SCHOOL OF ENGINEERING AND TECHNOLOGY

FINAL ASSESSMENT FOR THE BSC (HONS) INFORMATION TECHNOLOGY; BSC (HONS) COMPUTER SCIENCE; BACHELOR of SOFTWARE ENGINEERING (HONS)YEAR 2

ACADEMIC SESSION 2023; SEMESTER 3

PRG2104: OBJECT ORIENTED PROGRAMMING

Project: Digimon: The Chosen Memories
DEADLINE: Week 14

INSTRUCTIONS TO CANDIDATES

- This assignment will contribute 50% to your final grade.
- This is an individual assignment.

IMPORTANT

The University requires students to adhere to submission deadlines for any form of assessment. Penalties are applied in relation to unauthorized late submission of work.

- Coursework submitted after the deadline will be awarded 0 marks
-

Lecturer's Remark (Use additional sheet if required)

I Choon Han Wen Benjamin (Name) 22018345(std. ID) received the assignment and read the comments
BenjaminChoon 19/8/2023 (Signature/date)

Academic Honesty Acknowledgement

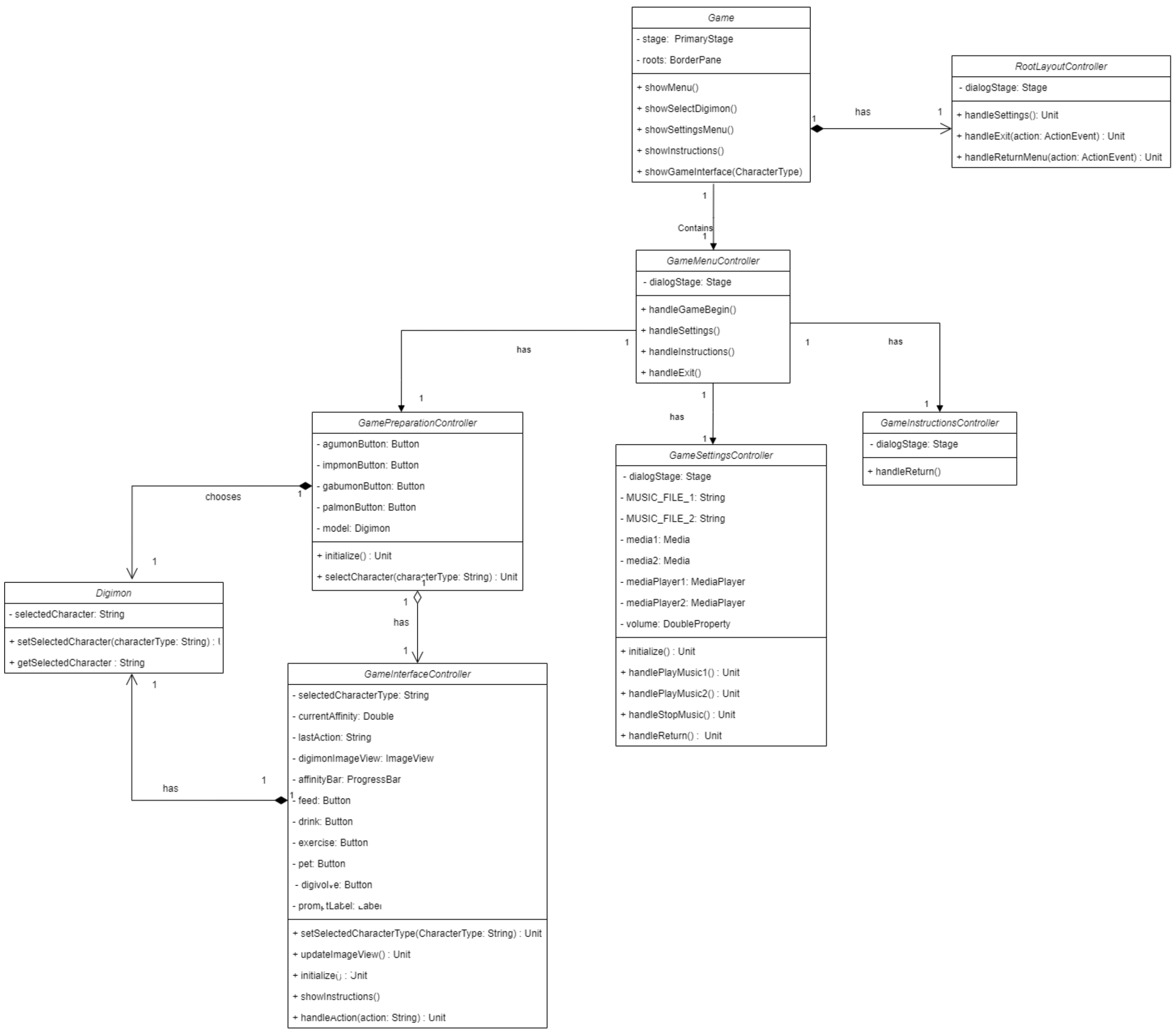
"I Choon Han Wen Benjamin (student name). verify that this paper contains entirely my own work. I have not consulted with any outside person or materials other than what was specified (an interviewee, for example) in the assignment or the syllabus requirements. Further, I have not copied or inadvertently copied ideas, sentences, or paragraphs from another student. I realize the penalties *(refer to page 16, 5.5, Appendix 2, page 44 of the student handbook diploma and undergraduate programme)* for any kind of copying or collaboration on any assignment."

BenjaminChoon 19/8/2023 (Student's signature / Date)

Table Of Contents

Cover Page 1
Table of Contents 2
UML Class Diagram 3
Introduction of Project 4
Functions 5
1.0: Main Menu 5
2.0: Select Character 9
3.0: Game Interface 12
4.0: Game Actions 17
5.0: Game Design Details 23
Reflection 24
References 25

<u>UML Class Diagram</u>



UML Class Diagram for digimonGame

Introduction of Project

This project consists of a mini game, which is based on numerous virtual pet games such as Talking Tom and Tamagochi. It aims to display the programming skills I learnt from this subject via the Scala language. This mini game implements the MVC (Model-View-Controller) structure, which is a design pattern commonly used in software development for creating well-organized and modular applications. Each component has a specific responsibility, contributing to a clear separation of concerns and making the application easier to maintain and extend.

To elaborate, the Model component is used generally to encapsulate the application's data and defines how it can be manipulated and accessed, and it is not affiliated with the View and Controller component. In my case, the Model component is not used widely since my "model", the Digimon only has one core attribute, its Character Type. For the View component, it is responsible for rendering the user interface and presenting the data. It does not contain any logic, since it is only responsible for displaying data. In my project, 6 .fxml files are used to represent my game interface. Lastly for the Controller component, it manages user interactions with the application. It receives user input from the View component and translates it into actions to be performed. Application logic are all handled here. Similarly to my View files, there are also 6 .scala files containing my game logic.

As for the game design, I chose to use Digimon as my main topic is due to my love of the series which was also my childhood. Originally, I planned to make a fighting game based on the Digimon series, but sadly my abilities were limited. The title of the game, "Digimon: The Chosen Memories", is a brainchild of me and my friend MysticK (alias), who is also a big fan of the series. There is no particular reason for the name actually, we just think that this name can represent our memories of the series. MysticK also drew the background images for the Menu and game Interface. I have included sound effects in my game, using sound effects from famous games such as Final Fantasy and Minecraft, to pay homage to these games that also played a part in my childhood.

Recognizing my own limitations is also a part of growth. For my game, the main limitations are the lack of a database and the problem of overlapping sound effects. For the latter, I've tried pausing the initial playing sound effect to accommodate the next sound effect, but the results were not good, so it was not implemented in the final product. Strength wise, I have put in effort in the details, for example having different alerts when exiting the game from different interfaces.

I believe that this game is the best of my efforts, and also wish that it catches your attention with its uniqueness and design. Thanks for your time and effort.

YouTube video link for project demo:

https://youtu.be/5otvs_DuSLE

Functions

1.0: Game Menu

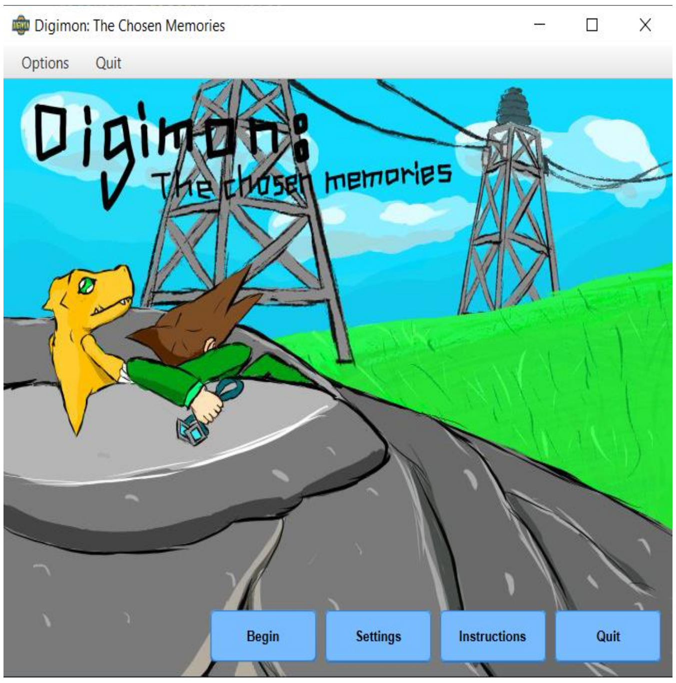
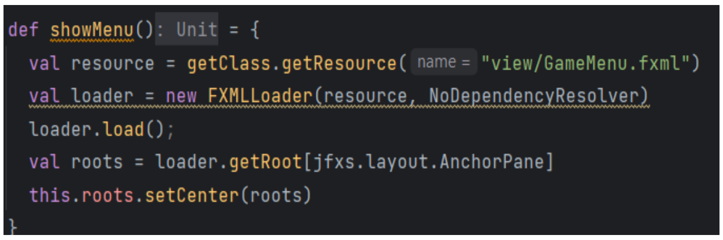
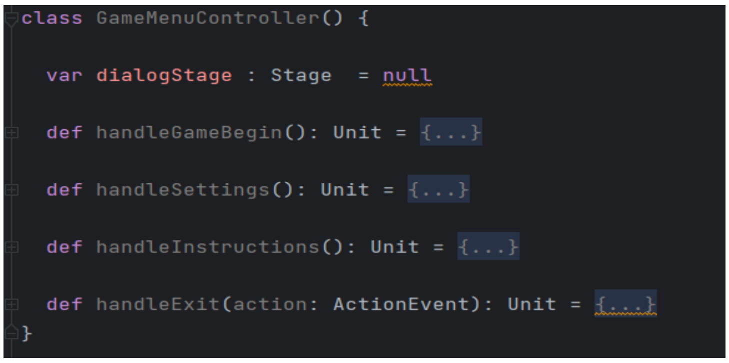


Figure 1: Game Menu



```
def showMenu(): Unit = {
  val resource = getClass.getResource( name = "view/GameMenu.fxml")
  val loader = new FXMLLoader(resource, NoDependencyResolver)
  loader.load();
  val roots = loader.getRoot[jfxs.layout.AnchorPane]
  this.roots.setCenter(roots)
}
```

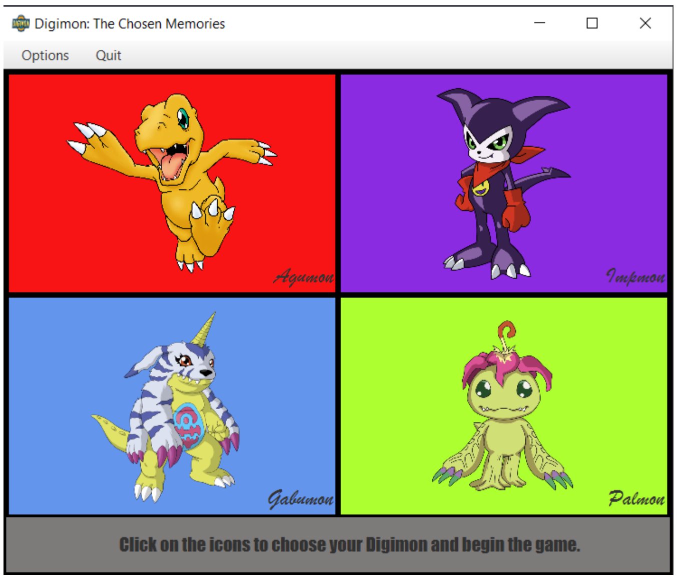
Figure 1.1: Method to display Game Menu



```
class GameMenuController() {

  var dialogStage : Stage  = null

  def handleGameBegin(): Unit = {...}

  def handleSettings(): Unit = {...}

  def handleInstructions(): Unit = {...}

  def handleExit(action: ActionEvent): Unit = {...}
}
```

Figure 1.2: Controller class for Game Menu

For the Game Menu, 4 buttons are included which are Begin, Settings, Instructions and Quit. These buttons are associated with its respective functions in the controller class (GameMenuController.scala). As an example, when the Quit button is clicked, the handleExit function will be called, which creates an instance of Alert with a confirmation dialog. These functions either accepts arguments or do not accept any arguments, such as handleGameBegin, which is parameterless unlike handleExit which has an ActionEvent parameter.



Click on the icons to choose your Digimon and begin the game.

Figure 1.3: Begin Button on trigger

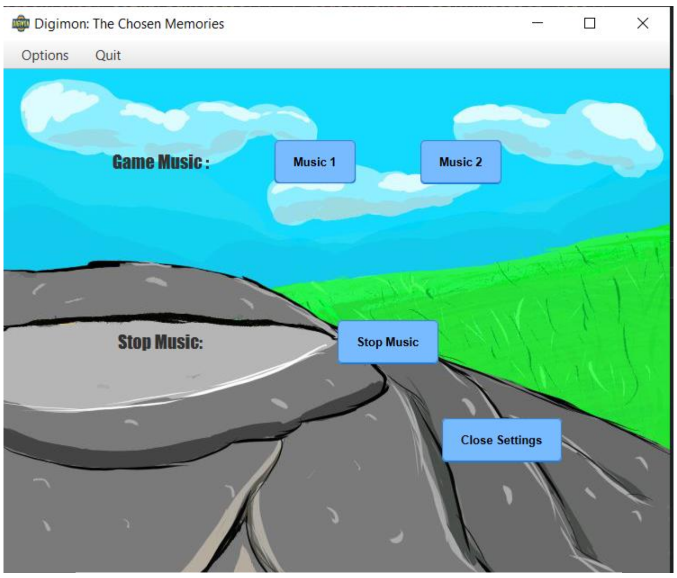
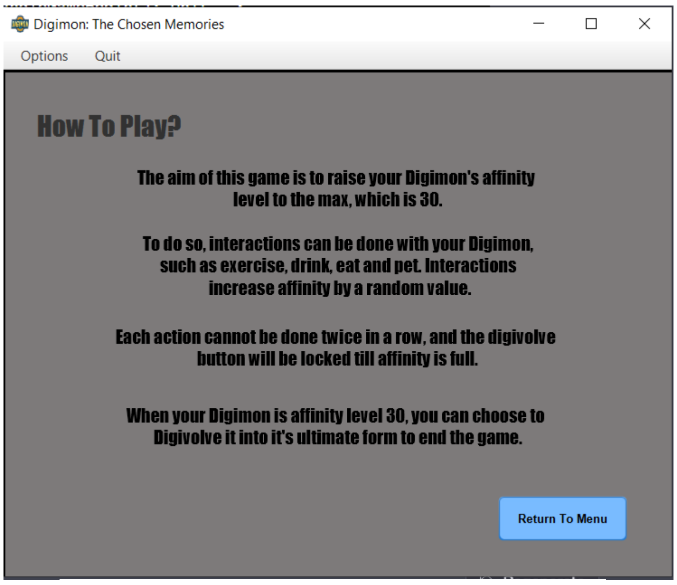


Figure 1.4: Settings Button on trigger



How To Play?

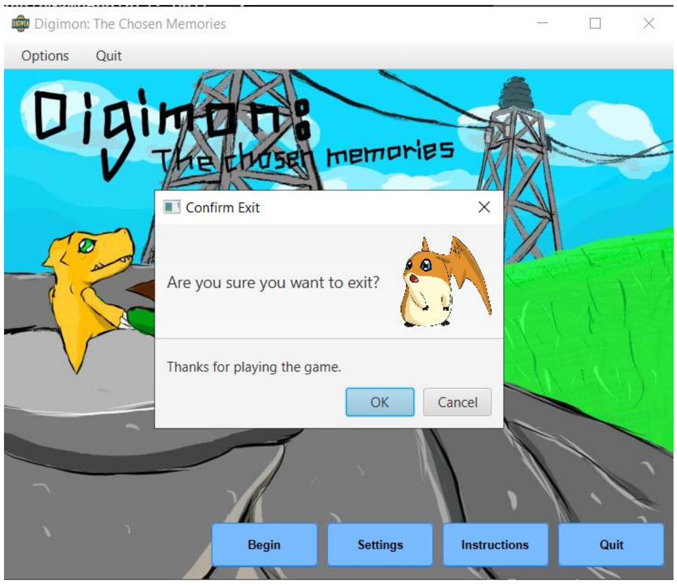
The aim of this game is to raise your Digimon's affinity level to the max, which is 30.

To do so, interactions can be done with your Digimon, such as exercise, drink, eat and pet. Interactions increase affinity by a random value.

Each action cannot be done twice in a row, and the digivolve button will be locked till affinity is full.

When your Digimon is affinity level 30, you can choose to Digivolve it into it's ultimate form to end the game.

Figure 1.5: Instructions Button on trigger



Are you sure you want to exit?

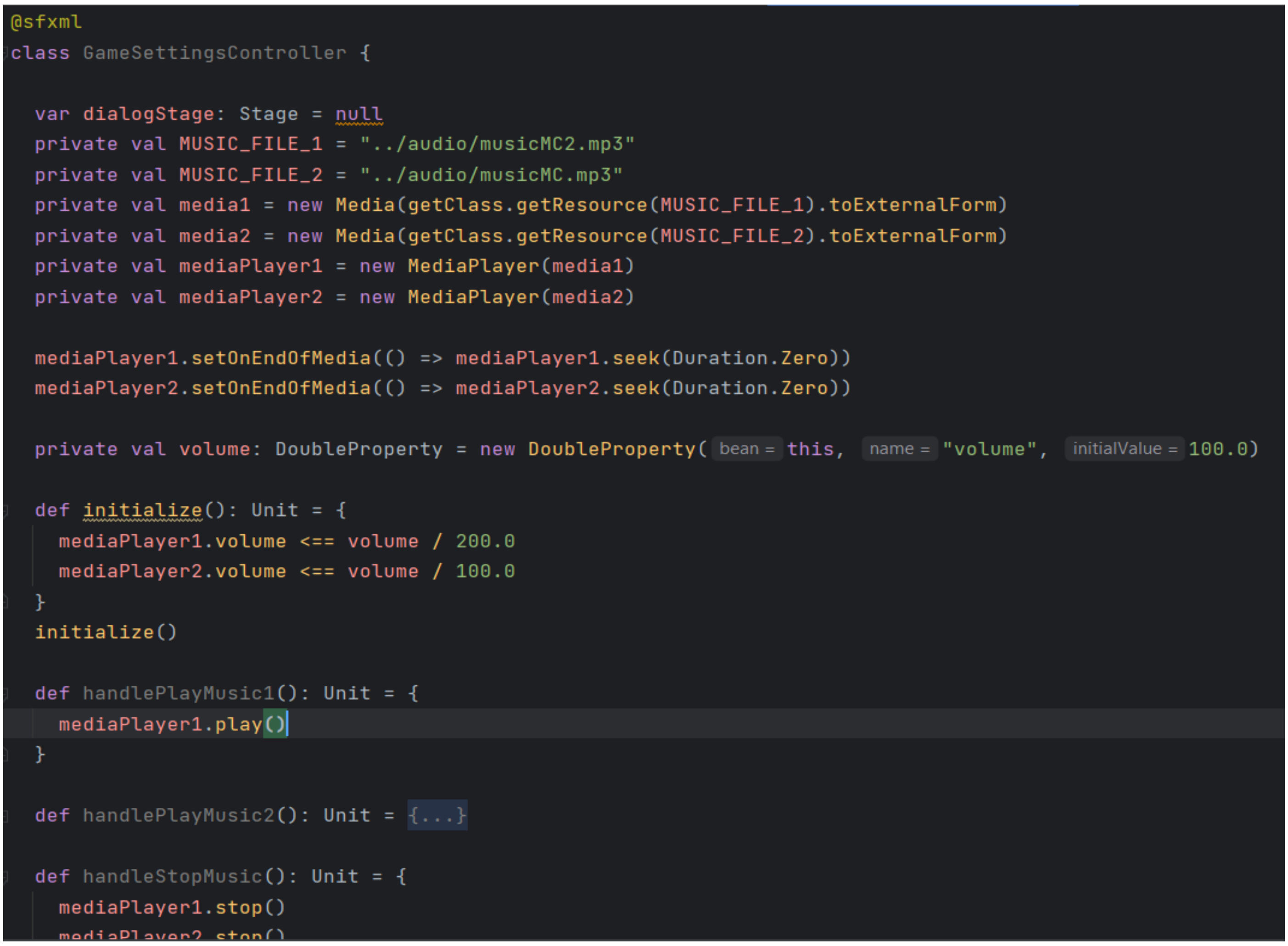
Thanks for playing the game.

Figure 1.6: Quit Button on trigger

Each button is bound to a function which will display the respective pages needed. For Figure 1.3, it will be discussed further in the report. As for Figure 1.4, the page has buttons which can control the BGM (Background Music) which is set to none on default. Figure 1.7 below shows the controller class for settings associated with settings.

In settings, user can choose from two types of BGM to play and also have choice to stop BGM. A limitation is found here, as the BGM can only be adjusted before or after a game.

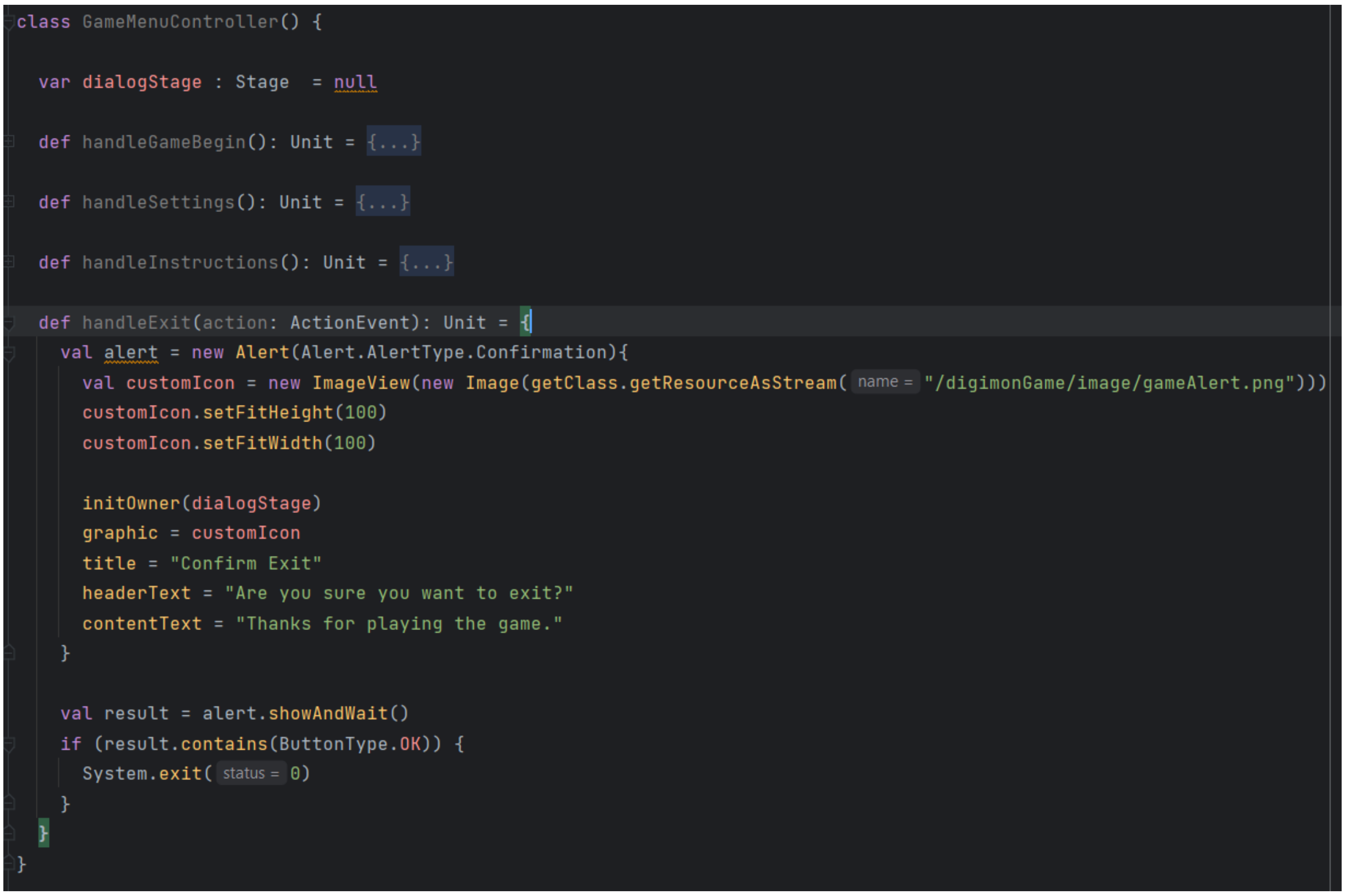
For code explanation, only important components will be mentioned. Basic components with almost no logic involved such as instructions will not be mentioned. Import statements will also not be displayed in the code screenshotd to save space. This also applies to other function explanation.



```
@sfxml
class GameSettingsController {

  var dialogStage: Stage = null
  private val MUSIC_FILE_1 = "../audio/musicMC2.mp3"
  private val MUSIC_FILE_2 = "../audio/musicMC.mp3"
  private val media1 = new Media(getClass.getResource(MUSIC_FILE_1).toExternalForm)
  private val media2 = new Media(getClass.getResource(MUSIC_FILE_2).toExternalForm)
  private val mediaPlayer1 = new MediaPlayer(media1)
  private val mediaPlayer2 = new MediaPlayer(media2)

  mediaPlayer1.setOnEndOfMedia(() => mediaPlayer1.seek(Duration.Zero))
  mediaPlayer2.setOnEndOfMedia(() => mediaPlayer2.seek(Duration.Zero))

  private val volume: DoubleProperty = new DoubleProperty( bean = this,  name = "volume",  initialValue = 100.0)

  def initialize(): Unit = {
    mediaPlayer1.volume <== volume / 200.0
    mediaPlayer2.volume <== volume / 100.0
  }
  initialize()

  def handlePlayMusic1(): Unit = {
    mediaPlayer1.play()
  }

  def handlePlayMusic2(): Unit = {...}

  def handleStopMusic(): Unit = {
    mediaPlayer1.stop()
```

Figure 1.7: Code associated with BGM

1. private val MUSIC_FILE_1 and private val MUSIC_FILE_2 are private constant variables with the path to the first music file.
2. private val media1 and private val media2 creates a Media object using the URL obtained from MUSIC_FILE_1 and 2.
3. private val mediaPlayer1 and private val mediaPlayer2 creates a MediaPlayer object using the first Media object. The media player is used to play audio.
4. mediaPlayer1.setOnEndOfMedia(() => mediaPlayer1.seek(Duration.Zero)): This line sets an event handler that seeks the media player back to the beginning when the end of the media is reached. This effectively loops the first music file. (Same goes for the next line).
5. private val volume: DoubleProperty = new DoubleProperty(this, "volume", 100.0): This line defines a DoubleProperty named volume that represents the volume level of the music. It's initialized with a default value of 100.0. It is encapsuled within a initialize() method which sets up the initial state of the controller and media players when the FXML components are initialized.
6. def handlePlayMusic1/2(): Unit = { ... }: This method is called when the user clicks a button to play the first/second music file using mediaPlayer1/2. handleStopMusic instead stops both music players.



```
class GameMenuController() {

  var dialogStage : Stage  = null

  def handleGameBegin(): Unit = {...}

  def handleSettings(): Unit = {...}

  def handleInstructions(): Unit = {...}

  def handleExit(action: ActionEvent): Unit = {
    val alert = new Alert(Alert.AlertType.Confirmation){
      val customIcon = new ImageView(new Image(getClass.getResourceAsStream( name = "/digimonGame/image/gameAlert.png")))
      customIcon.setFitHeight(100)
      customIcon.setFitWidth(100)

      initOwner(dialogStage)
      graphic = customIcon
      title = "Confirm Exit"
      headerText = "Are you sure you want to exit?"
      contentText = "Thanks for playing the game."
    }

    val result = alert.showAndWait()
    if (result.contains(ButtonType.OK)) {
      System.exit( status = 0)
    }
  }
}
```

Figure 1.7: Code associated with quitting the game

1. def handleExit(action: ActionEvent): Unit = {: This method is called when the user tries to exit the game. It takes an ActionEvent as a parameter, which is typically triggered when a button or other control is interacted with.
2. val alert = new Alert(Alert.AlertType.Confirmation) { ... }: This line creates a new instance of the Alert class with the type Confirmation. The curly braces { ... } indicate creation of an instance of an anonymous subclass of Alert which allows customization of the appearance and behavior of the alert dialog.
3. Within the curly braces, val customIcom = new ImageView(…) creates a new ImageView with a custom icon from the alert dialog. customIcon.setFitHeight()/Width() is used to specify the height and width of the icon.
4. initOwner(dialogStage): This line sets the owner of the alert dialog to be the dialogStage. This helps in proper positioning and focus management. graphic = customIcon sets the custom icon as the graphic for the alert dialog. Title, headerText and contextText sets the text for the dialog.
5. val result = alert.showAndWait(): This line shows the alert dialog and waits for the user's response. It returns an Option[ButtonType] that represents the button the user clicked on (OK, Cancel, etc.). If the user clicked “OK”, an if statement below will execute to terminate the application via System.exit(0).

2.0: Select Character

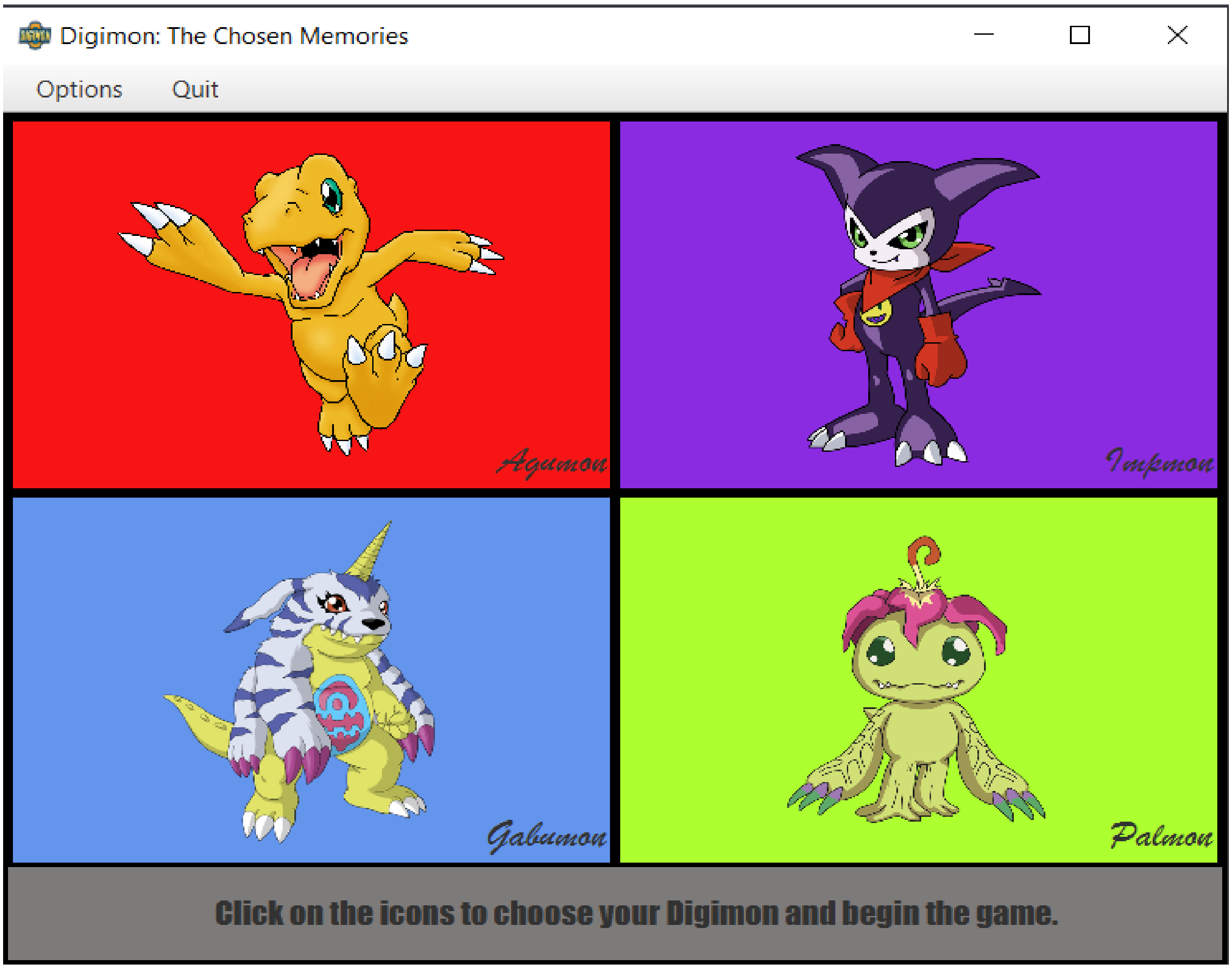
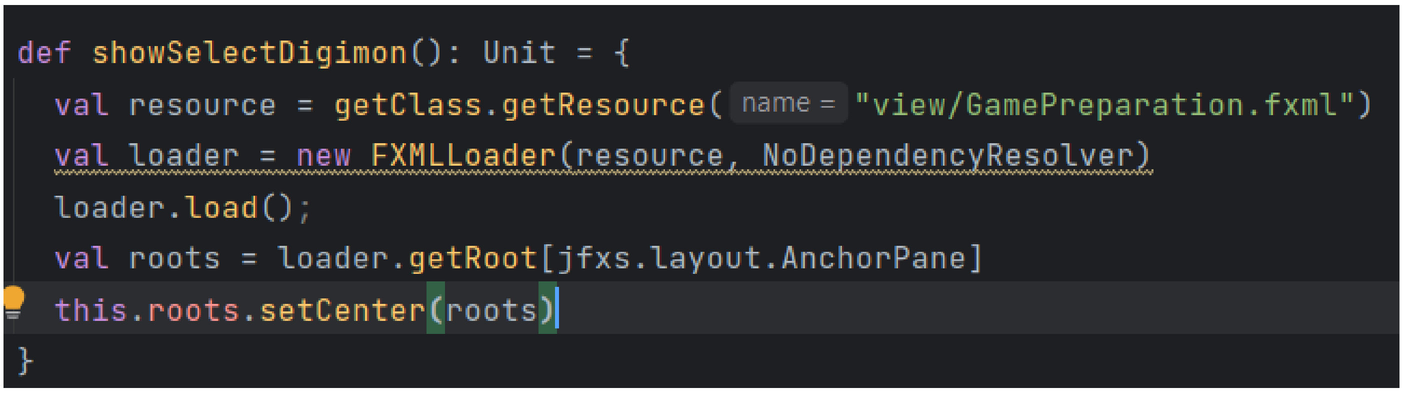

Digimon: The Chosen Memories
Options
Quit
Agumon
Impmon
Gabumon
Palmon
Click on the icons to choose your Digimon and begin the game.

Figure 2.0: Select Character Interface

After the Begin button is clicked, the user will be directed to this page which requires the user to choose a Digimon character for their game instance. This page is implemented via GridPane in scenebuilder, where each pane contains a button with the CSS -fx-background-color property set to “transparent”. This page is considered a function on its own since it has components which are responsible for parsing data from one controller to another, which is a important programming technique.

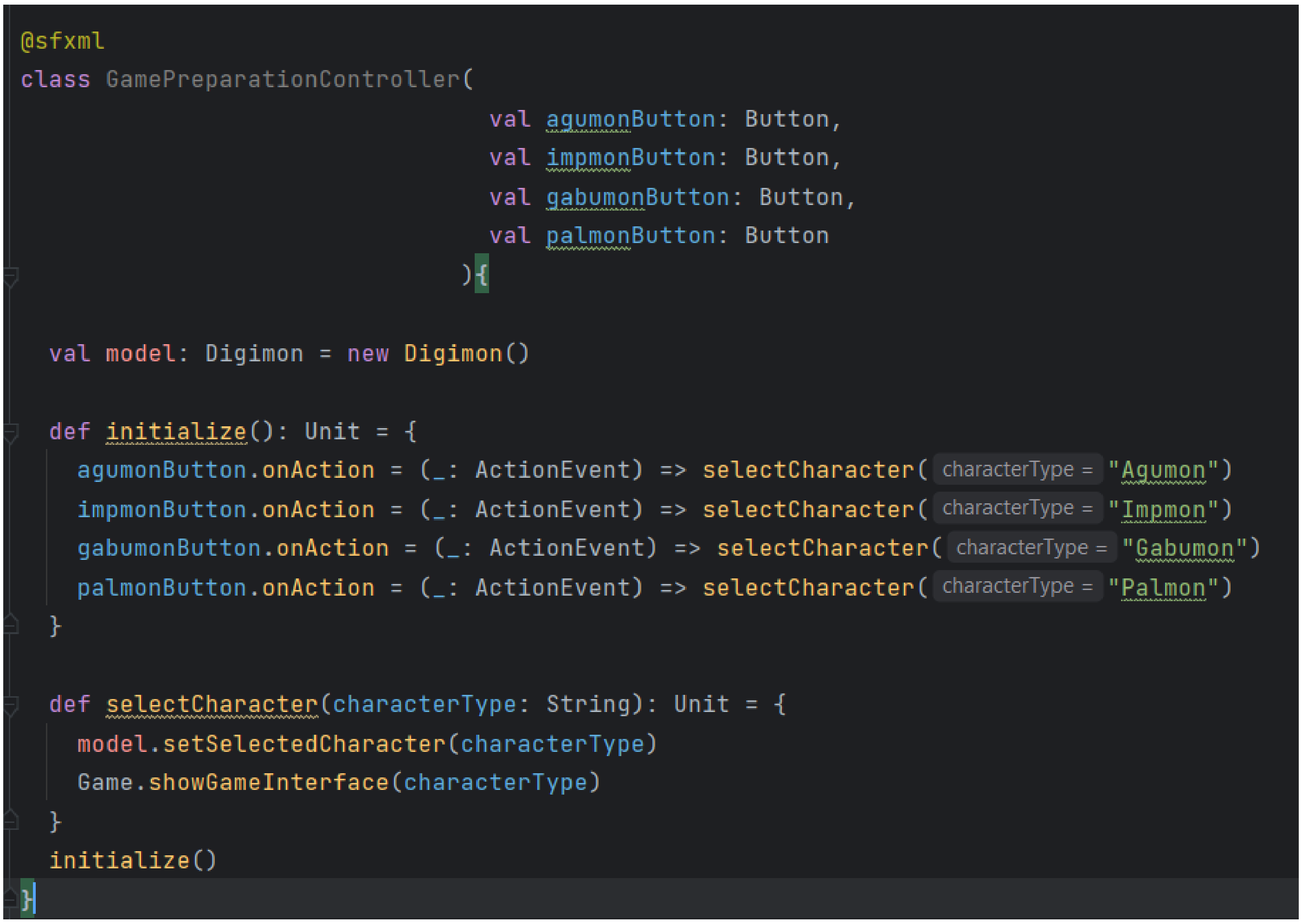


```scala
def showSelectDigimon(): Unit = {
  val resource = getClass.getResource( name = "view/GamePreparation.fxml")
  val loader = new FXMLLoader(resource, NoDependencyResolver)
  loader.load();
  val roots = loader.getRoot[jfxs.layout.AnchorPane]
  this.roots.setCenter(roots)
}
```

Figure 2.1: Code used to call and display interface

1. val resource = getClass.getResource("view/GamePreparation.fxml"): This line retrieves the URL of the FXML resource file named "GamePreparation.fxml" located in the "view" package of your application. FXML files are used to define the user interface layout.
2. val loader = new FXMLLoader(resource, NoDependencyResolver): This line creates a new instance of the FXMLLoader class, which is used to load FXML files. It takes the resource URL and a NoDependencyResolver, which indicates that you're not using dependency injection for the FXML file.
3. loader.load(): This line loads the FXML file specified by the resource URL. It creates the user interface components defined in the FXML file and establishes any necessary bindings and event handlers.
4. val roots = loader.getRoot[jfxs.layout.AnchorPane]: This line retrieves the root node of the FXML file, which is an instance of AnchorPane layout container. The specific type is indicated by the [jfxs.layout.AnchorPane] type parameter.
5. this.roots.setCenter(roots): This line updates the content of the main layout's center area with the roots AnchorPane. this.roots refers to the main layout's center area, and setCenter is used to set its content. This effectively changes the visible screen to the one defined in "GamePreparation.fxml".

In summary, most functions to display new interfaces after user actions are similar to the code in Figure 2.1 except for the code that displays the main game interface. This is due to data from the GamePreparationController.scala beign passed into GameInterfaceController.scala.



```
@sfxml
class GamePreparationController(
                                  val agumonButton: Button,
                                  val impmonButton: Button,
                                  val gabumonButton: Button,
                                  val palmonButton: Button
                                ){

  val model: Digimon = new Digimon()

  def initialize(): Unit = {
    agumonButton.onAction = (_: ActionEvent) => selectCharacter( characterType = "Agumon")
    impmonButton.onAction = (_: ActionEvent) => selectCharacter( characterType = "Impmon")
    gabumonButton.onAction = (_: ActionEvent) => selectCharacter( characterType = "Gabumon")
    palmonButton.onAction = (_: ActionEvent) => selectCharacter( characterType = "Palmon")
  }

  def selectCharacter(characterType: String): Unit = {
    model.setSelectedCharacter(characterType)
    Game.showGameInterface(characterType)
  }
  initialize()
}
```

Figure 2.2: Code associated with selecting a character

1. val *CharacterName*Button : Button indicated that this parameter is a type of "Button" associated with a button with fxid *CharacterName* in the fxml file. The val keyword declares it as a read-only property. These 4 parameters are all part of the GamePreparationController class and the buttons in the FXML file will be injected with these parameters when an instance of this controller is created.
2. def initialize(): Unit = { ... }: This method is called when the controller is initialized. It contains the initialization logic for the controller. In this case, it's used to set up event handlers for the buttons representing different Digimon characters.
3. For each *CharacterName*Button, it has their own onAction property set to a lambda function. This lambda function is triggered when the button is clicked and calls the selectCharacter method with the related character type as its argument.
4. def selectCharacter(characterType: String): Unit = { ... }: This method is called when a character button is clicked. It sets the selected character in the model using the provided characterType. It then calls Game.showGameInterface(characterType) to show the game interface with the selected character.

3.0: Game Interface

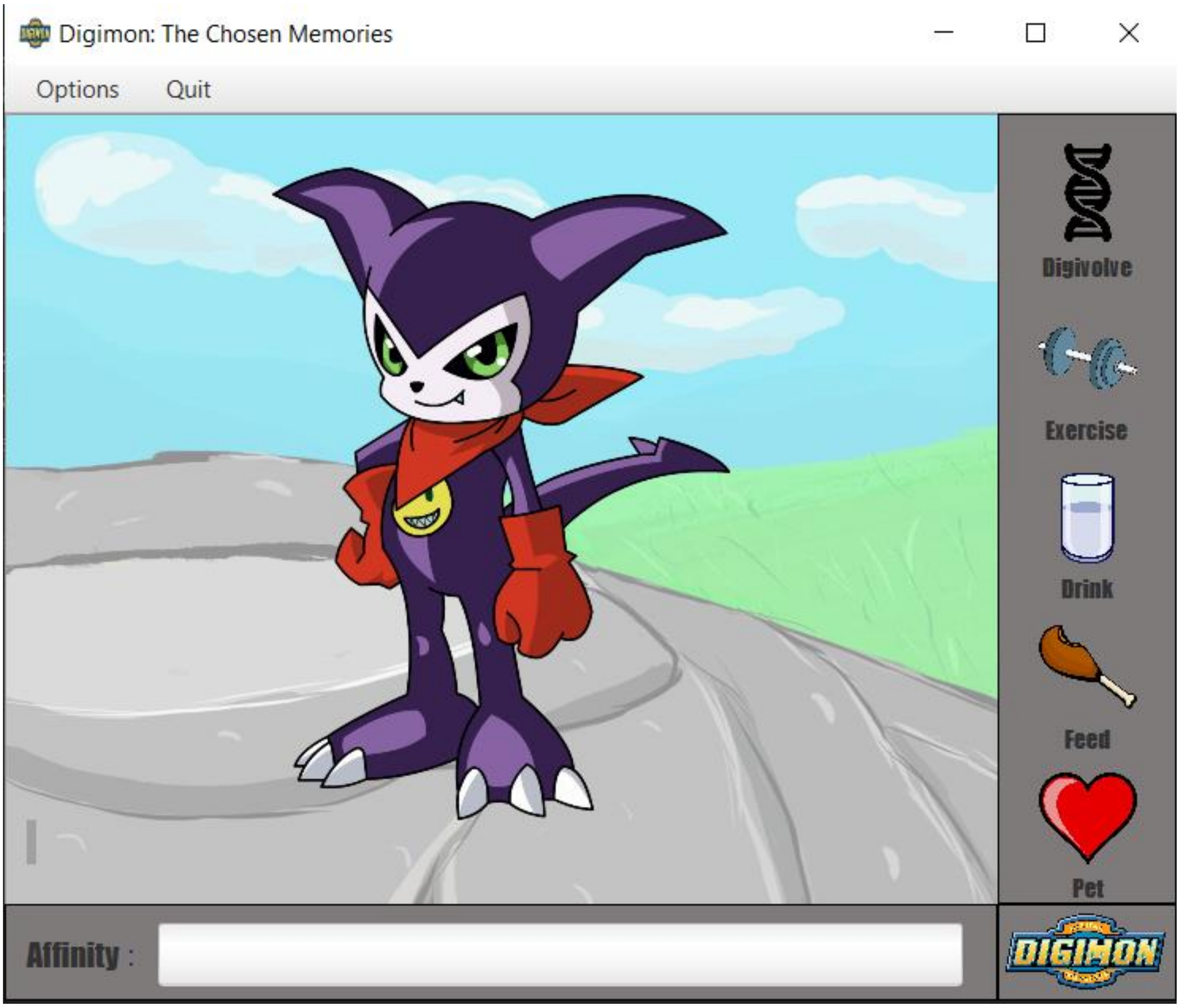
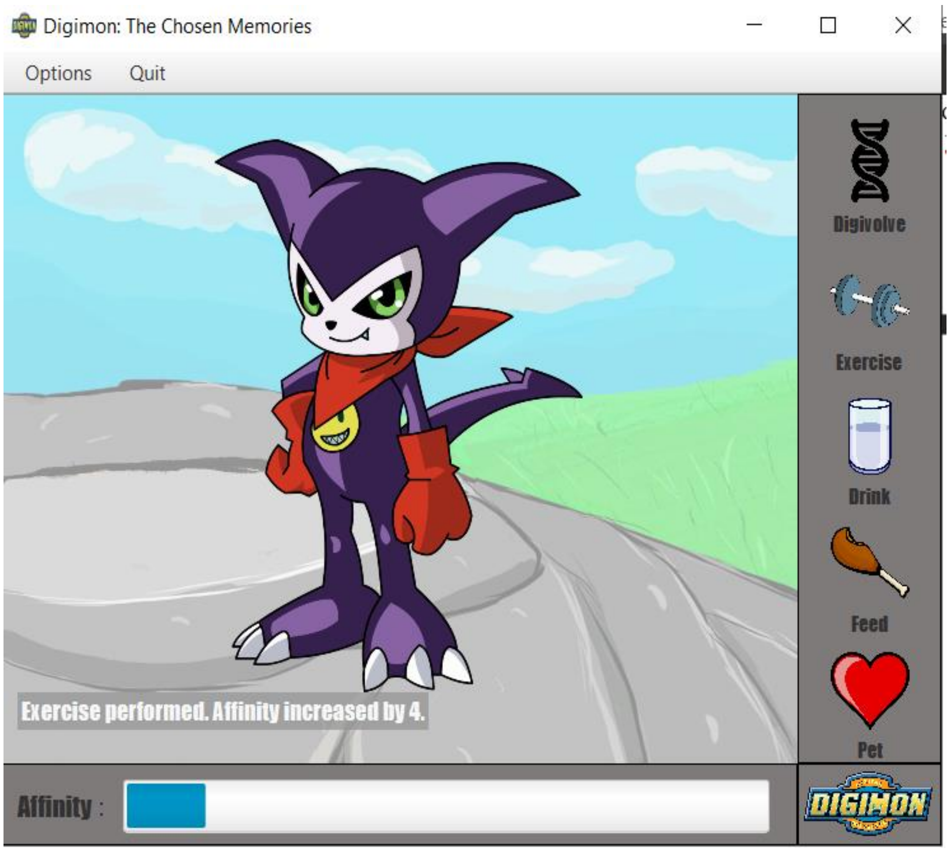


Figure 3.0: Main Game Interface

After the user selects a character, the game will officially begin. The image of the selected Digimon will be displayed here, as it is passed on from the GamePreparationController to the GameInterfaceController which manages the logic used in the game.

The objective of the game is simple, the user has to fill up the affinity bar at the bottom to its max. This can be done by selecting action buttons on the left to perform on your Digimon. To prevent spamming buttons, each button is limited so that it cannot be executed twice in a row. Respective prompts will be displayed along with audio prompts when an action is performed.

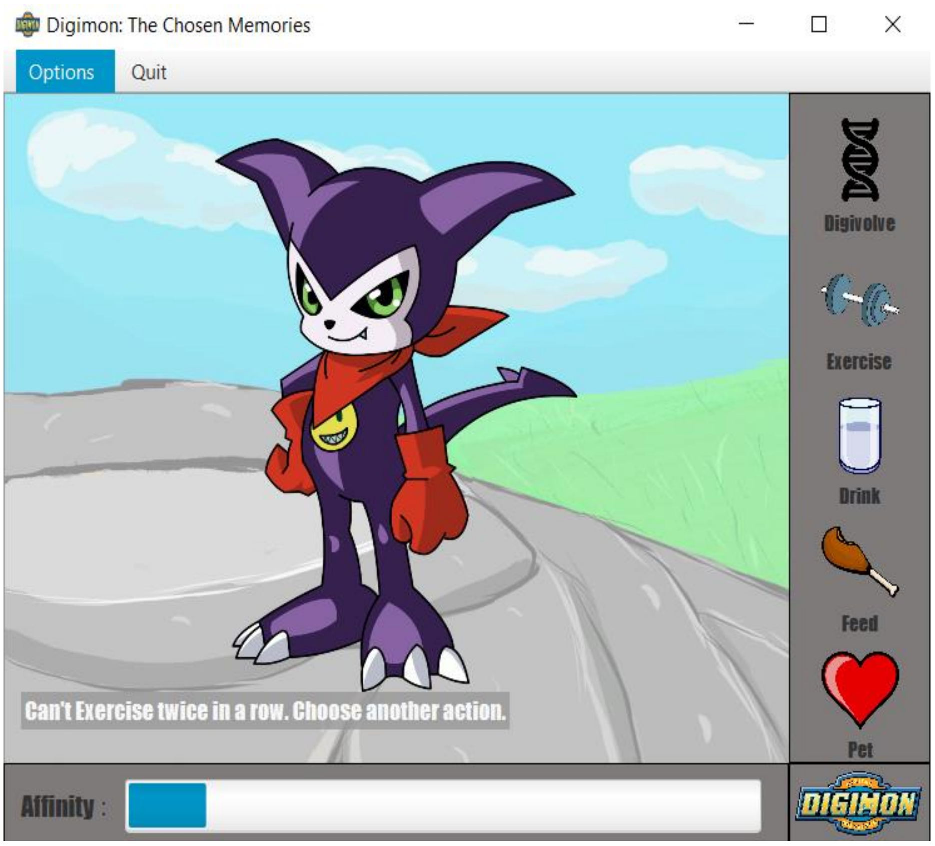
Initially, the Digivolve button will be locked since the affinity bar is not at its max. When the affinity bar is at max, the user can click the Digivolve button to evolve the Digimon and end the game. Minor animations will also be displayed for each action.



Exercise performed. Affinity increased by 4.

A label in the fxml file will be updated whenever the user performs an action. The increased affinity amount will be displayed along with a sound effect. Each action has a different sound effect.

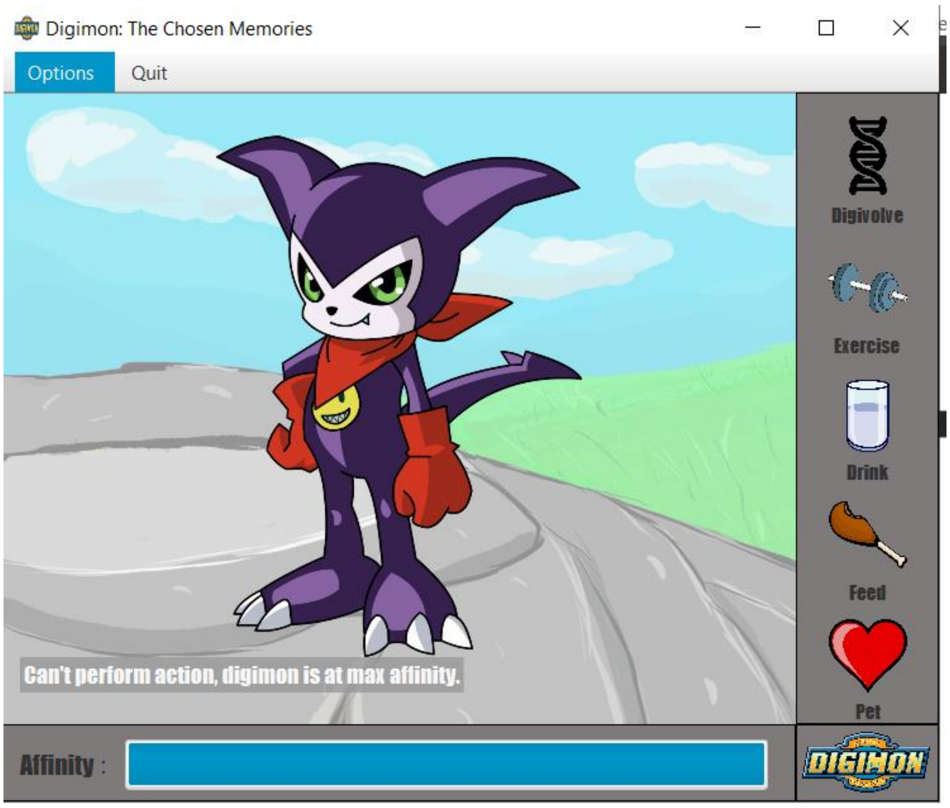
Figure 3.1: Performing Action



Can't Exercise twice in a row. Choose another action.

Duplicate actions will result in an error message along with an error sound effect.

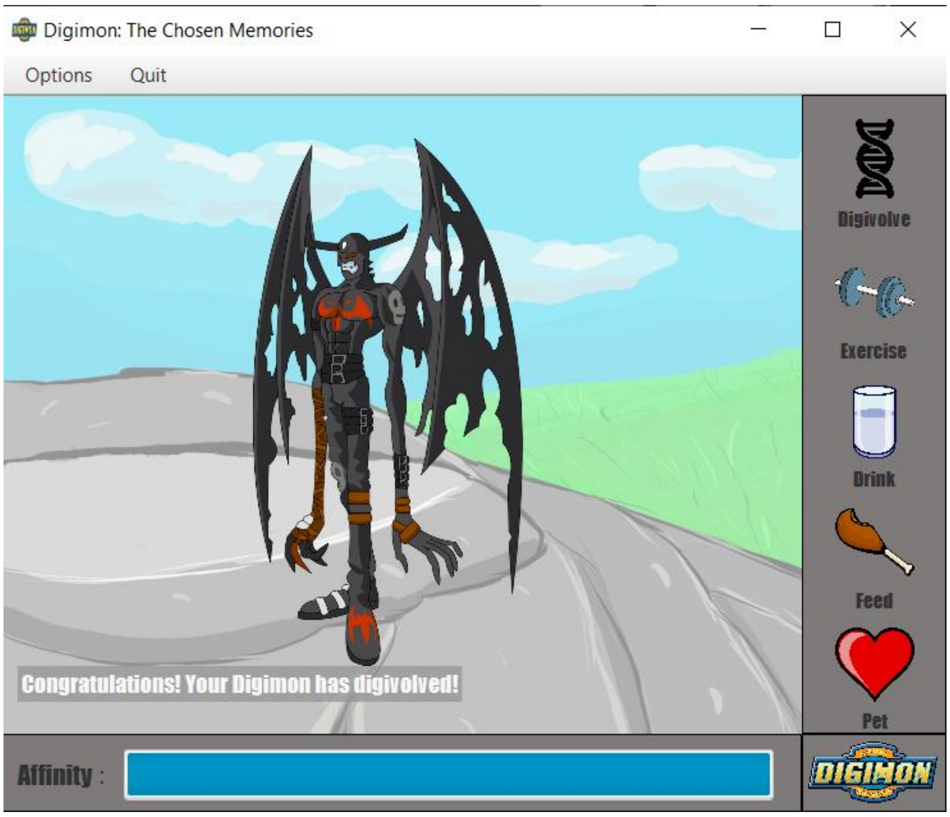
Figure 3.2: Duplicate Action



Can't perform action, digimon is at max affinity.

Figure 3.1: Max Affinity

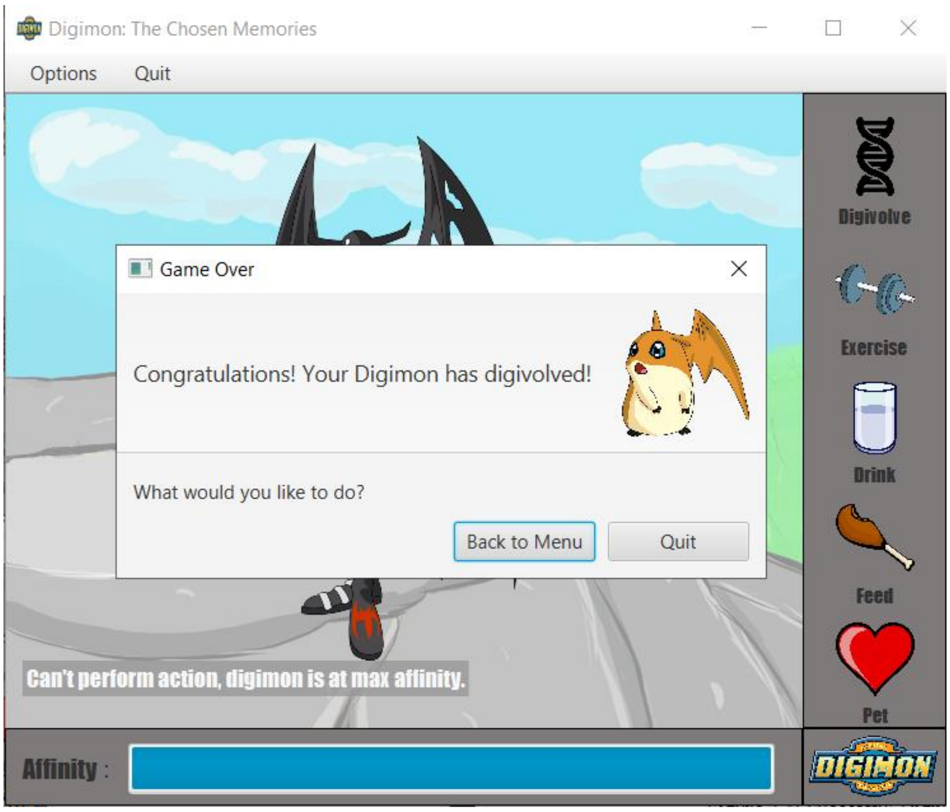
Label will also be updated to inform user that the Digimon is at is max affinity. Error sound effect will also be played.



Congratulations! Your Digimon has digivolved!

Figure 3.4: Successful Digivolve

End of game music will be played when Digivolve button is clicked. Label will also be updated.


Digimon: The Chosen Memories

Game Over

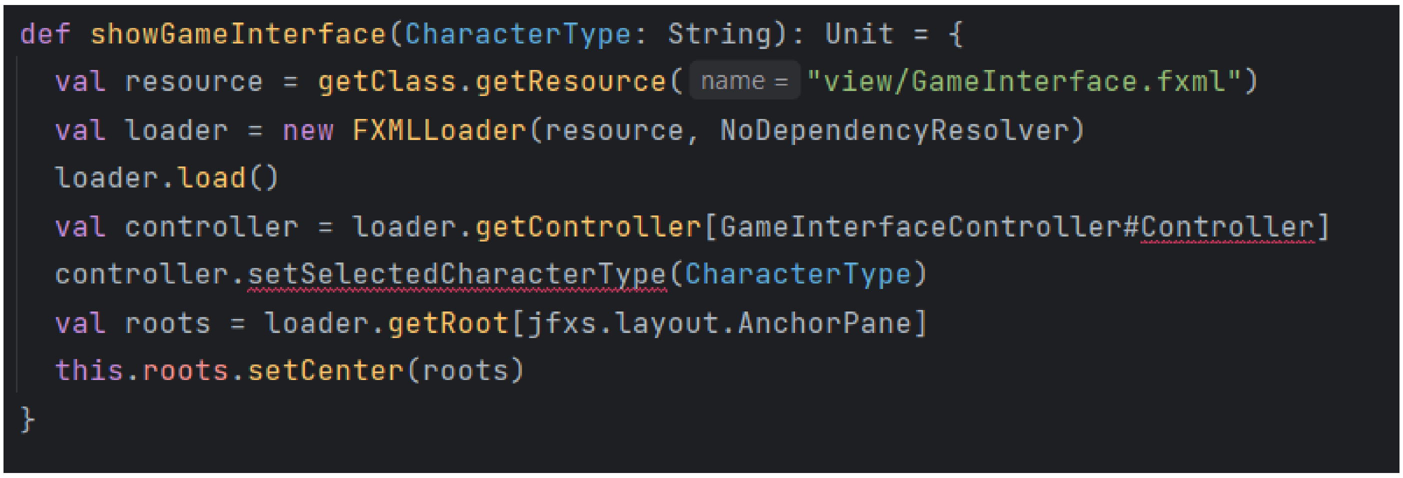
Congratulations! Your Digimon has digivolved!

What would you like to do?

Can't perform action, digimon is at max affinity.

Figure 3.4: Successful Digivolve

After a set time, an alert window will be displayed asking user for their next action. User can choose to return to menu or exit the game.



```
def showGameInterface(CharacterType: String): Unit = {
  val resource = getClass.getResource( name = "view/GameInterface.fxml")
  val loader = new FXMLLoader(resource, NoDependencyResolver)
  loader.load()
  val controller = loader.getController[GameInterfaceController#Controller]
  controller.setSelectedCharacterType(CharacterType)
  val roots = loader.getRoot[jfxs.layout.AnchorPane]
  this.roots.setCenter(roots)
}
```

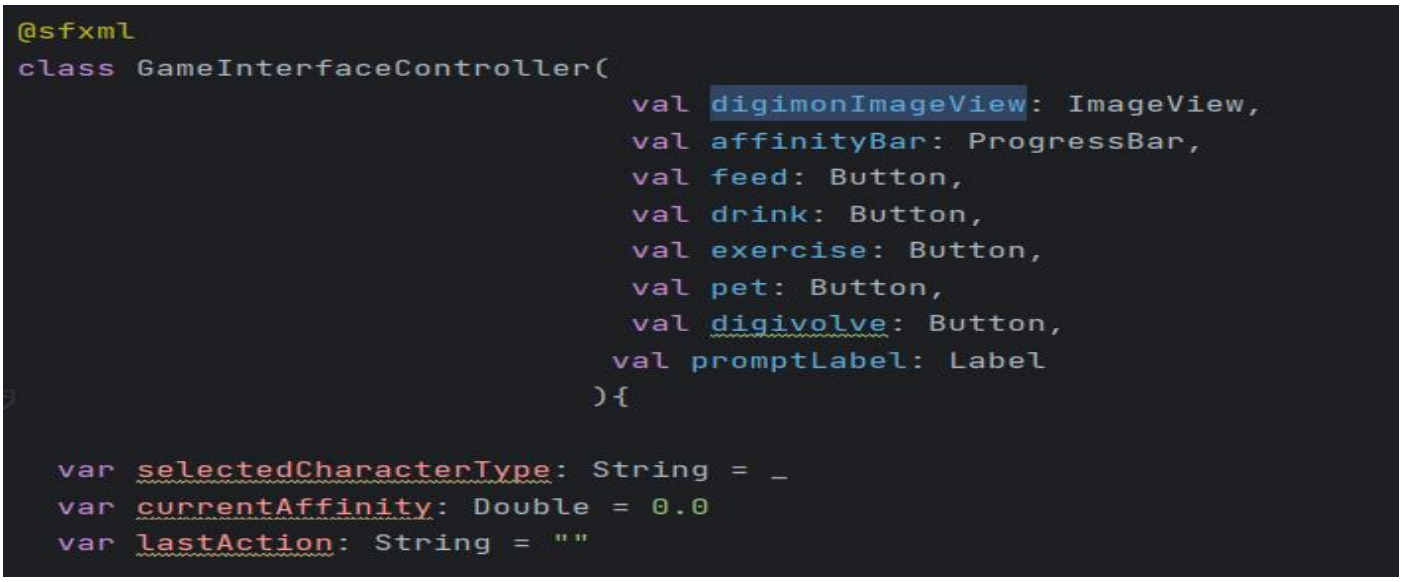
Figure 3.5: Code associated with displaying game interface

This section of code is mostly similar with other codes used to display interfaces. However, one key difference is that it actually assigns parameters to a method in the controller class retrived from the GamePreparationController (Figure 2.2).

1. val controller = loader.getController[GameInterfaceController#Controller]: The controller associated with the loaded FXML is retrieved using getController.
2. controller.setSelectedCharacterType(CharacterType): The setSelectedCharacterType method of the controller is called, passing the CharacterType parameter. This method is assumed to be part of the inner Controller class of the GameInterfaceController.

4.0: Game Actions

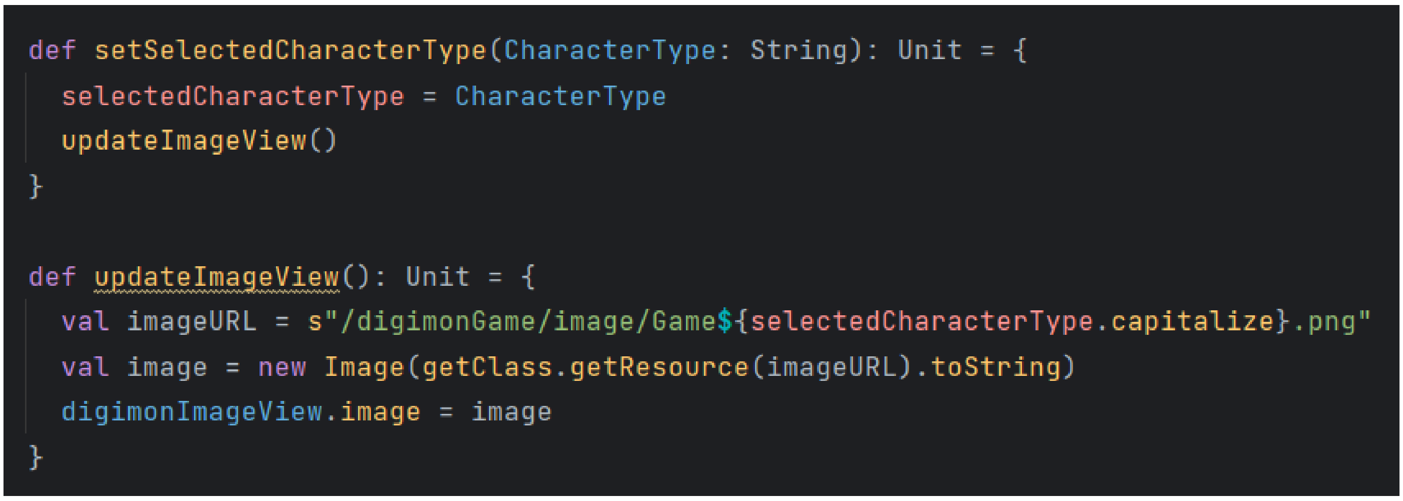
This section will focus on the logic behind the actions performed. Due to the code being too long, it will be segmented heavily to improve readability.



```
@sfxml
class GameInterfaceController(
                                               val digimonImageView: ImageView,
                                               val affinityBar: ProgressBar,
                                               val feed: Button,
                                               val drink: Button,
                                               val exercise: Button,
                                               val pet: Button,
                                               val digivolve: Button,
                                              val promptLabel: Label
                                             ){

  var selectedCharacterType: String = _
  var currentAffinity: Double = 0.0
  var lastAction: String = ""
```

Figure 4.0: Value declaration

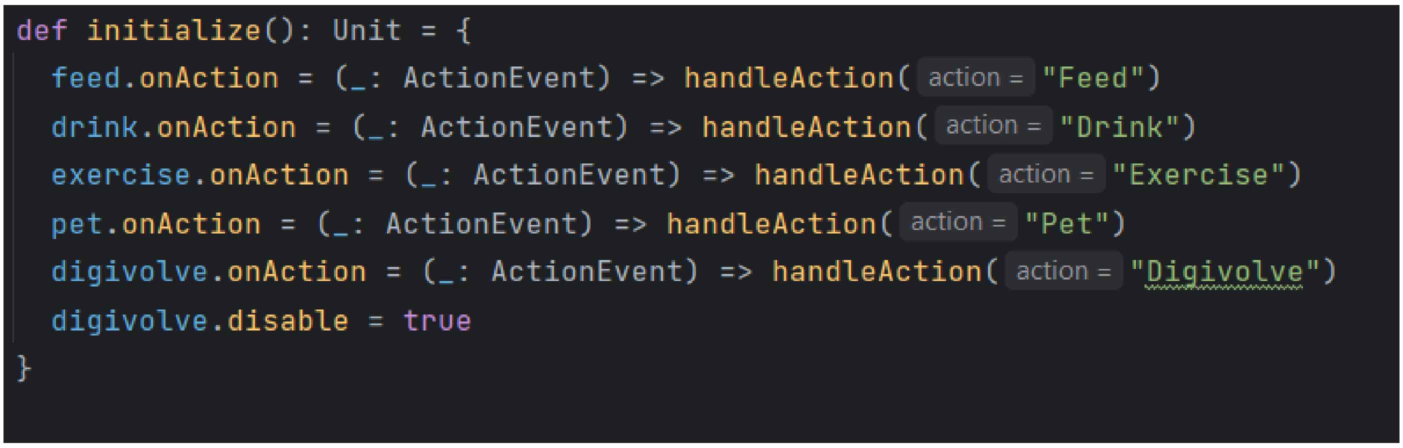
Similar to Figure 2.2, this part focuses on decalring values.



```
def setSelectedCharacterType(CharacterType: String): Unit = {
  selectedCharacterType = CharacterType
  updateImageView()
}

def updateImageView(): Unit = {
  val imageURL = s"/digimonGame/image/Game${selectedCharacterType.capitalize}.png"
  val image = new Image(getClass.getResource(imageURL).toString)
  digimonImageView.image = image
}
```

Figure 4.1: Update image

These two methods focuses on updating the displayed image of the charcter in the Game Interface. setSelectedCharacterType(CharacterType: String): Unit: = {…} takes a CharacterType parameter representing the selected character and sets the selectedCharacterType field of the controller to this value. After updating the selected character type, it calls the updateImageView() method to update the displayed image. updateImageView(): Unit {…} constructs an image URL based on the selected character loads the corresponding image from the resources. It then updates the digimonImageView with the new image.

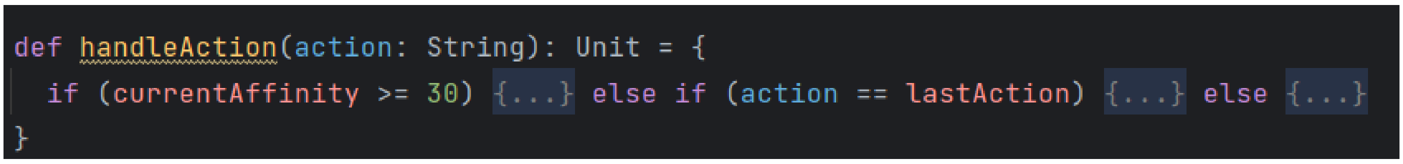


```scala
def initialize(): Unit = {
  feed.onAction = (_: ActionEvent) => handleAction( action = "Feed")
  drink.onAction = (_: ActionEvent) => handleAction( action = "Drink")
  exercise.onAction = (_: ActionEvent) => handleAction( action = "Exercise")
  pet.onAction = (_: ActionEvent) => handleAction( action = "Pet")
  digivolve.onAction = (_: ActionEvent) => handleAction( action = "Digivolve")
  digivolve.disable = true
}
```

Figure 4.2: Initialize

This method is called when the controller is initialized. It contains the initialization logic for the controller. It's responsible for setting up the initial event handlers and configurations for the buttons and controls in the game interface. As mentioned, for each action it has their own onAction property set to a lambda function. When a button is pressed the respective handleAction() method will be invoked.

digivolve.disable = true disables the digivolve button initially, so the user can't click it until specific conditions are met (as indicated by the handleAction method).



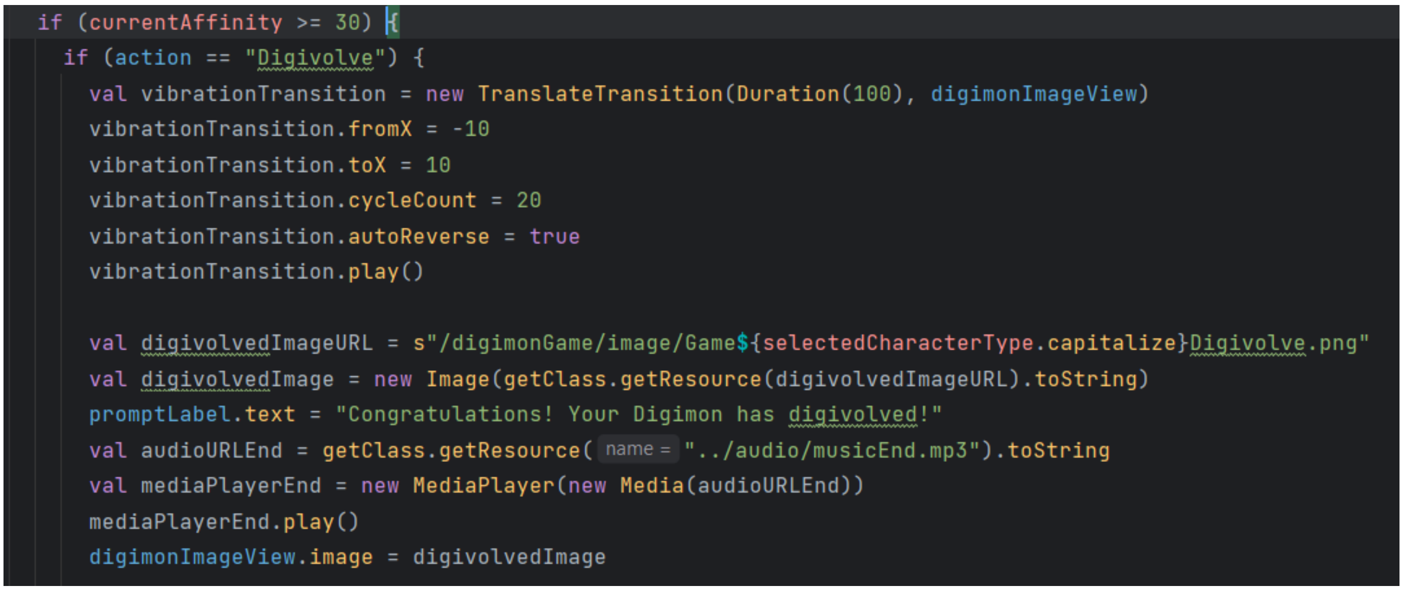
```scala
def handleAction(action: String): Unit = {
  if (currentAffinity >= 30) {...} else if (action == lastAction) {...} else {...}
}
```

Figure 4.3: If-else statement

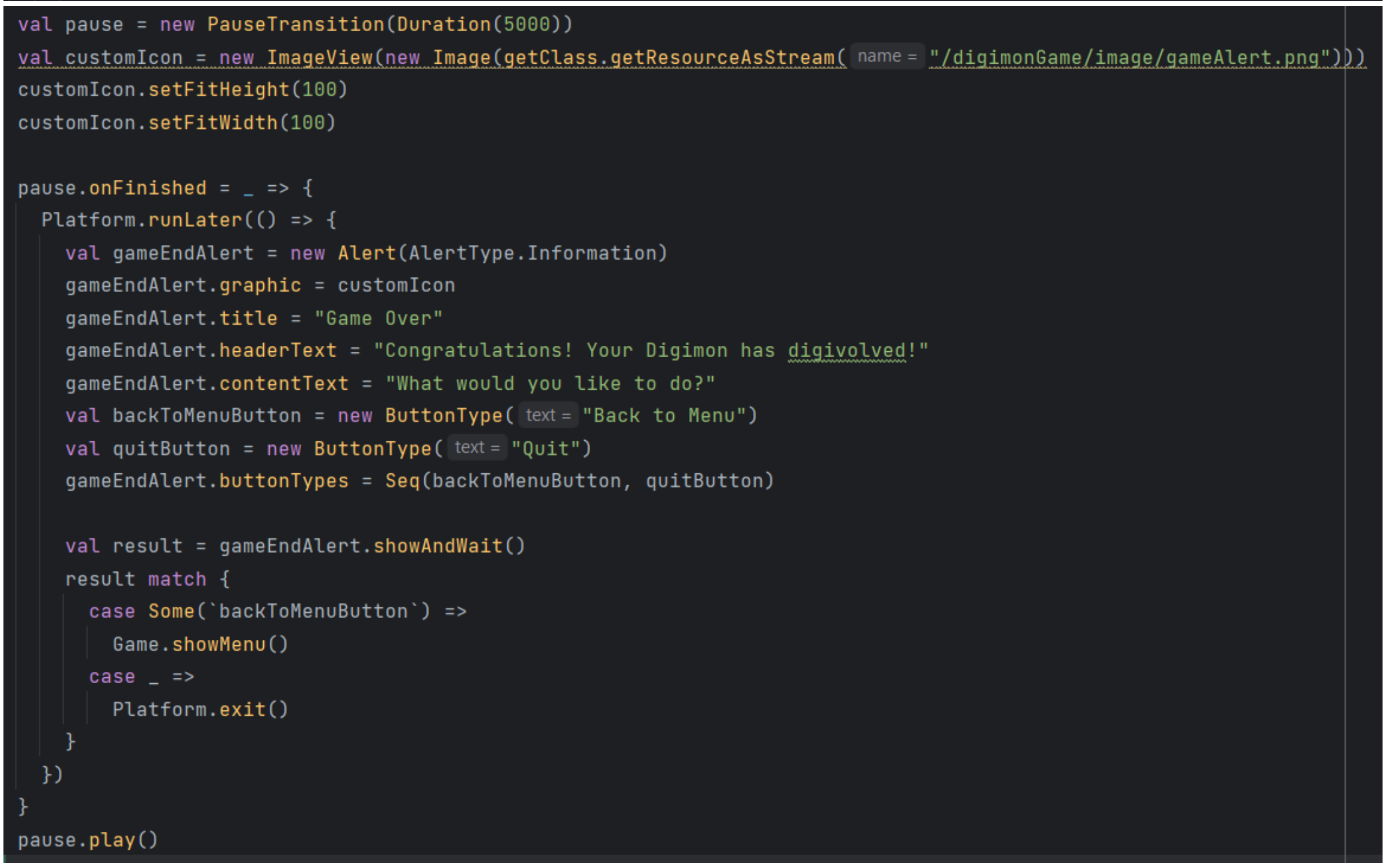
This if-else statement is crucial since it manages the main game logic. It handles three types of situations, one where affinity is over 30, one where the current action is identical to the last action performed and one where normal actions are performed,

Below will be a detailed explanation for each situation.

Situation 1: Digimon Affinity >= 30



```
if (currentAffinity >= 30) {
  if (action == "Digivolve") {
    val vibrationTransition = new TranslateTransition(Duration(100), digimonImageView)
    vibrationTransition.fromX = -10
    vibrationTransition.toX = 10
    vibrationTransition.cycleCount = 20
    vibrationTransition.autoReverse = true
    vibrationTransition.play()

    val digivolvedImageURL = s"/digimonGame/image/Game${selectedCharacterType.capitalize}Digivolve.png"
    val digivolvedImage = new Image(getClass.getResource(digivolvedImageURL).toString)
    promptLabel.text = "Congratulations! Your Digimon has digivolved!"
    val audioURLEnd = getClass.getResource( name = "../audio/musicEnd.mp3").toString
    val mediaPlayerEnd = new MediaPlayer(new Media(audioURLEnd))
    mediaPlayerEnd.play()
    digimonImageView.image = digivolvedImage
```



```
val pause = new PauseTransition(Duration(5000))
val customIcon = new ImageView(new Image(getClass.getResourceAsStream( name = "/digimonGame/image/gameAlert.png")))
customIcon.setFitHeight(100)
customIcon.setFitWidth(100)

pause.onFinished = _ => {
  Platform.runLater(() => {
    val gameEndAlert = new Alert(AlertType.Information)
    gameEndAlert.graphic = customIcon
    gameEndAlert.title = "Game Over"
    gameEndAlert.headerText = "Congratulations! Your Digimon has digivolved!"
    gameEndAlert.contentText = "What would you like to do?"
    val backToMenuButton = new ButtonType( text = "Back to Menu")
    val quitButton = new ButtonType( text = "Quit")
    gameEndAlert.buttonTypes = Seq(backToMenuButton, quitButton)

    val result = gameEndAlert.showAndWait()
    result match {
      case Some(`backToMenuButton`) =>
        Game.showMenu()
      case _ =>
        Platform.exit()
    }
  })
}
pause.play()
```

Figure 4.4: Affinity >= 30

This code segment is part of the handleAction method in the GameInterfaceController class and is executed when the user clicks the "Digivolve" button in the game interface and manages to digivolve successfully with affinity of 30. Below are the main components of the code:

Vibration Animation: It starts by creating a TranslateTransition animation for the digimonImageView. This animation simulates a "vibration" effect by moving the image slightly back and forth horizontally. This creates a visual effect to indicate something dynamic is happening.

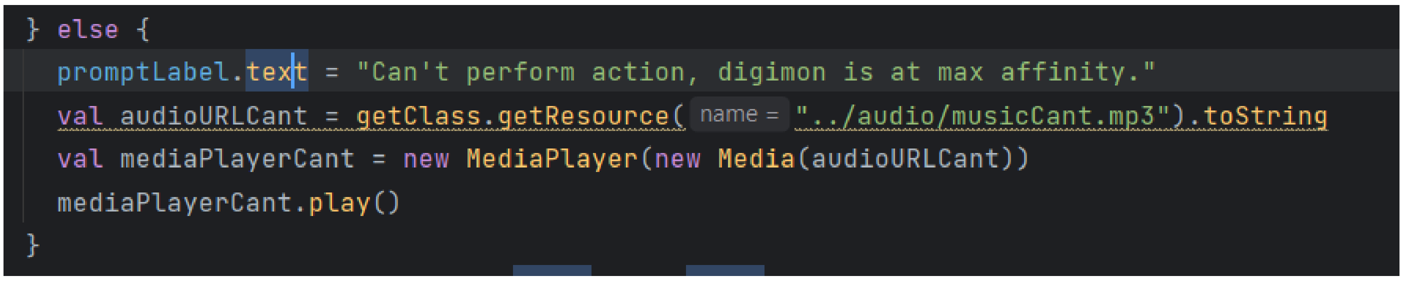
Digivolution Animation: After the vibration animation, it loads a new image URL corresponding to the digivolved form of the selected character type. It sets this new image as the digimonImageView to visually represent the digivolution. It also updates the promptLabel to show a congratulatory message for the digivolution.

Audio Playback: The code then initializes and plays an audio clip, signaling the achievement of the digivolution. It uses the MediaPlayer class to play an audio file located at the specified URL.

Game End Alert: A PauseTransition is used to create a delay before displaying a game end alert. This is done using the Platform.runLater method to ensure the alert is shown on the ScalaFX application thread.

Game End Alert Creation: Inside the Platform.runLater block, a custom game end alert is created. It contains an icon, a title, a header text, and content text to inform the user about the successful digivolution. Two button types are defined: "Back to Menu" and "Quit."

Alert Result Handling: The result of the alert is obtained using the showAndWait method. Based on the user's choice, the code either navigates back to the game menu using Game.showMenu() if the user chooses "Back to Menu," or it exits the application by calling Platform.exit() if the user chooses any other option.

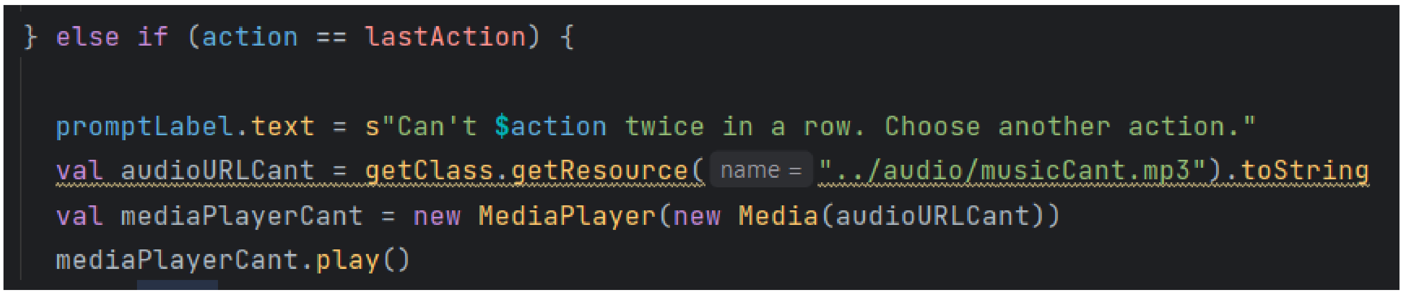


```
} else {
  promptLabel.text = "Can't perform action, digimon is at max affinity."
  val audioURLCant = getClass.getResource( name = "../audio/musicCant.mp3").toString
  val mediaPlayerCant = new MediaPlayer(new Media(audioURLCant))
  mediaPlayerCant.play()
}
```

Figure 4.5: Affinity >= 30, perform normal action

This part of the code is executed when the Digimon is at it max affinity but the user continues to perform normal activities. Same as above, the label will be updated and audio feedback will be provided.

Situation 2: Duplicate Actions



```
} else if (action == lastAction) {

  promptLabel.text = s"Can't $action twice in a row. Choose another action."
  val audioURLCant = getClass.getResource( name = "../audio/musicCant.mp3").toString
  val mediaPlayerCant = new MediaPlayer(new Media(audioURLCant))
  mediaPlayerCant.play()
```

Figure 4.6: Duplicate actions

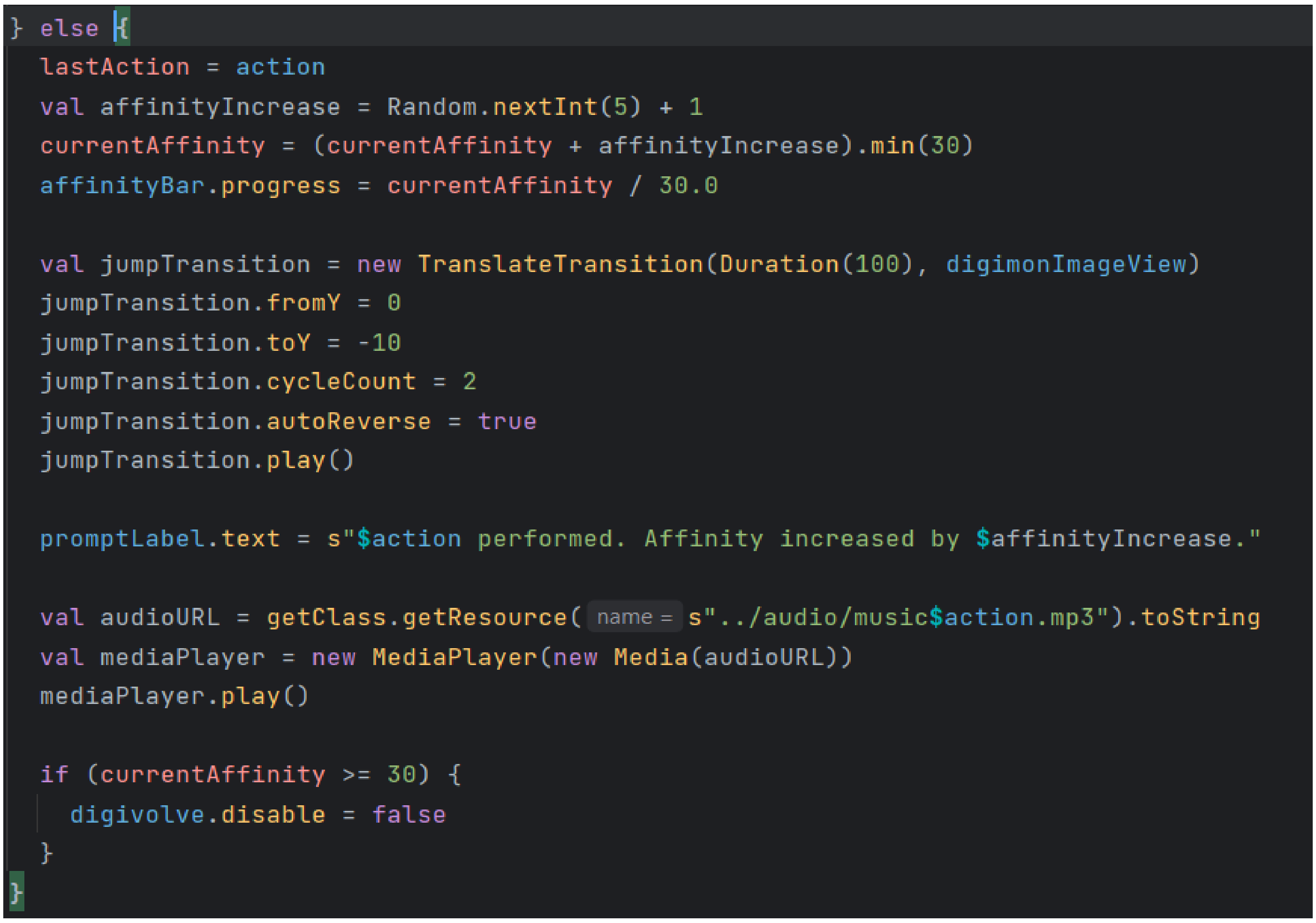
This part of the code focuses on dealing with duplicate actions. It is executed when the condition is met.

Checking for Last Action: The code checks if the current action being attempted (action) is the same as the last action that was performed (lastAction). This is to prevent the player from performing the same action consecutively.

Prompt Label Update: If the player is trying to perform the same action twice in a row, the promptLabel is updated with a message indicating that the action cannot be performed. The message informs the player that they can't perform the same action twice consecutively, and they should choose a different action.

Audio Playback: Similar to the previous code block, the code initializes and plays an audio clip using the MediaPlayer class. The audio clip is associated with the scenario of attempting to perform the same action twice in a row. This audio feedback reinforces the message presented in the promptLabel and provides auditory confirmation to the player.

Situation 3: Normal Actions



```
} else {
  lastAction = action
  val affinityIncrease = Random.nextInt(5) + 1
  currentAffinity = (currentAffinity + affinityIncrease).min(30)
  affinityBar.progress = currentAffinity / 30.0

  val jumpTransition = new TranslateTransition(Duration(100), digimonImageView)
  jumpTransition.fromY = 0
  jumpTransition.toY = -10
  jumpTransition.cycleCount = 2
  jumpTransition.autoReverse = true
  jumpTransition.play()

  promptLabel.text = s"$action performed. Affinity increased by $affinityIncrease."

  val audioURL = getClass.getResource( name = s"../audio/music$action.mp3").toString
  val mediaPlayer = new MediaPlayer(new Media(audioURL))
  mediaPlayer.play()

  if (currentAffinity >= 30) {
    digivolve.disable = false
  }
}
```

Figure 4.7: Normal action

For the last part of the if-else statement, it is executed whenever the user performs an activity that is not a duplicate or when the affinity is 30. Main components are explained below:

Updating Last Action: The lastAction variable is updated with the current action performed by the player. This is important for preventing consecutive identical actions.

Affinity Increase: A random value between 1 and 5 is generated using Random.nextInt(5) + 1. This value represents the increase in affinity points due to the performed action. The currentAffinity is updated by adding the affinity increase, capped at a maximum value of 30.

Affinity Bar Update: The affinityBar's progress property is updated to reflect the new affinity level. The affinity value is divided by 30.0 to get a value between 0 and 1 for the progress bar.

Jump Animation: An animation is triggered using the TranslateTransition class to create a jumping effect for the digimonImageView. The image moves up by a small distance and then returns to its original position, creating a jumping animation effect

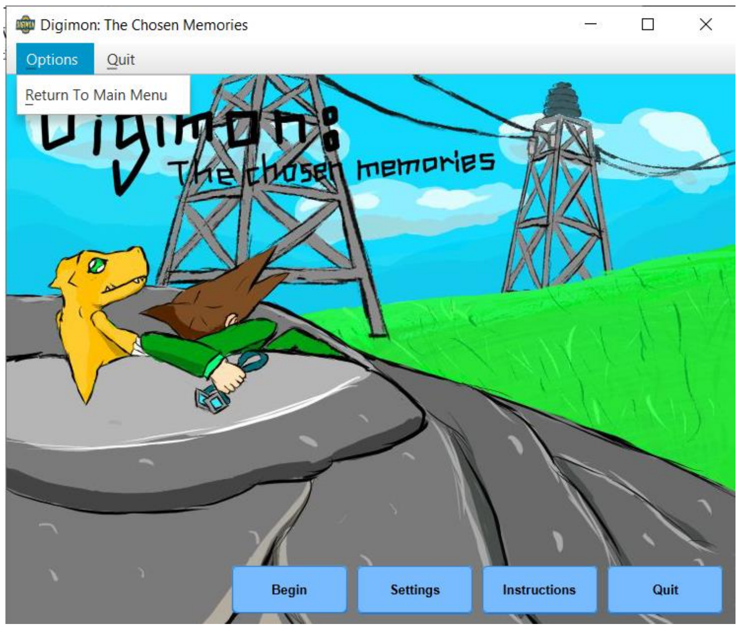
Prompt Label Update: The promptLabel is updated with a message indicating that the action was performed and specifying the increase in affinity points due to the action.

Audio Playback: An audio clip associated with the performed action is played using the MediaPlayer class. The audio clip provides auditory feedback to match the action performed.

Digivolve Button Activation: If the currentAffinity reaches or exceeds the maximum value of 30, the "Digivolve" button (digivolve) is enabled. This allows the player to proceed with the Digivolve action, as their Digimon's affinity is now high enough.

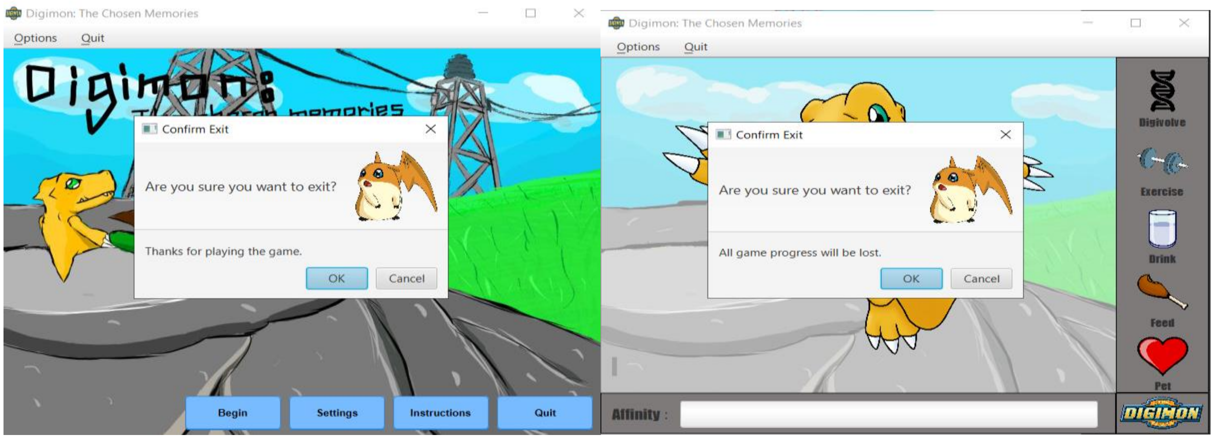
5.0: Game Design Details

While designing this program, a lot of effort are put into the small details. In this section all these small details will be mentioned.



Mnemonic parsing function is included to improve user experience.

Figure 5.0: Mnemonic Parsing



Are you sure you want to exit?

Thanks for playing the game.

Are you sure you want to exit?

All game progress will be lost.

Figure 5.1: Different Exit Messages and Icon

Different exit messages in different situations, custom icon is also used instead of default icon.

<u>**Reflection**</u>

Overall, this project was a very fun and enriching process for me. The topic I picked was interesting from my perspective and the techniques we learnt from the subject were best suited for this type of program. It is my first time delving into various aspects of game development using ScalaFX, including UI design, CSS styling, media integration, and controller logic. From designing buttons with custom CSS properties to incorporating animations, audio, and alerts, it was a really humbling and fun process. From the initial design I had in mind, I did not imagine that I will be integrating media elements such as audio and images, including handling their playback and customization in my program. However, thanks to the extensive library ScalaFx has, it all was implemented beautifully, giving users a polished and smooth experience in my opinion.

However, it would be a lie to say that I did not encounter any issues during development. Initially, it was something with the file structure, I did it purely based on my memory from tutorial classes which led to errors; then it was null pointer exceptions, which was caused by bad path referencing. Thanks to my lecturer Dr. Chin, who was willing to sacrifice this time to help go through and point out errors in my code, without him I probably would be still debugging till the cows come home. Besides that, I also did extensive debugging of the program, ridding it of any defects that persisted throughout the development. Alas, due to the exam week and deadline being close together, I in the end had to request an extension to perfect and fine-tune my code.

As for limitations, my program does not excel in sound processing. To give an example, if you were to press multiple action buttons in quick succession, the sound effects will “fight” over each other, which is not good for the user’s ears to hear. Next, this project cannot be run via SBT shell, this is due to the method used to link the resource files. After inquiring Dr. Chin, solutions were obtained but sadly not implemented due to the lack of time. For its strengths, the program is detailed in functionality, all minor details are taken note of. Also, the game incorporates audio and graphic visualization, which aids interest.

Before concluding this report, I also need to give thanks to my friends for helping in providing ideas, especially MysticK who help design the images used in the main menu and game interface. Besides that, numerous websites were visited and referenced in the making of this code, all of which will be listed in the reference list below.

References

"How Can I Show an Image Using the ImageView Component in Javafx and Fxml?" *Stack Overflow*, stackoverflow.com/questions/22710053/how-can-i-show-an-image-using-the-imageview-component-in-javafx-and-fxml. Accessed 20 Aug. 2023.

"How Do You Set the Icon of a Dialog Control Java FX/Java 8." *Stack Overflow*, stackoverflow.com/questions/27976345/how-do-you-set-the-icon-of-a-dialog-control-java-fx-java-8. Accessed 20 Aug. 2023.

"How to Add CSS to a JavaFX Project." *Www.youtube.com*, youtu.be/bvEbfqbfm4I. Accessed 20 Aug. 2023.

"JavaFX | Duration Class." *GeeksforGeeks*, 28 Aug. 2018, www.geeksforgeeks.org/javafx-duration-class/. Accessed 20 Aug. 2023.

"JavaFX 8 PauseTransition." *Www.youtube.com*, youtu.be/E6-Iv08DBWc. Accessed 20 Aug. 2023.

"JavaFX Button with Transparent Background." *Stack Overflow*, stackoverflow.com/questions/36566197/javafx-button-with-transparent-background. Accessed 20 Aug. 2023.

"JavaFX Not Accepting the Correct URL for Image Object." *Stack Overflow*, stackoverflow.com/questions/70996891/javafx-not-accepting-the-correct-url-for-image-object. Accessed 20 Aug. 2023.

"Javafx Translate Transition - Javatpoint." *Www.javatpoint.com*, www.javatpoint.com/javafx-translate-transition. Accessed 20 Aug. 2023.

"JavaFX: Inserting Image into a GridPane." *Stack Overflow*, stackoverflow.com/questions/36652453/javafx-inserting-image-into-a-gridpane. Accessed 20 Aug. 2023.

"ScalaFX API 8.0.102-R11." *Www.scalafx.org*, www.scalafx.org/api/8.0/#package.

Teck Min, Chin. *AddressApp Souce Code*.

"Uploading and Displaing an Image Using JavaFX." *Stack Overflow*, stackoverflow.com/questions/69704118/uploading-and-displaing-an-image-using-javafx. Accessed 20 Aug. 2023.

File Name	Source
musicCant.mp3	https://www.youtube.com/watch?v=bUIfMuphyeY
musicDrink.mp3	https://www.youtube.com/watch?v=WS0DuduF5IY
musicEnd.mp3	https://www.youtube.com/watch?v=m3wH9K9cDcI
musicExercise.mp3	https://youtu.be/9JThJ1anKY4
musicFeed.mp3	https://www.youtube.com/watch?v=wZGof6iUcmk
musicMC.mp3	https://youtu.be/laZusNy8QiY
musicMC2.mp3	https://youtu.be/qq-RGFyaq0U
musicPet.mp3	https://youtu.be/owDftW3-G3A
gameAgumon.png	https://www.pngwing.com/en/free-png-nfawv
gameAgumonDigivolved.png	https://www.pngwing.com/en/free-png-swydy
gameAlert.png	https://www.pngwing.com/en/free-png-zfduh
gameBackground.png	Drawn by MysticK, who does not want to share his contact
gameDigimonLogo.png	https://www.pngwing.com/en/free-png-zfdit
gameDigivolve.png	https://www.kindpng.com/imgv/hbJbmx_dna-computer-icons-genetics-clip-art-life-science/
gameDrink.png	https://www.pinclipart.com/maxpin/JmoJw/
gameExercise.png	https://www.nicepng.com/ourpic/u2t4t4i1i1u2r5w7_dumbbell-weight-clipart/
gameFeed.png	https://www.pngkit.com/bigpic/u2q8u2i1e6o0t4y3/
gameGabumon.png	https://www.pngwing.com/en/free-png-nplsl
gameGabumonDigivolve.png	https://www.stickpng.com/img/games/digimon/digimon-character-weregarurumon
gameImpmon.png	https://www.pngwing.com/en/free-png-cjobq
gameImpmonDigivolve.png	https://www.pngwing.com/en/free-png-ndqso
gameLogoInterface.png	https://www.pngwing.com/en/search?q=digimon+Adventure+Tri
gamePalmon.png	https://www.pngwing.com/en/free-png-zfdit
gamePalmnDigivolve.png	https://www.pngwing.com/en/free-png-nuxpe
gamePet.png	https://pixabay.com/vectors/heart-symbol-red-shape-sign-love-24037/
gameTitle.png	Drawn by MysticK, who does not want to share his contact