way to solve the problems, in order to achieve a good performance rate.

4.2 High Quality Graphics

Since gameplay is the most important aspect of a video game, we will prioritize having high quality graphics. We will try to create a good and user-friendly interface which will have high quality graphics. Also, having a smooth gameplay also improves the gameplay, so we will try to work with high FPS rates.

4.3 Portability

We will try to make our game portable as possible so that the game can find players from different hardware/software environments.

5. System Models

5.1 Use Case Models

This section provides the main use case model of Animal Uprising game and detailed use case information.

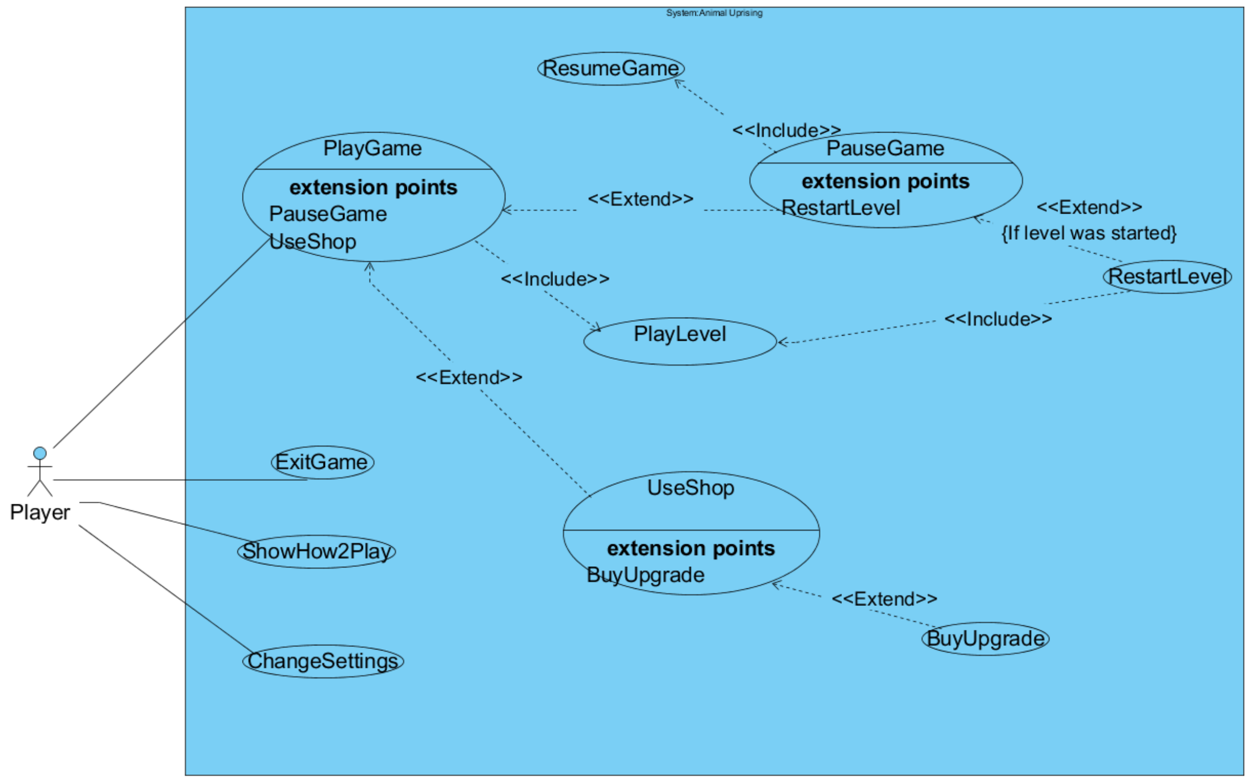


Figure 5.1.1 – The use-case model of Animal Uprising

5.1.1 Play Game

- Use case name: PlayGame
- Primary Actor: Player
- Entry Condition: Player has pressed the “PlayGame” button on the main menu.
- Exit Condition:
    - · Player chooses to go back to main menu.
- Main Flow Of Events:

    1. The system starts displaying the “game” scene (Level selection buttons, shop button etc.)
    2. Player chooses which level to play
    3. The selected level is played (Refer to “PlayLevel” use case)
    4. Repeat from step 1
- **Exceptions to Flow of Events**
    - · Player can pause the game (Refer to “PauseGame” use case)
    - · Player can go to shop (Before step 2) (Refer to “UseShop” use case)

5.1.2 PlayLevel

- use case name: PlayLevel
- Participating actors: Player
- Entry Condition: Player has selected a level to play on the “game” scene, or restarted a level
- Exit condition: The level has ended.
- Main flow of events:

1. The system loads the selected level and starts displaying the “level” scene (enemy castle, hero character etc.)

2. Hero object is updated by the system with accordance to the players input.

3. All other objects are updated by the system.

4. Repeat from step 2 until an exit condition is met.

5. Go back to play menu

5.1.3 PauseGame

- use case name: PauseGame
- participating actors: Player
- Triggering event(s): The pause key is pressed by the user during PlayGame
- Exit condition: Player presses the “Resume Game” button
- Main flow of events:

1.The system starts displaying the “pause” scene (“resume game button etc.”)

2.The player presses the “Resume Game” button

3. The system reverts the scene back to what it was before the pause.

Exceptions to Main Flow of Events:

Player can press the “restart level” button (Before step 2) if the game was paused during a level (Refer to “RestartLevel” use case)

5.1.4 RestartLevel

- use case name: RestartLevel
- participating actors: Player
- Triggering event(s): The “Restart Level” button is pressed, either on “pause” or “level lost” scene.
- Exit condition: The level is restarted.
- Main flow of events:

1. The system re-selects the level that was played.
2. The system invokes the PlayGame use case

5.1.5 UseShop

- use case name: UseShop
- participating actors: Player
- Triggering event(s): The “shop” button is pressed.
- Exit condition: The “back to level selection” button is pressed.
- Main flow of events:

1. The system starts displaying the “shop” scene(list of available upgrades, player coins etc.)
2. The player presses the “back to level selection”
3. The system finalizes UseShop

Alternative(s) to Main Flow of Events:

The player can buy an upgrade before step 2 (Refer to “BuyUpgrade” use case)

5.1.6 BuyUpgrade

- use case name: BuyUpgrade
- participating actors: Player
- Triggering event(s): an upgrade button is pressed in the “shop” scene.
- Exit condition:
- Main flow of events:

1. The system checks if the player has enough coins to buy the upgrade. The system finalizes the BuyUpgrade case on this step if player does not have enough coins.

2.The system reduces the amount of coins the player has and updates the upgrade list.

5.1.7 Change Settings

- Use Case Name: Change Settings
- Primary Actor: Player
- Entry Condition: Player should be in the main menu.
- Trigger evet: Player selects “Settings” from main menu
- Exit Condition: Player returns to the main menu.
- Main Flow of Events:
    1. Player selects “settings” to change or view game settings.
    2. The system displays game settings to the player.
    3. Player changes the settings according to his/her own choice.
    4. System updates the game settings.

- Alternative Flow of Events:
    - **a.** Player wants to return to the main menu.
        - **1)** Player pushes the “Back” button
        - **2)** Player returns to the main menu.

b. Player wants to restore the default settings.

   1) Player selects “Default Settings” to restore default settings.

   2) The system updates the game settings to default.

5.1.8 ShowHowToPlay

- Use Case Name: ShowHow2Play
- Primary Actor: Player
- Trigger event: Player clicks “How to Play” button from main menu.
- Entry Condition: Player should be in the main menu.
- Exit Condition: Player should return to the main menu.
- Main flow of events:
  - The system starts shoving the “how2play” scene.
  - Player clicks the back button.

    System returns to main menu.

5.2 Dynamic Models

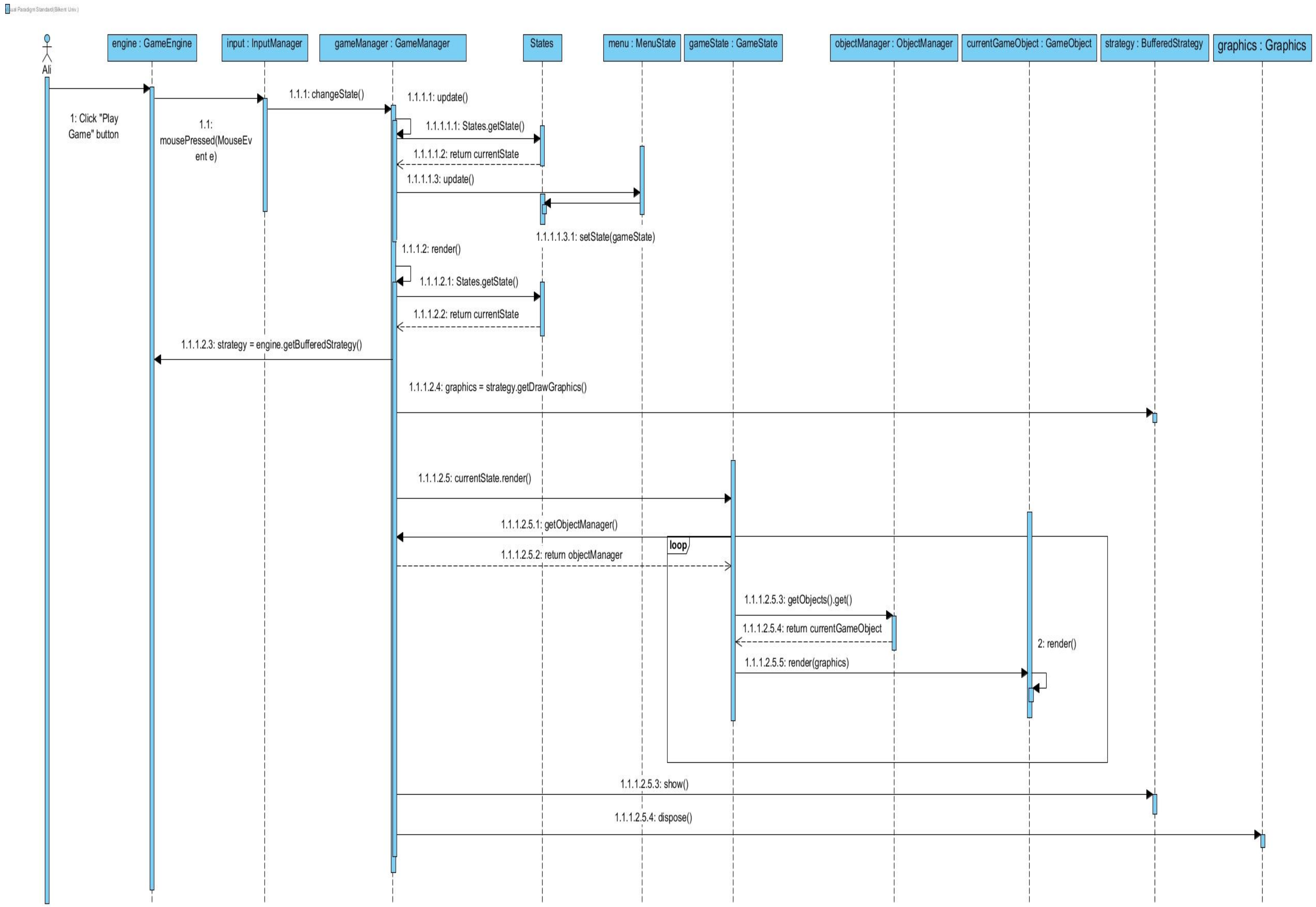
The following section covers the Dynamic Models of the game with respect to certain scenarios

5.2.1 Sequence Diagrams

5.1.1 Play Game

Scenario: Ali is in the main menu and he sees three buttons: Play Game, How To Play and Settings. He decides to play the game so he clicks the “Play Game” button, and he starts playing the game. Finally, he sees his character, and the enemy castle.

Description: Initially, the game starts with the initialization of Game Manager and its components: Input Manager, Object Manager and Sound Manager which are parameters of Game Manager object. After the initialization of the Game Manager, It first initializes the Game Engine, which has the components of the UI Components. The Game Manager also, loads the required object images through Image Manager. Then Game Manager initializes the states of the game: Game State, Menu State, How To Play State, Settings State and Pause State. Then as default, it sets the current state as Menu State. So, when this scenario comes to play, the main game loop is already running, and all of the states are ready to go.

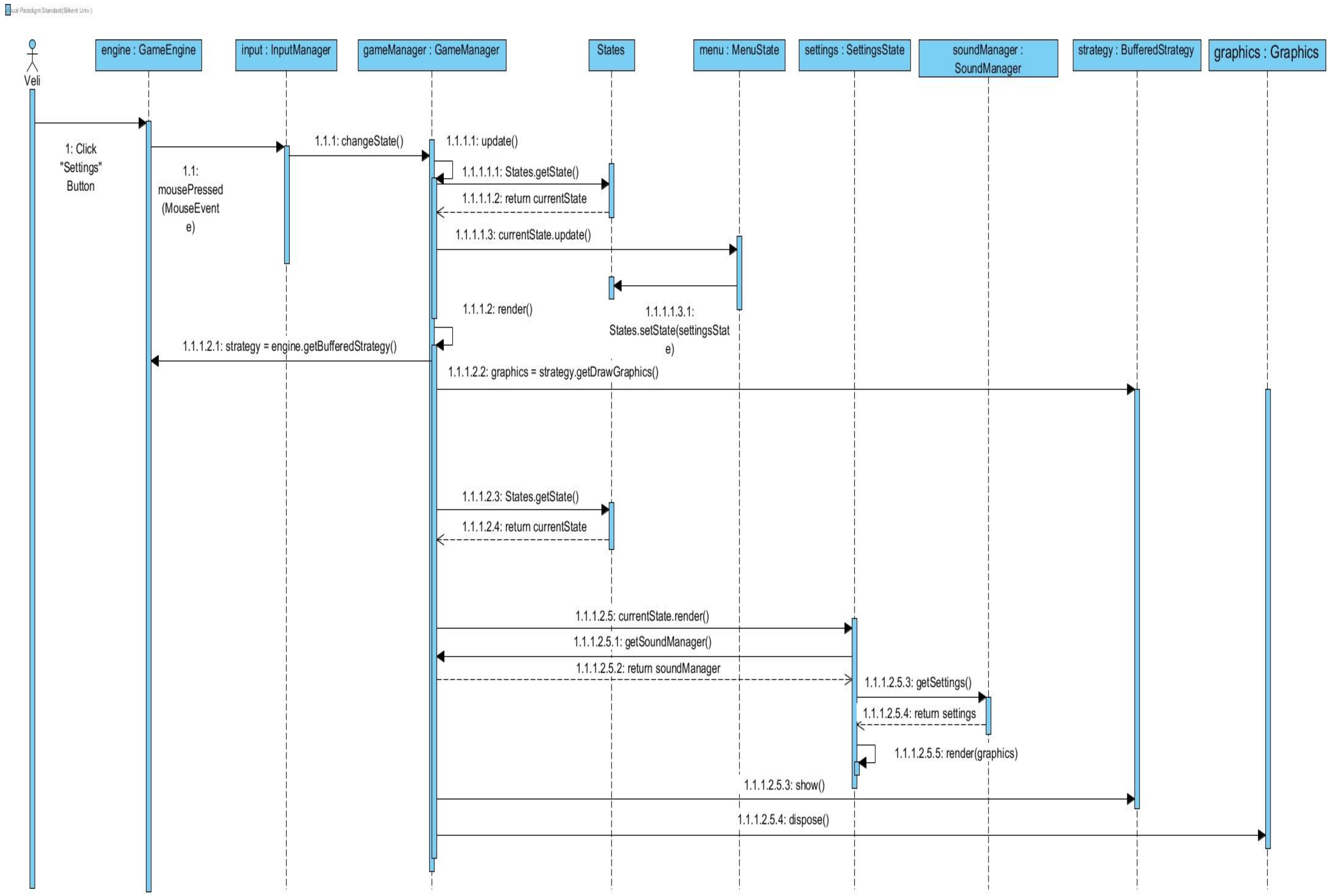


Visual Paradigm Standard(Bilkent Univ.)
Ali
engine : GameEngine
input : InputManager
gameManager : GameManager
States
menu : MenuState
gameState : GameState
objectManager : ObjectManager
currentGameObject : GameObject
strategy : BufferedStrategy
graphics : Graphics
1: Click "Play Game" button
1.1: mousePressed(MouseEvent e)
1.1.1: changeState()
1.1.1.1: update()
1.1.1.1.1: States.getState()
1.1.1.1.2: return currentState
1.1.1.1.3: update()
1.1.1.1.3.1: setState(gameState)
1.1.1.2: render()
1.1.1.2.1: States.getState()
1.1.1.2.2: return currentState
1.1.1.2.3: strategy = engine.getBufferedStrategy()
1.1.1.2.4: graphics = strategy.getDrawGraphics()
1.1.1.2.5: currentState.render()
1.1.1.2.5.1: getObjectManager()
1.1.1.2.5.2: return objectManager
loop
1.1.1.2.5.3: getObjects().get()
1.1.1.2.5.4: return currentGameObject
1.1.1.2.5.5: render(graphics)
2: render()
1.1.1.2.5.3: show()
1.1.1.2.5.4: dispose()

5.1.2 Settings

Scenario: Veli is in the main menu and he wants to see the settings of the game, so he first he clicks the “Settings” button to get into the Settings window in order to see the current settings of the game.

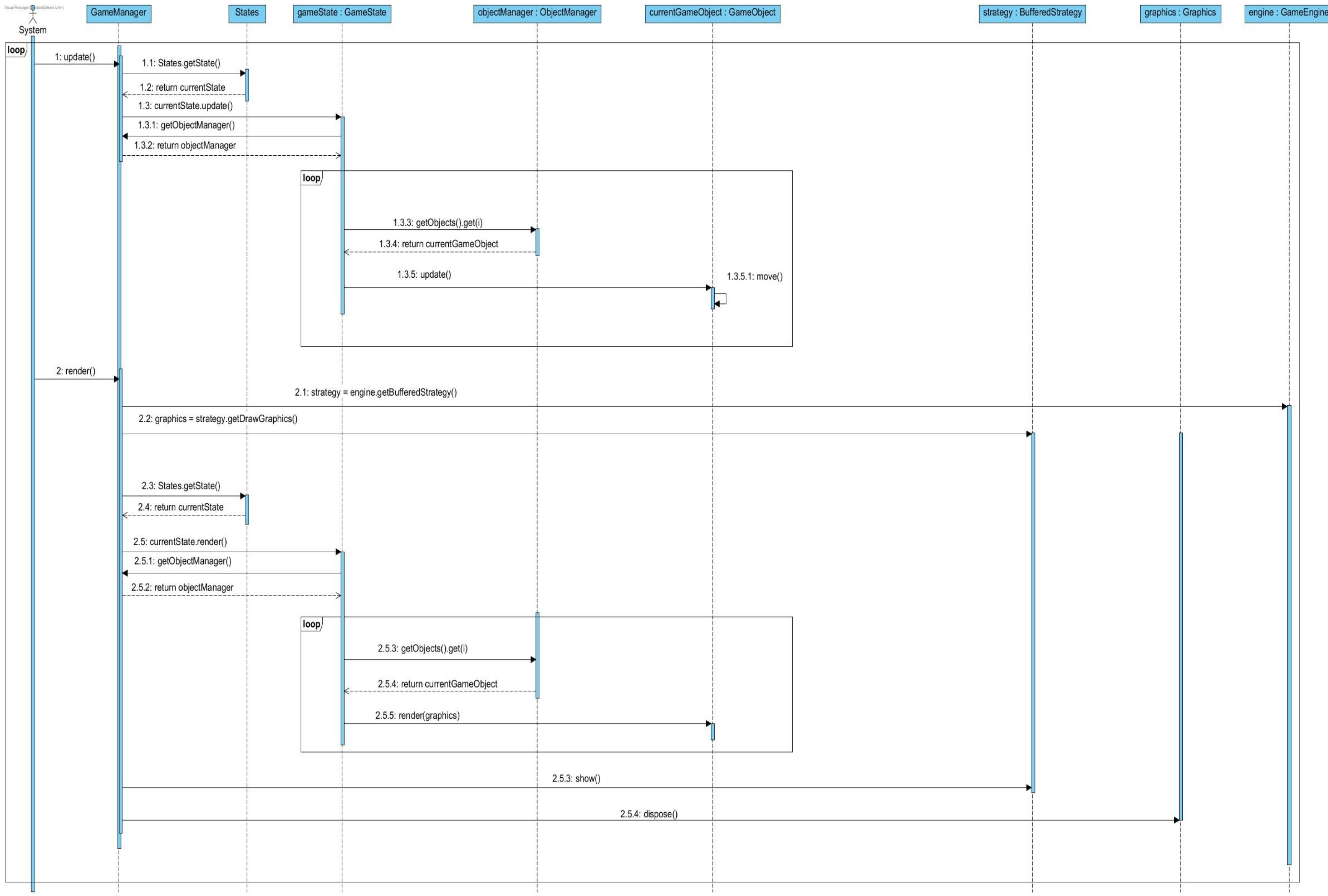
Description: Since the scenario starts with Veli on main menu, the current state of the game is Menu State. However, when Veli clicks on the “Settings” button, game changes its current state to Settings State. Once the state of the game has changed, the Game Engine adjusts to the new current state and draws the frame accordingly. Also, the Sound Manager is responsible for changing the music and sound options. So, when the options have been changed, the Sound Manager adjusts itself accordingly and the game continues with the new options.


Visual Paradigm Standard(Bilkent Univ.)
Veli
engine : GameEngine
input : InputManager
gameManager : GameManager
States
menu : MenuState
settings : SettingsState
soundManager : SoundManager
strategy : BufferedStrategy
graphics : Graphics
1: Click "Settings" Button
1.1: mousePressed (MouseEvent e)
1.1.1: changeState()
1.1.1.1: update()
1.1.1.1.1: States.getState()
1.1.1.1.2: return currentState
1.1.1.1.3: currentState.update()
1.1.1.1.3.1: States.setState(settingsState)
1.1.1.2: render()
1.1.1.2.1: strategy = engine.getBufferedStrategy()
1.1.1.2.2: graphics = strategy.getDrawGraphics()
1.1.1.2.3: States.getState()
1.1.1.2.4: return currentState
1.1.1.2.5: currentState.render()
1.1.1.2.5.1: getSoundManager()
1.1.1.2.5.2: return soundManager
1.1.1.2.5.3: getSettings()
1.1.1.2.5.4: return settings
1.1.1.2.5.5: render(graphics)
1.1.1.2.5.3: show()
1.1.1.2.5.4: dispose()

5.1.3 Update & Render Mechanism

Scenario: The system is updating each object's location, health and rendering them until the game state ends.

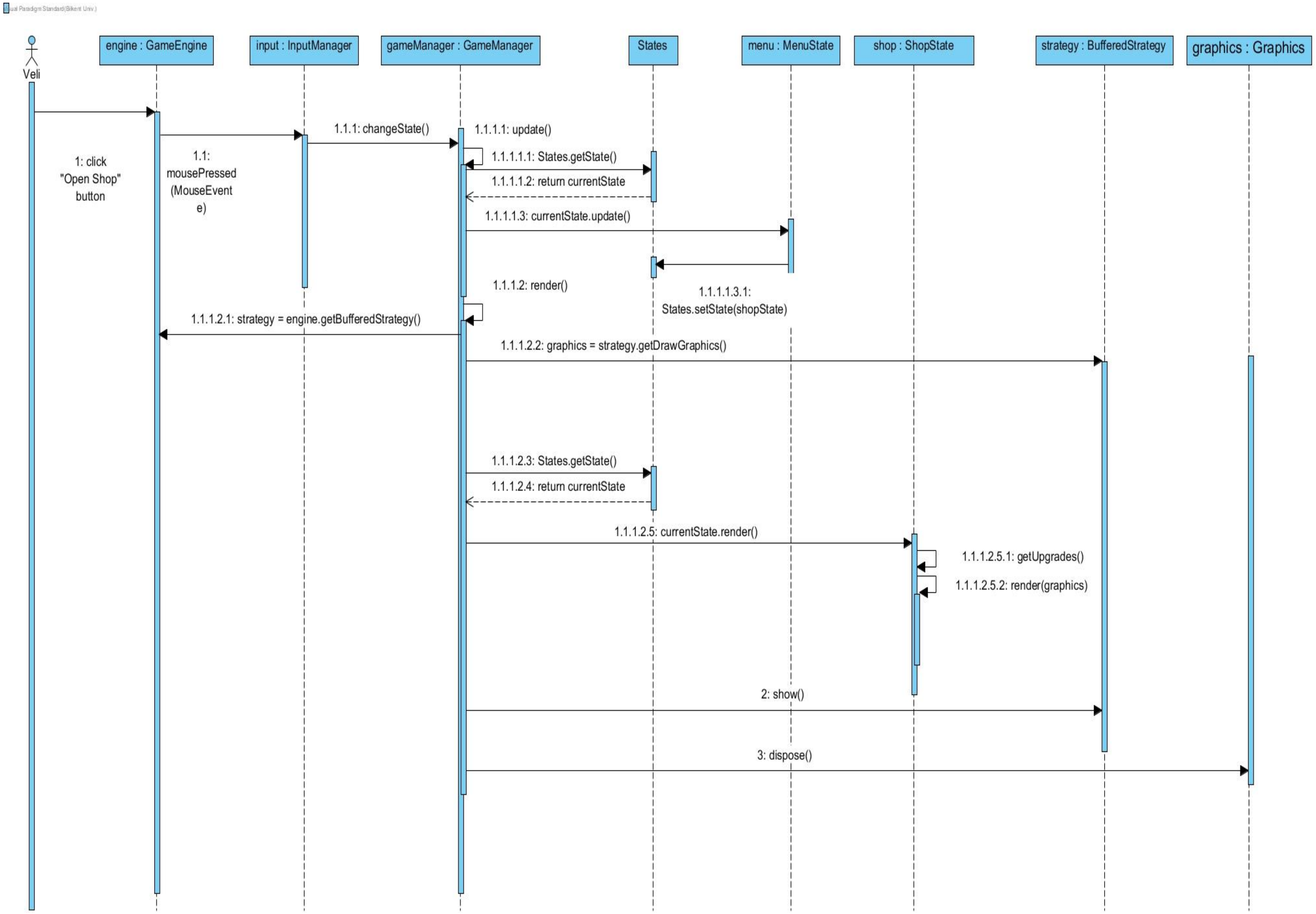
Description: The main game loop will constantly update the game objects and draw them to the screen, so the locations of each Game Object will be determined and updated automatically by the system, except the Hero Object. The movement of the hero object is dependent to the user input. However, the system should handle the update & render operations in the same manner.


System
GameManager
States
gameState : GameState
objectManager : ObjectManager
currentGameObject : GameObject
strategy : BufferedStrategy
graphics : Graphics
engine : GameEngine
loop
1: update()
1.1: States.getState()
1.2: return currentState
1.3: currentState.update()
1.3.1: getObjectManager()
1.3.2: return objectManager
loop
1.3.3: getObjects().get(i)
1.3.4: return currentGameObject
1.3.5: update()
1.3.5.1: move()
2: render()
2.1: strategy = engine.getBufferedStrategy()
2.2: graphics = strategy.getDrawGraphics()
2.3: States.getState()
2.4: return currentState
2.5: currentState.render()
2.5.1: getObjectManager()
2.5.2: return objectManager
loop
2.5.3: getObjects().get(i)
2.5.4: return currentGameObject
2.5.5: render(graphics)
2.5.3: show()
2.5.4: dispose()

5.1.4 OpenShop

Scenario: Fatma is in the main menu and she sees the shop button. She decides to open the shop and see what are the possible upgrades

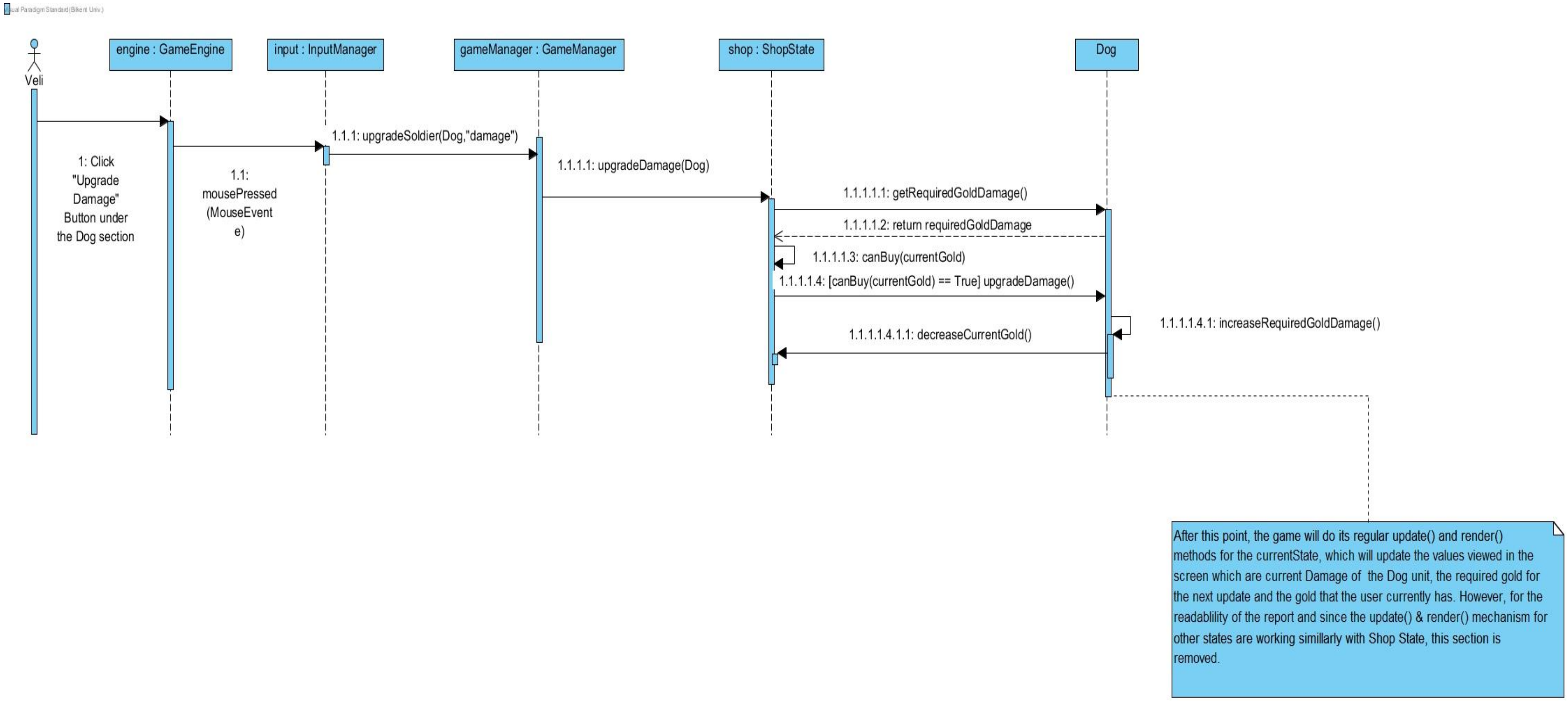
Description The current state of the game is Menu State since the scenario starts with Fatma being in main menu. When she decides to go to the Shop, the game changes its current state to Shop State. After this point, the Game Engine will be updated according to the Shop.


Visual Paradigm Standard(Bilkent Univ.)
Veli
engine : GameEngine
input : InputManager
gameManager : GameManager
States
menu : MenuState
shop : ShopState
strategy : BufferedStrategy
graphics : Graphics
1: click "Open Shop" button
1.1: mousePressed (MouseEvent e)
1.1.1: changeState()
1.1.1.1: update()
1.1.1.1.1: States.getState()
1.1.1.1.2: return currentState
1.1.1.1.3: currentState.update()
1.1.1.1.3.1: States.setState(shopState)
1.1.1.2: render()
1.1.1.2.1: strategy = engine.getBufferedStrategy()
1.1.1.2.2: graphics = strategy.getDrawGraphics()
1.1.1.2.3: States.getState()
1.1.1.2.4: return currentState
1.1.1.2.5: currentState.render()
1.1.1.2.5.1: getUpgrades()
1.1.1.2.5.2: render(graphics)
2: show()
3: dispose()

5.1.5 BuyUpgrade

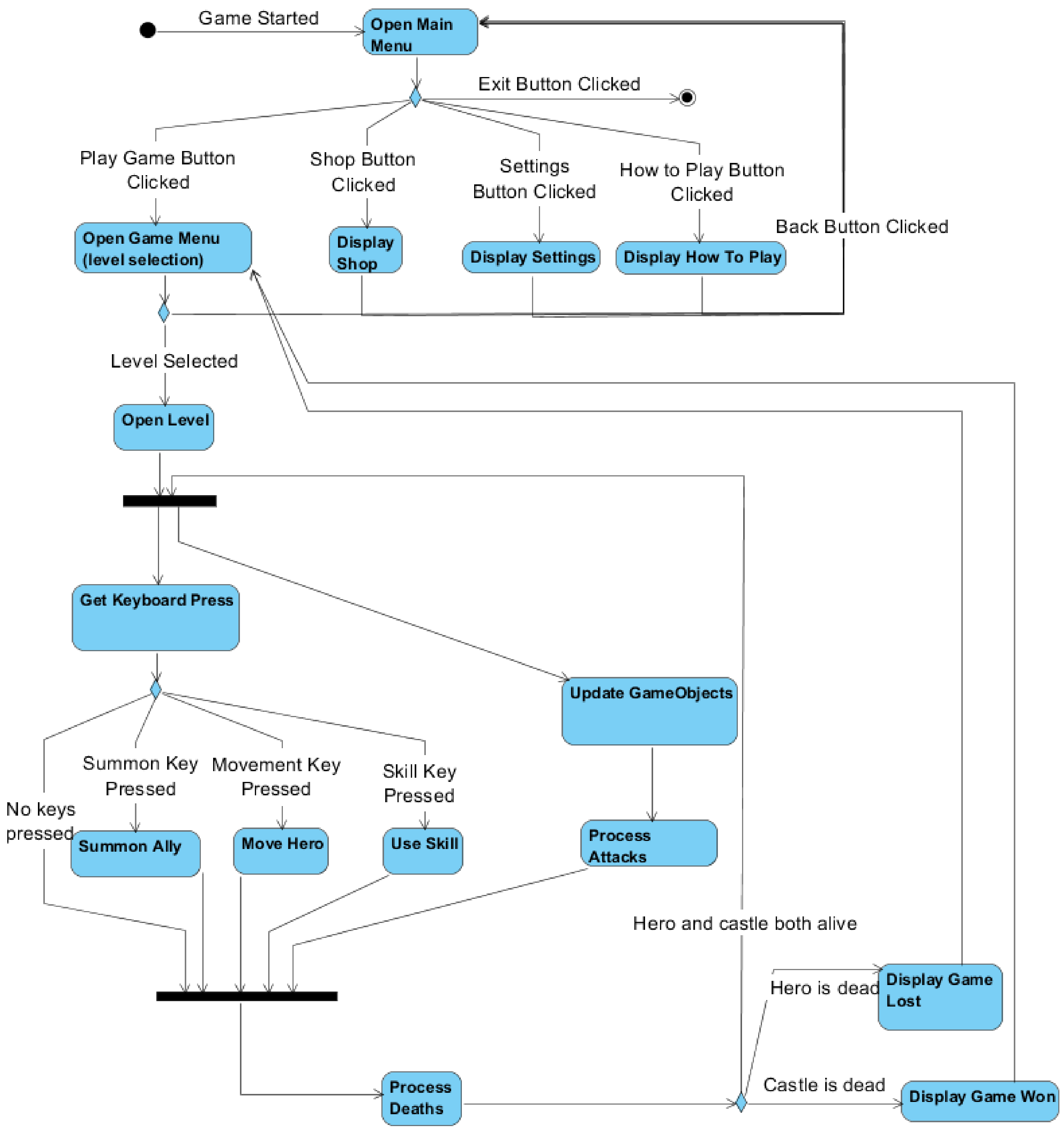
Scenario: Fatma is in the shop menu. She sees all of the upgradable soldiers, and upgradable skills of hero. Then she sees her current gold and immediately she checks what she can buy. Sadly, she can only buy damage upgrade to the Dog and she buys it.

Description: The game is currently at Shop State, so the game does not require to change the state but do operations within. Firstly, once the buy input is given, the Shop State will get the required gold to buy damage upgrade for the Dog, and if the player does have enough gold, it will increase base damage of the Dog, increase the gold required for the next upgrade and decrease the gold that the player currently has.


Visual Paradigm Standard(Bilkent Univ.)
Veli
engine : GameEngine
input : InputManager
gameManager : GameManager
shop : ShopState
Dog
1: Click "Upgrade Damage" Button under the Dog section
1.1: mousePressed (MouseEvent e)
1.1.1: upgradeSoldier(Dog,"damage")
1.1.1.1: upgradeDamage(Dog)
1.1.1.1.1: getRequiredGoldDamage()
1.1.1.1.2: return requiredGoldDamage
1.1.1.1.3: canBuy(currentGold)
1.1.1.1.4: [canBuy(currentGold) == True] upgradeDamage()
1.1.1.1.4.1: increaseRequiredGoldDamage()
1.1.1.1.4.1.1: decreaseCurrentGold()

After this point, the game will do its regular update() and render() methods for the currentState, which will update the values viewed in the screen which are current Damage of the Dog unit, the required gold for the next update and the gold that the user currently has. However, for the readablility of the report and since the update() & render() mechanism for other states are working simillarly with Shop State, this section is removed.

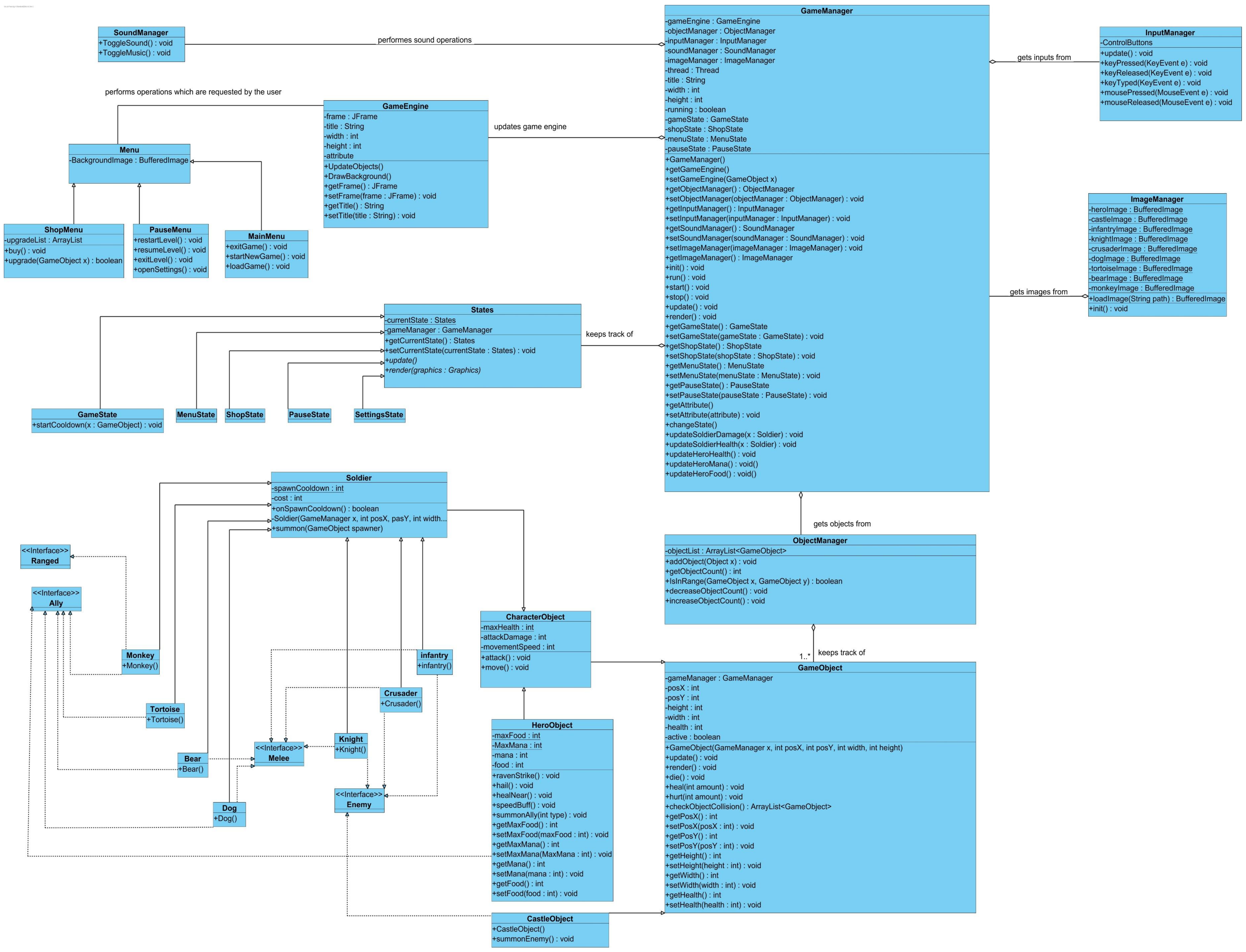
5.2.2 Activity Diagram



5.2.2.1 Activity Diagram Description

Everything starts when the user runs the game. The game opens and displays the main menu. Once one of the 5 buttons in that main menu is pressed, an action is taken according to the which button is pressed. The “Shop” button opens and displays the shop, the “Settings” button opens and displays the settings menu, the “How to Play” button displays the how to play screen, the “Play Game” button opens and displays the level selection menu and the “Exit” button exits the game. All these displays have a back button that goes back to the main menu when pressed. Once a level is selected in the level selection menu, the selected level is loaded and opened. The level is a loop that checks the keyboard presses, updates the GameObjects and processes the attacks. Different keyboard presses affect the game in different ways. The loop is terminated when either the hero or the castle is dead. Once the loop is over, the appropriate level-end screen is shown(“Level Won” or “Level Lost”). Once a level is over, the player is returned back to the level selection menu.

5.3 Object and Class Model



SoundManager
+ToggleSound() : void
+ToggleMusic() : void
performes sound operations
GameManager
-gameEngine : GameEngine
-objectManager : ObjectManager
-inputManager : InputManager
-soundManager : SoundManager
-imageManager : ImageManager
-thread : Thread
-title : String
-width : int
-height : int
-running : boolean
-gameState : GameState
-shopState : ShopState
-menuState : MenuState
-pauseState : PauseState
+GameManager()
+getGameEngine()
+setGameEngine(GameObject x)
+getObjectManager() : ObjectManager
+setObjectManager(objectManager : ObjectManager) : void
+getInputManager() : InputManager
+setInputManager(inputManager : InputManager) : void
+getSoundManager() : SoundManager
+setSoundManager(soundManager : SoundManager) : void
+setImageManager(imageManager : ImageManager) : void
+getImageManager() : ImageManager
+init() : void
+run() : void
+start() : void
+stop() : void
+update() : void
+render() : void
+getGameState() : GameState
+setGameState(gameState : GameState) : void
+getShopState() : ShopState
+setShopState(shopState : ShopState) : void
+getMenuState() : MenuState
+setMenuState(menuState : MenuState) : void
+getPauseState() : PauseState
+setPauseState(pauseState : PauseState) : void
+getAttribute()
+setAttribute(attribute) : void
+changeState()
+updateSoldierDamage(x : Soldier) : void
+updateSoldierHealth(x : Soldier) : void
+updateHeroHealth() : void
+updateHeroMana() : void()
+updateHeroFood() : void()
gets inputs from
InputManager
-ControlButtons
+update() : void
+keyPressed(KeyEvent e) : void
+keyReleased(KeyEvent e) : void
+keyTyped(KeyEvent e) : void
+mousePressed(MouseEvent e) : void
+mouseReleased(MouseEvent e) : void
performs operations which are requested by the user
GameEngine
-frame : JFrame
-title : String
-width : int
-height : int
-attribute
+UpdateObjects()
+DrawBackground()
+getFrame() : JFrame
+setFrame(frame : JFrame) : void
+getTitle() : String
+setTitle(title : String) : void
updates game engine
Menu
-BackgroundImage : BufferedImage
ShopMenu
-upgradeList : ArrayList
+buy() : void
+upgrade(GameObject x) : boolean
PauseMenu
+restartLevel() : void
+resumeLevel() : void
+exitLevel() : void
+openSettings() : void
MainMenu
+exitGame() : void
+startNewGame() : void
+loadGame() : void
ImageManager
-heroImage : BufferedImage
-castleImage : BufferedImage
-infantryImage : BufferedImage
-knightImage : BufferedImage
-crusaderImage : BufferedImage
-dogImage : BufferedImage
-tortoiseImage : BufferedImage
-bearImage : BufferedImage
-monkeyImage : BufferedImage
+loadImage(String path) : BufferedImage
+init() : void
gets images from
States
-currentState : States
-gameManager : GameManager
+getCurrentState() : States
+setCurrentState(currentState : States) : void
+update()
+render(graphics : Graphics)
keeps track of
GameState
+startCooldown(x : GameObject) : void
MenuState
ShopState
PauseState
SettingsState
Soldier
-spawnCooldown : int
-cost : int
+onSpawnCooldown() : boolean
-Soldier(GameManager x, int posX, pasY, int width...
+summon(GameObject spawner)
gets objects from
ObjectManager
-objectList : ArrayList<GameObject>
+addObject(Object x) : void
+getObjectCount() : int
+IsInRange(GameObject x, GameObject y) : boolean
+decreaseObjectCount() : void
+increaseObjectCount() : void
<<Interface>>
Ranged
<<Interface>>
Ally
CharacterObject
-maxHealth : int
-attackDamage : int
-movementSpeed : int
+attack() : void
+move() : void
1..*
keeps track of
GameObject
-gameManager : GameManager
-posX : int
-posY : int
-height : int
-width : int
-health : int
-active : boolean
+GameObject(GameManager x, int posX, int posY, int width, int height)
+update() : void
+render() : void
+die() : void
+heal(int amount) : void
+hurt(int amount) : void
+checkObjectCollision() : ArrayList<GameObject>
+getPosX() : int
+setPosX(posX : int) : void
+getPosY() : int
+setPosY(posY : int) : void
+getHeight() : int
+setHeight(height : int) : void
+getWidth() : int
+setWidth(width : int) : void
+getHealth() : int
+setHealth(health : int) : void
Monkey
+Monkey()
infantry
+infantry()
Crusader
+Crusader()
Tortoise
+Tortoise()
Knight
+Knight()
<<Interface>>
Melee
Bear
+Bear()
<<Interface>>
Enemy
Dog
+Dog()
HeroObject
-maxFood : int
-MaxMana : int
-mana : int
-food : int
+ravenStrike() : void
+hail() : void
+healNear() : void
+speedBuff() : void
+summonAlly(int type) : void
+getMaxFood() : int
+setMaxFood(maxFood : int) : void
+getMaxMana() : int
+setMaxMana(MaxMana : int) : void
+getMana() : int
+setMana(mana : int) : void
+getFood() : int
+setFood(food : int) : void
CastleObject
+CastleObject()
+summonEnemy() : void

GameManager class is the facede class of the system, the remaining control objects provide inputs for the GameManager and Game Manager decides which action will be taken.

InputManager: provides user input.

ImageManager: class to load images only once rather than loading them every time an object is created.

GameEngine: class that provides the view elements.

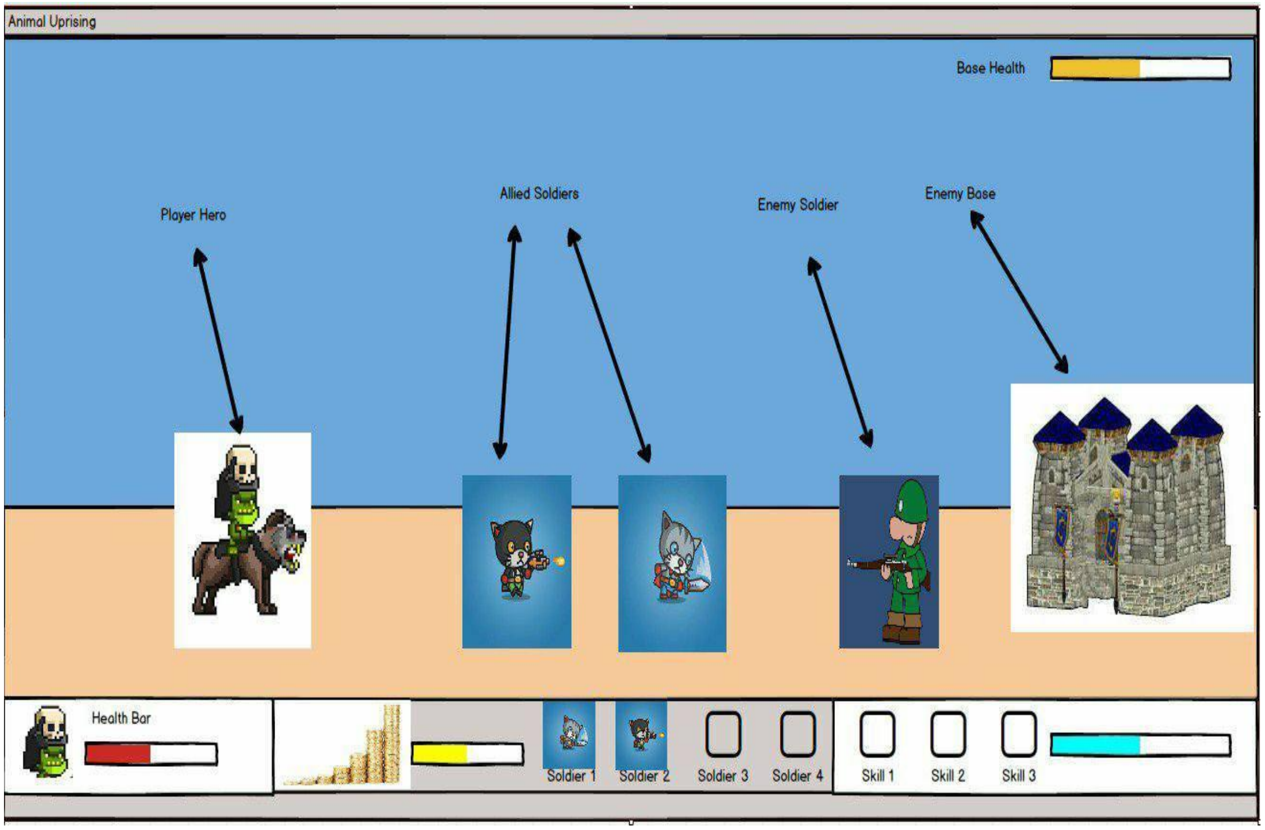
ObjectManager: class that keeps track of the GameObjects, essentially provides a list of game objects and whenever an object is added or removed, it is removed from the list. Also, whenever an update and render methods called for the game object, they require Object Manager to access the objects.

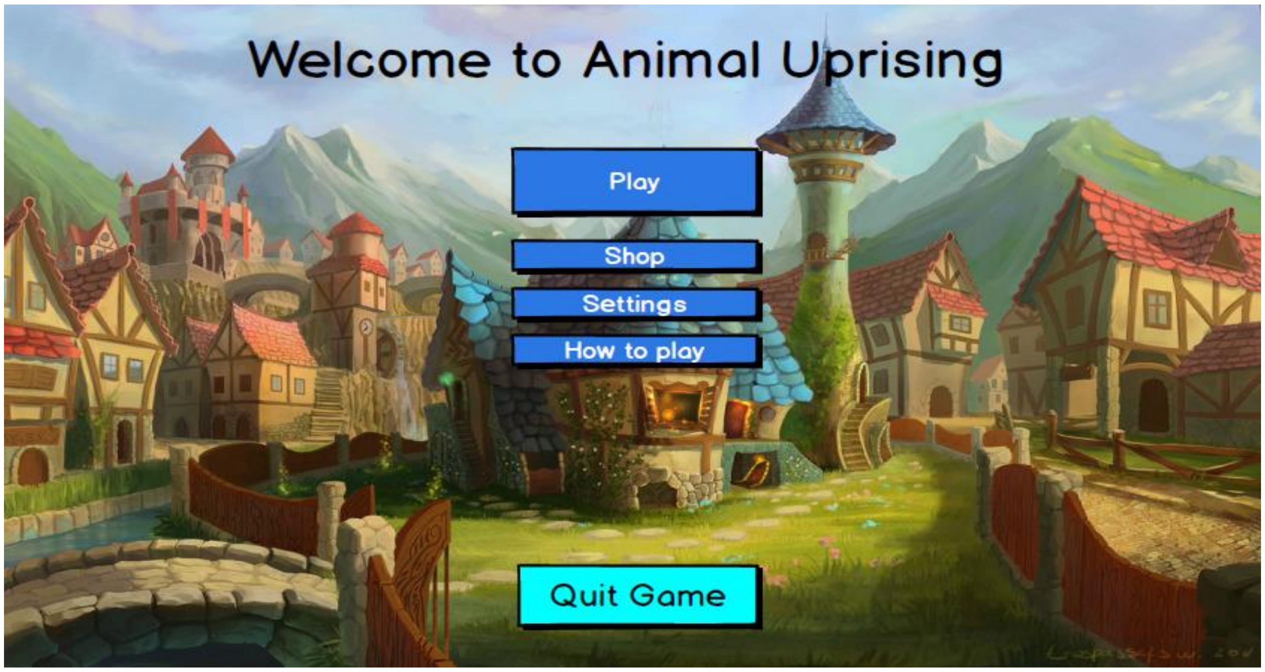
GameObject: The parent class of all game objects, it is an abstract class.

HeroObject: class which the attributes and methods of the hero is kept.

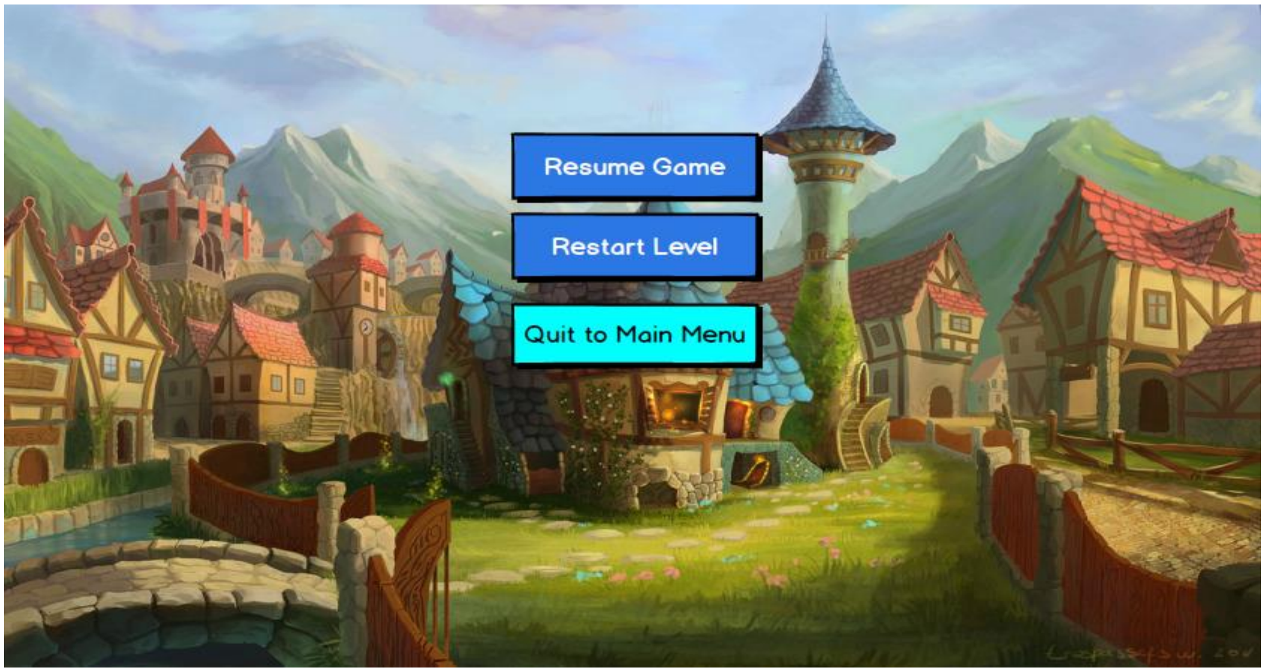
5.4 User interface – navigational paths and mock ups

5.4.1 Mockups:

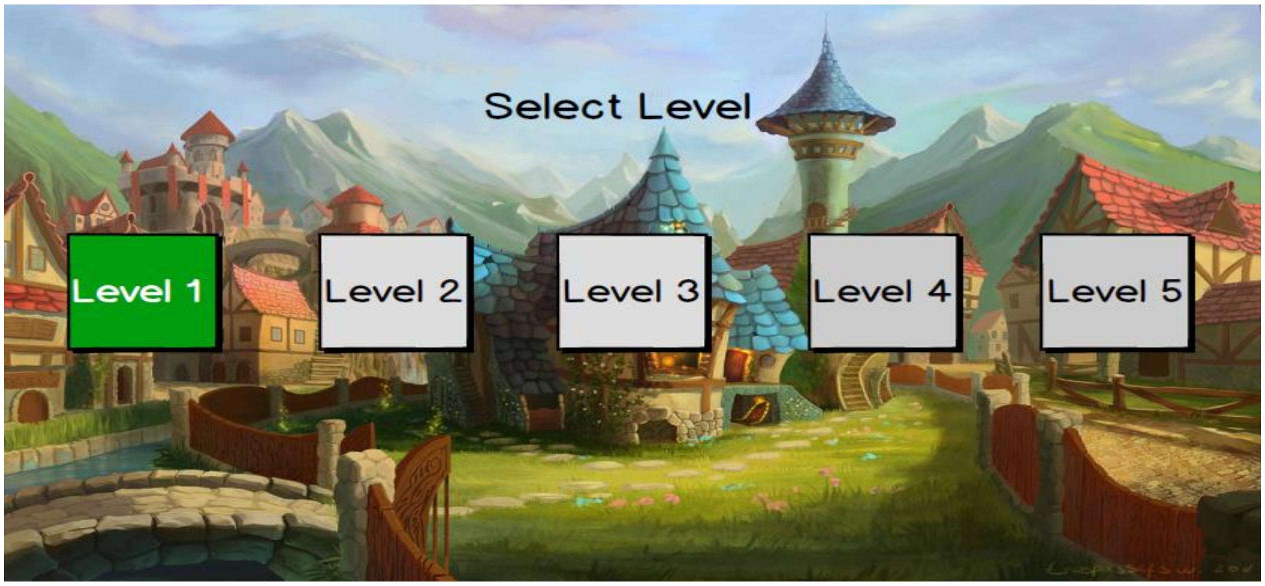




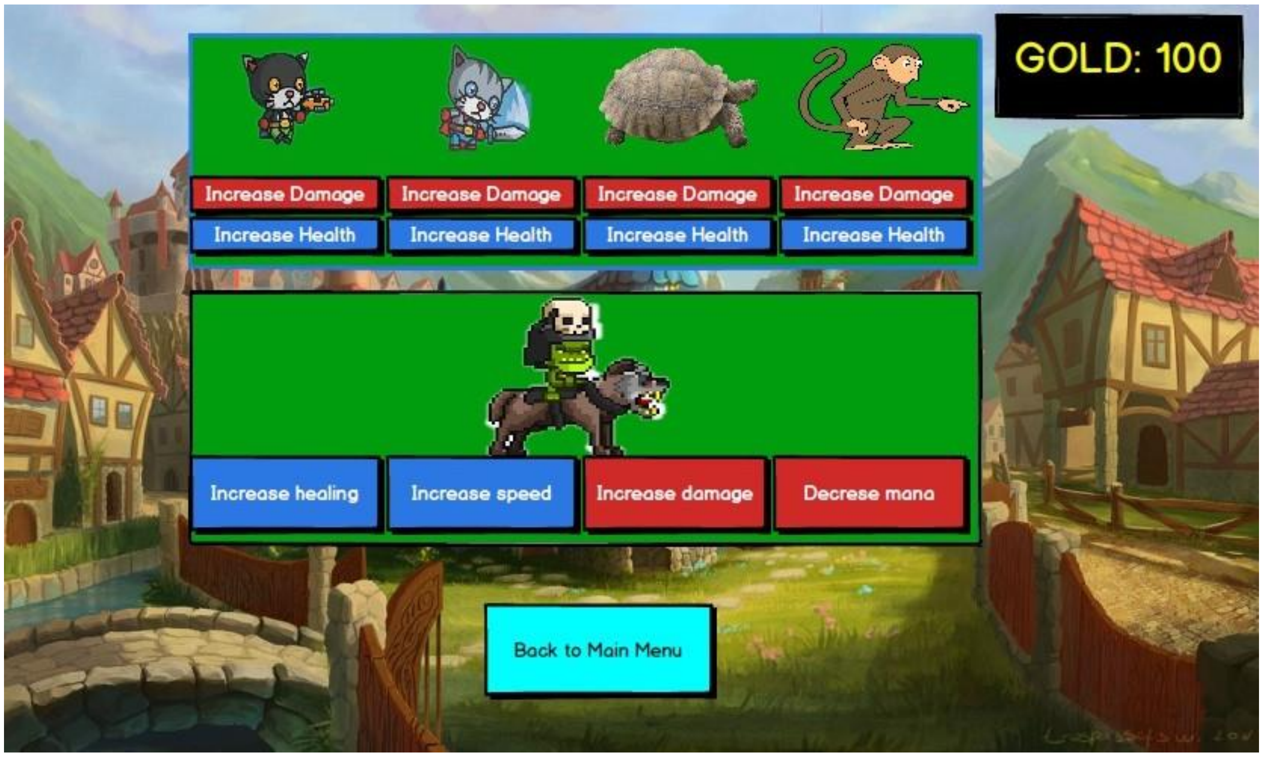
Main Menu: The Main Menu will be displayed as soon as the game is opened. The user can start to play by choosing Play button, it can go to Shop to update the hero or the allies, or it can view How to Play if it is the first time. The user can also change the settings by clicking Settings button or it can Quit the game.



Pause Menu: In the Pause Menu the user can either Resume playing the game, it can restart the level if he/she thinks they are not doing well, or they can return to the Main Menu by choosing Quit to Main Menu button.

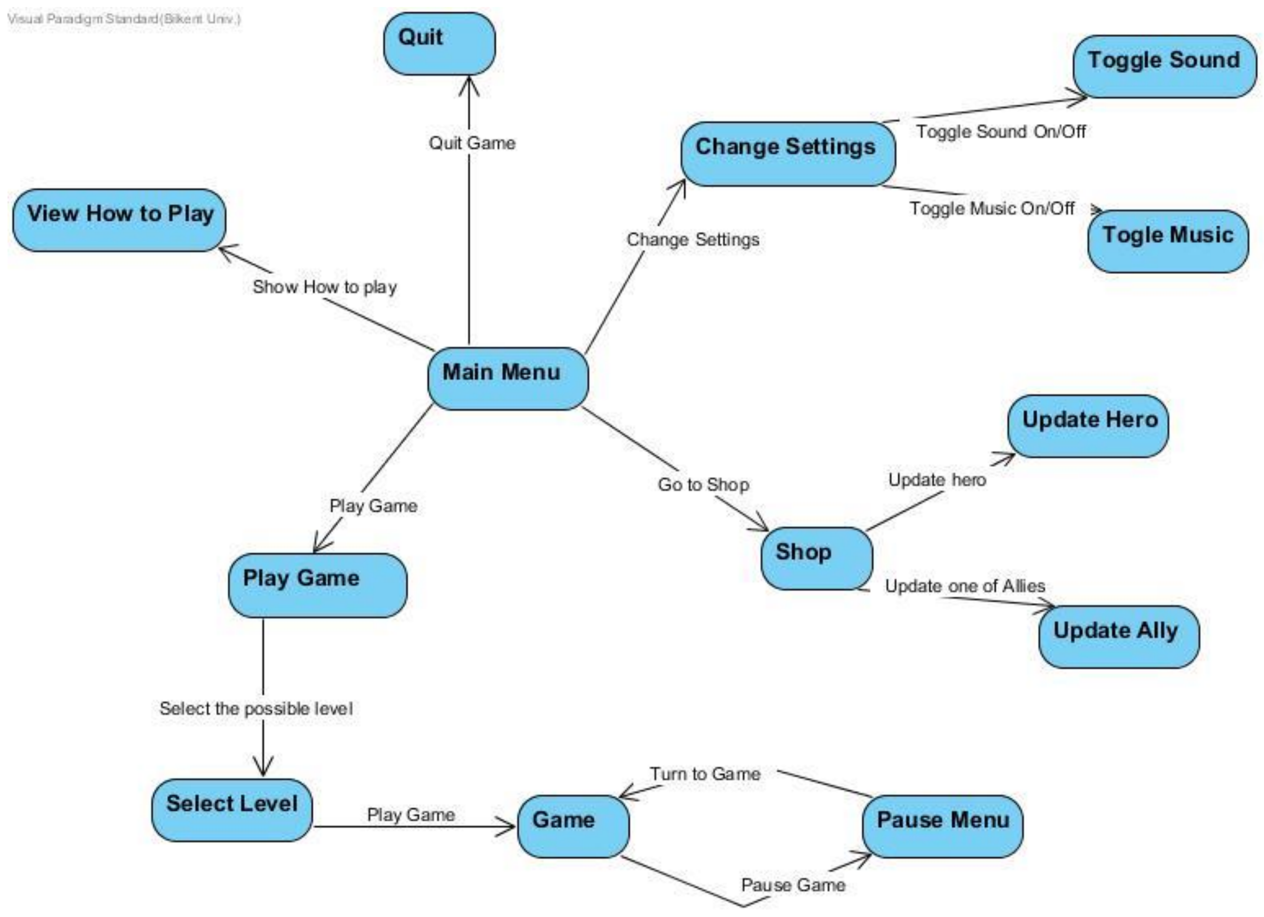


Levels: We will have 5 levels in our game, and this screen will be showed to the user as soon as he clicks Play button on the Menu Menu. Here the 5 levels will be shown, but only the unlocked ones can be chosen by the user.



Shop: Shop is not a Menu actually, but since it has lots of options to update the Hero or the Allies another mockup was needed for it.

5.4.1 Navigational Path



5 Conclusion

In this report, we have analyzed and refined our castle war game called Animal Uprising. There are 5 main categories in this report: Introduction, Overview, Functional requirements. Non-Functional requirements and System models. This report introduced and explained the core elements of our game: the characters and their details are defined. The functional and non-functional requirements are identified and the game is modeled using Use Case Diagrams, Sequence Diagrams, an Activity Diagram, and a Class Diagram. These diagrams can be referred during design process. The report also includes interface mockups to give the reader an idea about how things are going to look after the implementation. The contents in this report are subject to change during the design and implementation process.

6 What has changed?

From the first iteration, we have added sequence diagrams, activity diagram, navigational path and some new mock ups. Also, we designed a new use case model and updated the object & class model. The sequence diagrams were done with using the new use case model. As for the use case model, we changed the "shop" use case to "open shop" in order to provide a verbal activity. As for the object model, we added new classes, attributes and methods to our existing class diagram. Therefore, the class diagram became more detailed. A part from all these, we also changed the non-functional requirements and added performance, high quality graphics and portability to them. Also, another major change which is a topic of analysis, is that we removed the items from our game, and decided to work on the soldier and skills upgrades.

7 Glossary & Reference

1. Object-Oriented Software Engineering, Using UML, Patterns, and Java, 2nd Edition, by Bernd Bruegge and Allen H. Dutoit, Prentice-Hall, 2004, ISBN: 0-13-047110