Recreating Street Fighter arcade game in Java

Drew Alexander Jonation Berry

Registration Number: 2000878

BSc Computer Science

Supervisor: Yunfei Long

Second Assessor: Faiyaz Doctor

Acknowledgements

This entire project is based off the Street Fighter franchise created by Capcom, this means that the game concept and all assets, such as characters, background images, and music, are property of Capcom. My game will not be distributed or used to generate any revenue.

Special thanks to my supervisor, Dr Yunfei Long, and my second assessor, Dr Faiyaz Doctor, for the advice they have given during meetings and the interim oral interview. This project would not have been completed to the quality that it has without such guidance.

Abstract

Street Fighter is a classic two-dimensional PVP fighting arcade game made by Capcom in the 1980s, where the aim is to defeat the opponent by any means necessary before you are defeated. This project aimed to faithfully recreate this beloved game, while also giving my own interpretation. This was to be made from the ground up in Java, using various tools such as Java Swing. The game includes a full roster of characters which will fight with many different attacks, in locations familiar to veteran players. Battle through multiple game modes against the computer or face off against a friend.

Contents

1) Problem Summary……………………………………………………………………………………………………………………….. 5

2) Problem Analysis………………………………………………………………………………………………………………………….. 7

3) Class Definitions…..………………………………………………………………………………………………………………………. 8

3.1) Main Classes…………………………………………………………………………………………………………………. 8

3.2) Utility Classes……………………………………………………………………………………………………………… 24

4) Implementation…………………………………………………………………………………………………………………………. 32

4.1) Gameplay…………………………………………………………………………………………………………………… 32

4.2) Controls and Attacks…………………………………………………………………………………………………… 35

4.3) AI………………………………………………………………………………………………………………………………… 42

4.4) Graphics……………………………………………………………………………………………………………………… 43

4.5) Window and Menus……………………………………………………………………………………………………. 44

` 4.6) Music and Sounds………………………………………………………………………………………………………. 45

5) Testing……………………………………………………………………………………………………………………………………….. 46

6) Project planning…………………………………………………………………………………………………………………………. 47

7) Conclusions……………………………………………………………………………………………………………………………….. 48

8) References…………………………………………………………………………………………………………………………………. 49

9) Appendices………………………………………………………………………………………………………………………………… 50

9.1) Appendix A: Risk Register……………………………………………………………………………………………. 50

9.2) Appendix B: Character Sprite Sheets…………………………………………………………………………… 51

9.3) Appendix C: Open Day Poster…………………………………………………………………………………….. 52

9.4) Appendix D: Project Links…………………………………………………………………………………………… 53

Problem Summary

The aim of this project was to recreate the classic arcade fighting game Street Fighter in Java. I have been a fan of the franchise for many years and was quite interested in the development process of such games, so choosing this as my capstone project seemed right. My game should be a faithful recreation but offer my own interpretation, and was built from the ground up, which will show the extent of my technical abilities.

I should reiterate that all characters, background images, music, and sounds are property of Capcom. I do not claim to own these and will not be distributing this game.

The development of this project was completed in Intellij, with the use of Gitlab and Jira for project organisation.

Research

The Street Fighter franchise, first seen in 1987, is a series of fighting games originally played on arcade machines, but eventually made their way over to game consoles. Developed by Capcom, the games allow two players to choose from a roster of characters and fight in one-on-one battles with the goal of defeating each other's character.

Over the years, the franchise has grown to become one of the most popular and influential fighting games franchises in the history of video games. Street Fighter II, released in 1991, is often credited with popularizing the fighting game genre and introducing many of the now-famous features such as special moves, combos, and multiple playable characters.

The original game, known simply as “Street Fighter”, has spawned numerous sequels and spin-offs, including a popular anime and manga series, and has had a lasting impact on popular culture. The game has been referenced or featured in countless films, TV shows, and other forms of media, cementing its place as a cultural icon.

The Street Fighter franchise has also had a significant impact on the competitive gaming scene, with many tournaments and events focused on the game and its sequels. The game's competitive scene has given rise to many legendary players and continues to attract a dedicated fanbase and community of players to this day.

Players control their chosen character using a combination of joystick and button inputs to execute a variety of moves, including punches, kicks, throws, and special moves. The number and layout of these buttons can vary depending on the specific arcade cabinet and game version, but typically, there are six buttons in total: three punch buttons (light, medium, and heavy) and three kick buttons (also light, medium, and heavy).

Special moves are performed using specific button combinations and joystick movements, and often require precise timing and execution to be performed successfully. For example, to perform Ryu's Hadoken, the player must input the following commands: press the down on the joystick, then move the joystick quarter circle towards the right, and finally press the Punch button. This sequence of input commands will cause Ryu to execute the Hadoken move, which launches a ball of energy at the opponent.

In addition to special moves, players can also perform combos, which involve chaining together a series of attacks to create a more powerful and devastating sequence of moves. Many characters also have powerful super and ultra moves that can be performed using a special meter that fills up over time as the player performs offensive actions and takes damage.

To block an attack in Street Fighter, the player must hold back on the joystick to move away from the opponent and then press and hold the back button on the controller. This will cause the character to enter a defensive stance and block any incoming attacks. However, some attacks are unblockable and can only be avoided by jumping or using specific character moves. It's important to note that blocking still results in chip damage, meaning the player will still take some damage from the attack, but not as much as if they were not blocking at all. Additionally, if a player blocks for too long, they will become vulnerable to throws and other moves that can go through blocks.

Stun and knockback are mechanics used to disrupt an opponent's offense and create opportunities for counterattacks. Stun occurs when a player takes a certain amount of damage or is hit by a series of rapid attacks, causing them to lose control of their character temporarily. While stunned, the player is vulnerable to further attacks and cannot block or perform their own moves. The length of time the player remains stunned depends on the character and the attack, but typically lasts for a few seconds. Knockback occurs when a player is hit by an attack with sufficient force or damage, causing them to be knocked away from the attacker. This can create distance between the players and give the attacker an opportunity to follow up with another attack. The amount and direction of knockback depends on the type and strength of the attack, as well as the character's weight and position.

Characters each have a unique moveset, meaning that they have their own set of moves and abilities that distinguish them from the other characters in the game. This includes normal attacks, special moves, throws, and super combos. Each character also has their own movement speed, jump arc, and other physical attributes that affect their gameplay. While some moves or abilities may be similar between different characters, there are always subtle differences that make them unique. These differences can be in the inputs required to execute the move, the range of the attack, the speed of the move, the amount of damage it deals, and more. The unique movesets of each character allow players to customize their playstyle and choose a character that suits their preferences and strengths.

For additional understanding, this video by (World of Longplays, 2013) shows how Street Fighter II plays.

During the summer of 2022, in the lead up to the start of this project, I conducted much research into the franchise to get a feel for the games so that I could create a game with similar style and physics, which would feel authentic. This research included playing some of the games on consoles and a computer but was unfortunately unable to play on an actual arcade cabinet. This allowed me to get a feel for how it plays. I also watched countless videos, like the one previously referenced, which gave me more knowledge on characters, and their move sets, and the pacing of gameplay.

Problem Analysis

To recreate Street Fighter to a good quality, I aimed to create a gameplay loop that would feel like the classic games. This would require allowing two players to fight each other to the death in a “best of 3” format. Each player would choose a character from a roster of fighters, which each have unique move sets and animations to fight with, and attacks would have to land correctly with the proper reactions (taking damage, being stunned etc.). Players would be able to dodge or block attacks to avoid damage, or even counterattack, and outlive their opponent. Players would choose one of three game modes to play: story mode would allow the player to fight against an AI, arcade mode would provide classic pvp gameplay, and training would allow players to become familiar with the controls. To navigate between these game modes, a menu system would be implemented and would offer an options menu where the control schemes could be changed. Players would choose their fighter and the location to fight in through these menus. Finally, the game would have the retro graphics and sound effects that the original games are famous for.

These were the aims of the project and as a result, these objectives were decided upon during Challenge Week:

1. Recreate the main gameplay loop of Street Fighter.
   1. Two fighters face each other and fight until one runs out of health or time runs out.
   2. Played in a “Best of 3” format.
   3. Each unique fighter has a unique move set to fight with.
   4. Needs to have correct collision detection for when one player strikes the other.
   5. Players can block attacks to avoid the damage that would have been dealt.
2. Multiple game modes
   1. Story, arcade, and training modes.
   2. Story will include sections of dialogue between characters.
   3. Arcade is the classic game where you just fight between players.
3. Playable for player vs player and player vs AI.
   1. AI will be used in Story mode.
   2. AI shouldn’t be too complex.
4. Implement a menu system.
   1. Allows the player to choose a game mode, a fighter and choose background.
   2. Includes a menu to configure control scheme.
5. Many fighters to choose from.
   1. 6-8 of the main cast of Street Fighter.
6. Retro graphics and sound effects.
   1. Characters are represented as sprites.
   2. Backgrounds will be taken from the original games.
   3. This also includes the menus, which should fit the style.

Class Definitions

In this section, all classes, and their methods and attributes, will be described to provide an understanding of the program. This will lead into the implementation section which describes how these classes work together to create the game.

Main classes

Game……………………………………………………………………………………………………………………………………………….. 9

GUI…………………………………………………………………………………………………………………………………………………. 13

Player…………………………………………………………………………………………………………………………………………….. 16

AI……………………………………………………………………………………………………………………………………………………. 19

PlayerStates…………………………………………………………………………………………………………………………………… 20

PlayerFrames…………………………………………………………………………………………………………………………………. 21

Controls…………………………………………………………………………………………………………………………………………. 22

MenuNav……………………………………………………………………………………………………………………………………….. 23

Game

This class controls the enitre flow of the game, whether that be in menus or during the actual gameplay. The gameplay loop in this class consists of updating each player’s position, actions, and health, waiting several milliseconds, and then checking for any win or lose conditions. Each cycle of this is defined as a frame. The Game class holds instances of Player, GUI, Constants, Window, MenuNav, and Controls, which can be accessed by other classes for various purposes.

**Attributes**

**Public Access**

* gui
  + Type: GUI
  + Used to call the repaint method of the GUI after every frame of the game loop.
* player1 and player2
  + Type: Player
  + These hold both players of the game. The Game class directly uses these to call the update method of each player at every iteration of the game loop. The GUI class uses these attributes to display each player’s character sprite and health. The Controls class uses these attributes to modify each player's data when certain keys are pressed or released, such as changing a player's attacking state to true when an attack key is pressed.
* window
  + Type: Window
  + This holds the entire game window so that the Game class can add the menu navigation and game controls key listeners to the window when transitioning between menus and gameplay. This is so that the control schemes for both do not overlap each other and break the program.
* c1 and c2
  + Type: Controls
  + These are added as key listeners to the window when in the gameplay and are removed after returning to menus.
* menuNav
  + Type: MenuNav
  + This is added as a key listener to the window when in the menus and is removed after starting any gameplay.
* constants
  + Type: Constants
  + Holds all constant values and the constructor loads all images for the program, this constructor is called in the Game constructor.
* sprites
  + Type: ArrayList<PlayerSprite>
  + Holds each character’s sprite data for use in the GUI class to display the characters.
* timeLeft
  + Type: int
  + Used by the GUI class to display a timer at the top of the screen. This should start as 60 and tick down to 0. If this hits 0 before either player wins, “TIME UP” is displayed on screen and the current match ends.
* winner
  + Type: int
  + Used by the GUI class to show whether player 1 or player 2 have won. Default is 0. If player 1 wins the match, this will be 1, whereas if player 2 wins, this will be 2.
* round
  + Type: int
  + Keeps track of the current round number for display in GUI.
* roundStart
  + Type: boolean
  + Is true at the start of the round, so that the GUI can show the round start dialogue.
* p1Wins and p2Wins
  + Type: int
  + Hold the number of rounds won by each player. When this number reaches two, the corresponding player wins the game.
* start
  + Type: boolean
  + This determines whether the user is on the title screen.
* menu
  + Type: String
  + Holds which menu the user is currently viewing.
  + Can the following values: “story”, “arcade”, “main”, “options”, “charSelect”, “training”.
* menuChoice
  + Type: int
  + This holds the current menu option on any menu, so that the program knows which menu option the user wants to choose. This attribute is updated in MenuNav.
* p1 and p2
  + Type: int
  + These hold the selected characters for the upcoming fight. They are used to assign the character sprites.

**Private Access**

* startTime
  + Type: long
  + This records the start time of the current match and is then used to find the time left of the game for the timer at the top of the screen.

**Methods**

**Public Access**

Constructor

* Parameters: none
* Creates a new GUI instance, which is assigned to gui, and then sets start to true.

menus

* Parameters: none
* Return type: void
* While the user is in the menus, this method loops and repaints the GUI. When the user navigates to the gameplay, this loop is broken, and the two player sprites are assigned based on the user input. Lastly, a call to gameLoop is made.

**Private Access**

gameLoop

* Parameters: none
* Return type: void
* First this method calls resetGame, then it loops until the match is won or lost. This loop includes a call of updateGame, sleeping for a frame worth of milliseconds and then a call of checkWinLoseConditions.

updateGame

* Parameters: none
* Return type: void
* Calls both players update method, followed by recalculating timeLeft by subtracting startTime away from the current time and subtracting that from the max time, and finally calls the repainting the GUI.

resetGame

* Parameters: none
* Return type: void
* Starts the music clip for this new match, then creates two Players, sets startTime to the current time, and sets winner to its default.

checkWinLoseConditions

* Parameters: none
* Return type: void
* Checks if time is zero or either player's health is 0. If any of these conditions are true, add to the winning players win tally and end the music clip.

setPlayerSprites

* Parameters: none
* Return type: void
* Sets the current sprites for the characters in this current game.
* Works with the MenuNav class to do this.

**Main Method**

This creates a Game instance. Then, a call is made to the menus method of Game. This is a simple main method as the setup of the game is handled in the constructor as stated previously.

GUI (Extends JComponent)

This class’s purpose is to display all visual elements of the program. It extends JComponent to be able to override the paintComponent method and can be used in Window which extends JFrame.

**Attributes**

**Private Access**

* game
  + Type: Game
  + Used to access certain variables, including both players' data, that are used in the display.
* bg
  + Type: Image
  + This is the current background for the gameplay loop, not the menu.
* p1 and p2
  + Type: Image
  + This is the current player 1 image, which gets changed in the paintComponent method.
* bgTransf
  + Type: AffineTransform
  + Holds the transformation that will be applied to bg to make it fit the dimensions of the Window.
* player1Sprite and player2Sprite
  + Type: PlayerSprite
  + Hold the current sprite for each player so that they can be displayed.
* p1Config and p2Config
  + Type: int
  + Hold the index of the scaling data needed for each animation. This index refers to the 2d int arrays in the Constants class. See graphics section of the implementation for further explanation.

**Methods**

**Public Access**

Constructor

* Parameters: Game
* Only sets the game attribute to the given argument, which is always the only instance of Game.

paintComponent (override)

* Parameters: Graphics
* Return type: void
* Uses the game.menu attribute to display the correct menu using one of the menu methods in this class (see below) or displays the gameplay with the use of the gameplay method (see below). Is called via the use of repaint as it is an overwritten method from JComponent.

getPreferredSize()

* Parameters: none
* Return type: Dimension
* Returns the WINDOW\_DIMENSION constant from the Constants class. A method of this name must be included when extending JComponent.

**Private Access**

scaleBackground

* Parameters: none
* Creates an AffineTransform that is assigned to bgTransf and scales it to the ratio of the Window’s dimension to bg’s dimensions. So that when the background is drawn in paintComponent, it fits the window.

menus, charSelect, mainMenu, storyMenu, arcadeMenu, trainingMenu, optionsMenu

* Parameters: Graphics2D
* Return type: void
* These all create their respective menu, creating a background with text/images. These are used by paintComponent so that it is more compact/tidy.

player1Sprite and player2Sprite

* Parameters: Graphics2D
* Return type: void
* Determine which frame of an animation each player is currently on to display them. These are used by paintComponent so that it is more compact/tidy.

overlay

* Parameters: Graphics2D
* Return type: void
* Creates the overlay of the gameplay, this includes health bars, names, character portraits, timer, and game over messages. This is used by paintComponent so that it is more compact/tidy.

hitbox

* Parameters: Graphics2D
* Return type: void
* Draws a red square around each player that symbolises their hitbox.

text

* Parameters: int, int, int, String, Color, Graphics2D
* Return type: void
* Draws the given text at the given location.

Player

This class holds all data related to one player, not both, while also controlling most of the manipulation of this data, such as moving the player or updating an animation.

**Attributes**

**Public Access**

* other
  + Type: Player
  + This is a reference to the other player so that collisions can be detected.
* side
  + Type: String
  + Determines which side the player is on. “Left” refers to the player with their back to the left-hand side of the Window, whereas “Right” refers to their back being towards the right-hand side. The other player has a side attribute which is opposite to this one.
* initialY
  + Type: int
  + This is the y coordinates for use when a player is made. Also, it is used to stop the player falling through the floor when jumping, e.g. if the player has y coordinate greater than initialY, their y coordinate is reset to initialY.
* health
  + Type: int
  + Holds the players current health. Default is 100 and decreases via detectHit.
* radius
  + Type: int
  + Used to detect collision, in terms of walking into the other player.
* states
  + Type: PlayerStates
  + Holds all the states of the player, these are manipulated in many places, for example in Controls when a key is pressed or in adjustFrames when an animation ends.
* frames
  + Type: PlayerFrames
  + Holds all the current animation frame data of the player, these are added to and reset in adjustFrames, and are used in GUI to find the current animation image.
* position
  + Type: Vector
  + A Vector that holds the x and y coordinate of the player
* velocity
  + Type: Vector
  + A vector that holds the x and y speeds that the player is moving, only changed when moving or jumping.
* hitboxWidth and hitboxHeight
  + Type: int
  + Determine the dimensions of a players hitbox
* attackHitboxSize
  + Type: int
  + Holds the width/height of the square attack hit box.

**Methods**

**Public Access**

Constructor

* Parameters: none
* Sets all attributes to their default value, which includes calls to resetFrames and resetAttackStates.

setOther

* Parameters: Player
* Sets other to the given argument.

update

* Parameters: none
* First, add to the position based on the velocity. Then if jumping, add to the y direction of the players velocity attribute so that they fall back down. Then if the player has fallen below the initialY, reset them to initialY, reset the velocity in the y direction, and reset both the jumping state and frames. Then make sure the character hasn’t walked off screen, otherwise reset to the boundaries. Then, check for collisions between the bodies of both characters using radius. Finally, call detectSideSwap and adjustFrames.

**Protected Access**

adjustFrames

* Parameters: none
* Adds one to the current states respective frame attribute, then checks if the animation is finished, if so, change the state to false and set another to true, situation depending.

decideHit

* Parameters: int, int, int, int, int, int, int
* Decides whether an attack is ready to hit or not, based on what frame the animation is in.

detectHit

* Parameters: int, int, int, int
* Checks for any collisions with an attack hitbox and the other player. When an attack lands, damage is inflicted, and a stun/knockdown effect is applied.

detectSideSwap

* Parameters: none
* If the other player is on the other side of this player, make the turning state true.

AI (extends Player)

This class acts as an AI player and holds all its attributes. See the Player section of this chapter for all attributes or functions not mentioned here.

**Methods**

**Public Access**

Constructor

* Parameters: none
* Calls the super constructor with the argument “Right”.

update

* Parameters: none
* Return type: void
* Calls chooseAction() and then calls the super update() method.

**Private Access**

chooseAction

* Parameters: none
* Return type: void
* Uses random numbers to determine the AI player’s actions.

PlayerStates

This class is used to determine which state the player is currently in.

**Attributes**

**Public Access**

* All boolean attributes
  + Examples: idle, jumping, standingMedium, crouchedForwardHeavy etc.
  + When true, the player is currently in the state.
* standingLight and jumpingLight
  + Type: int
  + Hold a 0 when not in the state.
  + Hold a 1 when attacking once.
  + Hold a 2 when attacking twice.
* stunned
  + Type: int
  + Holds a 0 when not in the state.
  + Holds a 1 when lightly stunned.
  + Holds a 2 when heavily stunned.
  + Holds a 3 when crouched and stunned.

**Methods**

**Public Access**

Constructor

* Parameters: none
* Sets all states to false and then makes idle true, which creates the starting state of the player.

resetAttackStates

* Parameters: none
* Return type: void
* Sets all the attacking states to false.

resetStates

* Parameters: none
* Return type: void
* Sets all states to false.

PlayerFrames

This class holds all the frame data for the animations of a character. See the Graphics section of the Implementation chapter for an explanation of how this data is used.

**Attributes**

**Public Access**

* jumping, idle, crouched, walking, attacking, turning, stun, knockdown, death, block
  + Type: int
  + These hold the number of frames a current animation has lasted for.

**Methods**

**Public Access**

Constructor

* Parameters: none
* Sets all the attributes to 0.

resetAttackFrames

* Parameters: none
* Return type: void
* Sets the attacking attribute to 0.

resetAttackFrames

* Parameters: none
* Return type: void
* Sets all the attributes to 0.

Controls (extends KeyAdapter)

This class allows the user to perform certain actions based on my control scheme. Each button press or release only changes some attributes of the Game class or its two Players.

**Attributes**

**Private Access**

* game
  + Type: Game
  + Holds the Game instance so that the two Players it holds can be manipulated.
* type
  + Type: final int
  + Holds either a 1 or 2, so that certain actions only work for the correct player
* xSpeed
  + Type: final int
  + This is the speed that the player will move on the x axis when pressing the left or right button. This is added to the velocity of the Player.
* ySpeed
  + Type: final int
  + This is the speed that the player will initially move in the y direction when jumping

**Methods**

**Public Access**

Constructor

* Parameters: Game, int
* Sets the game and type attributes to the values from the given arguments. Sets xSpeed and ySpeed to certain values.

keyPressed / keyReleased

* Overrides the existing methods of the same names in the KeyAdapter.
* Using my control scheme, the given key event matches to a certain block in a switch statement, and then certain Player attributes are checked and manipulated.
* Refer to the Controls and Attacks section of the Implementation chapter for more understanding and examples.

MenuNav (extends KeyAdapter)

This class allows the user to navigate the menus.

**Attributes**

**Private Access**

* game
  + Type: Game
  + Allows this class to manipulate certain attributes in the Game class.
* charSelectCount
  + Type: int
  + Counts the number of characters selected so that no more than two are chosen in the character select menu.

**Methods**

**Public Access**

keyPressed

* Overrides the existing method of the same name in the KeyAdapter.
* Allows the use of the up and down arrow keys to navigate menu options, and the enter and escape keys for transitioning to and from menus.
* Allows the use of the number keys to choose a character.
* Refer to the Window and Menus section of the Implementation chapter for more understanding and examples.

Utility classes

Window…………………………………………………………………………………………………………………………………………. 25

Constants……………………………………………………………………………………………………………………………………….. 26

Vector……………………………………………………………………………………………………………………………………………. 27

SoundManager………………………………………………………………………………………………………………………………. 28

ImageManager………………………………………………………………………………………………………………………………. 29

PlayerSprite……………………………………………………………………………………………………………………………………. 30

Window

This class extends JFrame and isn’t overly complex, just adds a few attributes to the JFrame so that the window attributes are all in one place.

**Attributes**

**Public Access**

* gui
  + Type: Component
  + Takes the GUI from Game to be added to the window so it is shown.

**Methods**

**Public Access**

Constructor

* Parameters: Component
* Calls the superclass’s constructor and then sets gui to the argument. Then packs the gui into the window, and sets resizable to false, visible to true, and default close operation to exit on close.

Constants

Holds all values that are constant across the whole program. Each attribute is static so can be accessed easily from anywhere.

**Attributes**

* WINDOW\_HEIGHT and WINDOW\_WIDTH
  + Type: int
  + Hold the integer values that make up the windows dimensions.
* WINDOW\_DIMENSION
  + Type: Dimension
  + Made from WINDOW\_HEIGHT and WINDOW\_WIDTH, gives the window its dimensions.
* BACKGROUND\_IMAGES
  + Type: ArrayList<Image>
  + Holds all background images.
* FIGHTER\_SHEETS
  + Type: ArrayList<Image>
  + Holds all fighter sprite sheets.
* All sprite configuration 2d arrays
  + Type: int[][]
  + Each character has three 2d arrays that configure their sprites for all animations.
  + First array is used to locate the animation frames in the sprite sheet.
  + The other two arrays scale these animation frames to the correct size and location. One array is for when the character is on the left-hand side, the other is for the right-hand side.

**Methods**

**Public Access**

Constructor

* Parameters: none
* Loads all the needed images and adds them to the correct attributes.

Vector

An instance of this acts as a coordinate pair and this class has methods for manipulating the coordinates.

**Attributes**

**Public Access**

* x and y
  + Type: double
  + Act as x and y coordinates

**Methods**

**Public Access**

Constructor

* Parameters: double, double
* Sets x and y to the arguments.

set

* Parameters: double, double
* Sets x and y to the arguments.

equals

* Parameters: Object
* Returns a boolean based on if x equals the argument’s x and y equals the argument’s y.

add

* Parameters: double, double
* Adds the first argument to x and the second argument to y.

addScaled

* Parameters: Vector, double
* Adds the vector argument multiplied by the second argument to this vector.

subtract

* Parameters: double, double
* Takes the first argument away from x and takes the second argument away from y.

dist

* Parameters: Vector
* Math.hypot on this vector and another to find the distance between the two.

SoundManager

Manages all sound and music clips needed for the program.

**Attributes**

**Public Access**

* music
  + Type: ArrayList<Clip>
  + Holds all music for the game.
* sounds
  + Type: ArrayList<Clip>
  + Holds all sounds for the game.

**Methods**

**Public Access**

getClip

* Parameter: String
* Return type: Clip
* Uses the string as the file name and loads the file as a Clip, using AudioInputStream. This Clip is returned.

playClip

* Parameter: Clip
* Plays the given clip from the beginning.

stopClip

* Parameter: Clip
* Stops the given clip.

ImageManager

Manages the loading of images.

**Attributes**

**Public Access**

* path
  + Type: static String
  + Holds the file path to the images folder.
* ext
  + Type: static String
  + Holds the type of image file that needs to be loaded, in this instance “.png”.

**Methods**

**Public Access**

loadImage

* Parameters: String
* Throws: IOException
* Uses the argument as the file name and uses ImageIO to return a BufferedIamge.

PlayerSprite

This class holds all the images and configuration arrays for a specific character, and aides in the display of these images.

**Attributes**

**Public Access**

* sprite
  + Type: ArrayList<Image>
  + Holds all the animation frames for the character.
* portrait and name
  + Type: Image
  + Hold the images for the character portrait and name.

**Private Access**

* sheetConfig
  + Type: int[][]
  + This is set to specific character’s sprite array from the Constants class.
  + So, it holds all the configuration data for finding all animations on the sprite sheet.
* leftConfig and rightConfig
  + Type: int[][]
  + These are set to specific character’s left and right configuration arrays from the Constants class.
  + So, they hold all the configuration data for the scaling of animations.

**Methods**

**Public Access**

Constructor

* Parameters: int
* Sets all the attributes to values for the character, determined by the argument. This includes adding all the sub images of the sprite sheet to the sprite ArrayList.

drawLeftSprite

* Parameters: int, int, int, Graphics2D
* Return type: void
* Scales the correct image and draws it at the given coordiantes.
* This method is used for characters on the left-hand side of the screen.

drawRightSprite

* Parameters: int, int, int, Graphics2D
* Return type: void
* Scales the correct image and draws it at the given coordiantes.
* This method is used for characters on the right-hand side of the screen.

Implementation

This chapter will explain how I implemented my Street Fighter game and how the classes work with one another. Please refer to the previous chapter if any keywords are misunderstood.

Gameplay

To begin this section, I will detail how the players exist in the space/window that I have created.

In Java Swing, all x and y values for graphical components are computed from the top left corner of the window. This means that as the x values increase, the component moves right along the window, and as the y values increase, the component moves down the window. Due to this, all x and y values for both players are computed in the same way, to allow for correctly placed character sprites and correct player interactions. This leads me into detailing how the players exist in the space/window that I have created.

Each Player instance has a position Vector, this determines where in the window the player is currently located. The x and y values determine the top left point of a character sprite and are instantiated at default values that create the spawn location for the fighters. Each Player also has a velocity Vector. The x and y values of this velocity signify the rate of change in each direction of the position vector. This vector is manipulated by the Controls class, when the movement keys are pressed, the x value may be set to a positive or negative number depending on the direction of movement, or the y value may be increased to create a jump action. Then, in the update() method of the Player class, this velocity is added to the position to move the player, and when any movement is complete, the velocity is reset to zero. Finally, velocity is also used in the stun and knockback of a player. In the detectHit() method of the Player class, when a player is successfully hit and is stunned or knocked down, the velocity is set to certain values to push the player back. When this stun or knockback is finished, the velocity is reset to zero.

The Player class has a few attributes to determine what state the player is in or is about to change to. These include health, side, states, and frames. Health simply denotes how many health points the player has left, it is defaulted to 100 (may be subject to change), when falling below 0, the player dies, resulting in a death animation and the other player wins the round. This is checked by the CheckWinConditions() method of the Game class at every iteration of the game loop.

Side simply denotes which side the player is currently on. This value is used to correctly orient the player’s sprite, as well as contributing to the detection of hits in detectHit() and to any velocity given in the x direction.

The states attribute is of type PlayerStates which holds a series of boolean values that determine which action state the player is in, for example whether they are idle or attacking. This is the main foundation of the gameplay and is used in many places in the code, mostly in Player, GUI and Controls. A player will only be able to do certain actions while in certain states, such as while idle, a player can attack, crouch or block, however while knocked down, a player cannot do anything until the knockdown ends. This is how Controls uses the states, any key press has certain states that it cannot act in. The GUI class uses this states attribute to show the correct animation on screen, by a series of conditional statements.

Finally, the frames attribute is an instance of the PlayerFrames class, which holds integer values that indicate which frame of an animation should be shown. These values are manipulated in the adjustFrames() method of Player, after the correct PlayerFrames attribute is found, via conditional statements that use the states attribute, the integer value is increased by one and certain conditions are checked to determine whether the animation is complete, or a hit detection should be completed. When an animation finishes, the specific state is set to false and an appropriate state is set to true, for most examples this will be the idle state that becomes true. Also, the frames are reset to 0. The GUI class also uses this frames attribute to show the correct frame of an animation on screen. The frames and states attributes are constantly used together in this program.

Now that the fundamentals have been discussed, I can explain the main gameplay loop. The Game class holds all the relevant data needed for the Game and controls the whole flow of the program (this includes menus, but they will be discussed later). The gameplay loop consists of a while loop that continues to iterate until either player wins the best of 3. Players win a round by either defeating the opponent or having more health than the opponent when the timer runs out. The gameplay loop can also be broken by pressing the escape key and you are returned to the main menu. Each iteration of this loop consists of updating each player, setting the time left and repainting the GUI, followed by waiting for 20 milliseconds and then checking the win/loss conditions. The wait of 20 milliseconds creates a rate of execution, which could be considered as a frame rate or refresh rate.

The updating of players happens in the update() method in the Players class. Firstly, the velocity is added to the position so that the player moves appropriately. Next, the velocity in the y direction is increased, only when the player is jumping or being knocked down, this acts as gravity so that the player won’t fly off the screen. Following this, the player’s y position is compared to the initialY value, larger values are reset to initialY and sets the velocity in the y direction to 0, this creates a “floor” for the players to fight on and not fall off the screen. Then, the player’s x position is compared to the boundaries of the window, when outside of the boundaries, the x position is set back to the boundary value, this stops players from walking off screen. Next, body collision is checked, this is done by finding the distance between the two players’ position vectors, if this distance is less than the width of both players, they are pushed apart. Then, the update() method checks for side swaps, caused by jumping over a character. The two players’ x positions are compared, if the other player's x position is less and the current player is supposed to be on the left, change the side attribute to “right” and vice versa. Finally, adjust the frames using adjustFrames() (discussed earlier), which includes detecting any hits by attacks, in the detectHit() method.

The detectHit() method works by creating a hit box for the other player and an attack hit box, using position vectors. Each point in the attack hit box is compared to the other player’s hit box. If the attack is inside this area, the attack connects and damage is dealt, followed by a stun or knockdown effect. When a player is stunned or knocked down, they cannot perform any actions and get pushed away from the player that hit them. However, if the attacked player was moving away from the attacker, either walking\_forward or walking\_backward depending on the player’s side, when the attack lands, they block the attack and only take some chip damage, this results in no stun or knockback so that the attacked player can counterattack. Any stun or knockdown gives the player a velocity away from the player that attacked them. This gets removed by adjustFrames() when the stun/knockdown animation ends.

The checking of win/loss conditions occurs in the checkWinLoseConditions() method in the Game class. Firstly, this method checks if the time has run out, if so, it determines which player has the most health and gives them the win, adding to their win counter. Next, it checks whether a player now has two wins, meaning they have won the best of 3, if so, display the winner and navigate back to the main menu, if not, continue with the gameplay loop. If the time hasn’t run out, check if a player has no health left, if so it gives the player with health remaining the win, adding to their win counter and does the best of 3 winner check again, returning to the gameplay loop if there is no winner.

Upon starting a new game/round, certain attributes in Game need to be reset, these include the two players (player1 and player2), winner and timeLeft. Also, set each player’s other attribute to the other player and set the correct sprites to each player (determined by the character select menu, which is discussed later). Finally, set the startTime so that the timer can work correctly.

Controls and Attacks

The way a player controls their character is using a keyboard, each key press and release is captured by one of two Controls class instances held by Game. Controls extends KeyAdapter, so takes a KeyEvent from the keyboard and interprets the input. There are two instances held by Game so that, when playing arcade or training, both players can provide inputs at the same time.

Each player has 7 inputs that they can make, these are: an up action, a down action, a left action, a right action, a light attack, a medium attack, and a heavy attack.

These default to the following keys for the player that starts on the left (player1): w, s, a, d, h, j, and k. Whereas, they default to the following keys for the player that starts on the right (player2): up arrow, down arrow, left arrow, right arrow, numpad 4, numpad 5, numpad 6.

Now, I will discuss what happens when a certain key is pressed (referring to the keyPressed() method). The phrase “blocked from \_\_\_” in this section refers to being in the stunned, knockdown, or blocking state.

* Up action
  + If the player is not already jumping and is not blocked from jumping, the player’s jumping state becomes true, and the player’s velocity in the y direction becomes a large negative number to provide the lift of a jump.
* Down action
  + If the player is not already crouched and is not blocked from crouching, the player’s crouched state becomes true, and the player’s idle state becomes false.
* Left action
  + If the player is crouched, the walking\_backward state is set to true and the idle state is set to false, this allows for the crouched directional attacks.
  + However, if they are not crouched, not already moving, and not blocked from moving, the player’s x velocity becomes a large negative number, so that they will move left across the screen. Then, the walking\_backward state is set to true, and the idle state is set to false.
* Right action
  + If the player is crouched, the walking\_forward state is set to true and the idle state is set to false, this allows for the crouched directional attacks.
  + However, if they are not crouched, not already moving, and not blocked from moving, the player’s x velocity becomes a large positive number, so that they will move right across the screen. Then, the walking\_forward state is set to true, and the idle state is set to false.
* Light attack
  + All these conditions only work when the player is not blocked from attacking.
  + If the player is walking\_forward or walking\_backward, set the attacking state to true and set the player’s velocity to 0. Then, depending on the side attribute of the player, the forwardLight state or the backwardLight state is set to true.
  + If the player is crouched, set the attacking state to true. Then, if the player is walking\_forward or walking\_backward, depending on the side attribute of the player, set the crouchedLight state or the crouchedForwardLight state is set to true.
  + If the player is jumping, set the attacking state to true and the jumpingLight state to 1.
  + If the player is idle, set the attacking state to true, the standingLight state to 1 and the idle state is set to false.
  + If the player is jumpingLight or standingLight, increase the corresponding state to 2.
* Medium attack
  + All these conditions only work when the player is not blocked from attacking.
  + If the player is walking\_forward or walking\_backward, set the attacking state to true and set the player’s velocity to 0. Then, depending on the side attribute of the player, the forwardMedium state or the backwardMedium state is set to true.
  + If the player is crouched, set the attacking state to true. Then, if the player is walking\_forward or walking\_backward, depending on the side attribute of the player, set the crouchedMedium state or the crouchedForwardMedium state is set to true.
  + If the player is jumping, set the attacking state to true and the jumpingMedium state to true.
  + If the player is idle, set the attacking state to true, the standingMedium state to true and the idle state is set to false.
* Heavy attack
  + All these conditions only work when the player is not blocked from attacking.
  + If the player is walking\_forward or walking\_backward, set the attacking state to true and set the player’s velocity to 0. Then, depending on the side attribute of the player, the forwardHeavy state or the backwardHeavy state is set to true.
  + If the player is crouched, set the attacking state to true. Then, if the player is walking\_forward or walking\_backward, depending on the side attribute of the player, set the crouchedHeavy state or the crouchedForwardHeavy state is set to true.
  + If the player is jumping, set the attacking state to true and the jumpingHeavy state to true.
  + If the player is idle, set the attacking state to true, the standingHeavy state to true and the idle state is set to false.

Now, I will discuss what happens when a certain key is released (referring to the keyReleased() method). The phrase “blocked from \_\_\_” in this section refers to being in the stunned, knockdown, or blocking state.

* Down action
  + If the player is already crouched and isn’t attacking, set the idle state to true.
* Left action
  + If the character is already moving to the left and isn’t blocked from stopping moving, set the velocity in the x direction to zero, make the walking\_backward state false and reset the walking frames. Then if the player isn’t crouched make the idle state true.
* Right action
  + If the character is already moving to the right and isn’t blocked from stopping moving, set the velocity in the x direction to zero, make the walking\_forward state false and reset the walking frames. Then if the player isn’t crouched make the idle state true.

In this section, I will give visual examples of all the implemented attacks, with Ryu as the model, and will describe their effects. Any stun effects applied to a jumping character become knockdown effects.

|  |  |
| --- | --- |
| Idle (not an attack but shows the default pose) |  |
| Standing light:   * Causes a small amount of damage with a light stun effect. * Can be used twice in a row. * Must be activated from the idle state. * Activated by the light action. |  |
| Standing medium:   * Causes a moderate amount of damage with a heavy stun effect. * Must be activated from the idle state. * Activated by the medium action. |  |
| Standing heavy:   * Causes a large amount of damage with a knockdown effect. * Must be activated from the idle state. * Activated by the heavy action. |  |
| Crouched light:   * Causes a small amount of damage with a light stun effect. * Must be activated from the crouched state. * Activated by the light action. |  |
| Crouched medium:   * Causes a moderate amount of damage with a heavy stun effect. * Must be activated from the crouched state. * Activated by the medium action. |  |
| Crouched heavy:   * Causes a large amount of damage with a knockdown effect. * Must be activated from the crouched state. * Activated by the heavy action. |  |
| Crouched forward light:   * Causes a small amount of damage with a knockdown effect. * Must be activated from the crouched state, while walking towards the opponent. * Activated by the light action. |  |
| Crouched forward medium:   * Causes a moderate amount of damage with a knockdown effect. * Must be activated from the crouched state, while walking towards the opponent. * Activated by the medium action. |  |
| Crouched forward heavy:   * Causes a large amount of damage with a knockdown effect. * Must be activated from the crouched state, while walking towards the opponent. * Activated by the heavy action. |  |
| Jumping light:   * Causes a small amount of damage with a light stun effect. * Must be activated from the jumping state. * Activated by the light action. |  |
| Jumping medium:   * Causes a moderate amount of damage with a heavy stun effect. * Must be activated from the jumping state. * Activated by the medium action. |  |
| Jumping heavy:   * Causes a large amount of damage with a knockdown effect. * Must be activated from the jumping state. * Activated by the heavy action. |  |
| Forward light:   * Causes a small amount of damage with a light stun effect. * Must be activated while walking towards the opponent. * Activated by the light action. |  |
| Forward medium:   * Causes a moderate amount of damage with a heavy stun effect. * Must be activated while walking towards the opponent. * Activated by the medium action. |  |
| Forward heavy:   * Causes a large amount of damage with a knockdown effect. * Must be activated while walking towards the opponent. * Activated by the heavy action. |  |
| Backward light:   * Causes a small amount of damage with a light stun effect. * Must be activated while walking away from the opponent. * Activated by the light action. |  |
| Backward medium:   * Causes a moderate amount of damage with a heavy stun effect. * Must be activated while walking away from the opponent. * Activated by the medium action. |  |
| Backward heavy:   * Causes a large amount of damage with a knockdown effect. * Must be activated while walking away from the opponent. * Activated by the heavy action. |  |

When transitioning between the gameplay and the menus, the two Controls class instances (c1 and c2) are removed from the window using removeKeyListener() and an instance of the MenuNav class (menuNav) is added to the window using addKeyListener. This also occurs in the opposite way, when transitioning between the menus and the gameplay, the MenuNav instance is removed, and the two Controls instances are added. This allows the menu navigation controls and the gameplay controls to act correctly without any overlap.

AI

The AI class extends Player so inherits all its attributes and methods. This allows it to act as a player but has an extra method called chooseAction(). This method is used before every update of the player and determines what the AI should do. This works by finding a random number and deciding which action to perform. Each action must satisfy the correct conditions to work correctly, for example, if the random number decides that the AI player should attack but this player is currently stunned, the attacking action will not occur. Every action, such as crouching, walking, or jumping, has a ten percent chance of occurring except the attacking action which has a forty percent chance. Each attack has an equal chance to occur but require being in the correct activation state, for example, when in the idle state and a light attack is chosen, make the standingLight state true.

Graphics

The GUI class handles all graphical elements within the window. Every iteration of the gameloop or change in the menus, calls the GUI's paintComponent() method. The next section, Window and Menus, will go over how to show menus.

During the gameplay loop, the background image is drawn first, then both players, and finally the overlay.

A random image from the BACKGROUND\_IMAGES arraylist is chosen at the start of each game. When drawn, the background image is scaled to the window dimensions, so that it fills the window.

Players are shown as character sprites, which come from images called sprite sheets, see appendix B for examples. To show the correct sprite for each animation, each sub image of the sprite sheet is found, and scaled to the correct size and location, by using the configuration arrays in the Constants class. This uses the getSubImage() and getScaledInstance() methods from the BufferImage class.

The correct frame of an animation is shown by checking the data in the PlayerFrames class. This class holds many integers that tell the GUI how far through an animation the player currently is. Also, the PlayerStates class contributes by allowing GUI to find the right animation through a series of conditional statements.

The overlay consists of the following for both players: a health bar, a score tally, a character portrait, and a name. It also includes the timer and sits at the top of the screen. Health bars are created by drawing a red rectangle, and then drawing a green rectangle on top. The green rectangle’s length is determined by the player’s health, so when at full health, the health bar is fully green, whereas, when lower on health, some red is shown. The score tally uses the p1Wins and p2Wins attributes of Game to colour in up to two circles to show how many rounds a player has won. The timer shows the time left of this round in the top middle of the screen.

All sprite sheets are taken from this website, (SpriteDatabase, n.d.).

Window and Menus

The game window itself is simple. As discussed in the previous Graphics section, most of the graphical interface is controlled by the GUI class, so the Window class doesn’t do too much. Window is an extended JFrame so acts quite similarly. Window uses the GUI instance from the Game class and packs it into the content pane. The window also has a default close operation that terminates the program and cannot be resized.

These are all the menus: start, main, story, arcade, training, options, and character select. The start menu is the title screen, showing a logo and “press enter to start”. The main menu gives the user 5 menu options to choose from: story, arcade, training, options, and quit. The story menu describes what the story mode is and sends the player to the character select menu for the story mode. The arcade menu describes what the arcade mode is and sends the player to the character select menu for the arcade mode. The training menu describes what the training mode is and sends the player to the character select menu for the training mode. Quit just closes the program.

The options menu allows the user to change each player’s control scheme. This is done by prompting the user to input the desired key for each action, see the Controls and Attacks section for more detail on the different types of action. The keys that are entered cannot be duplicate keys on either control scheme, otherwise the user is given an error message and is sent back to the main menu.

The character select menu shows the user all the characters, using their portraits, and prompts the user to input a number to select the character. Duplicate characters are allowed but the player 2’s sprite is colour swapped, so that the players don’t get confused which character they are.

Menus are shown by drawing a background image and then drawing text over the top. The text on a menu is drawn with an outline to make the letters stand out, this is done by using a GlyphVector.

Music and Sounds

The class SoundManager handles all music and sounds in the game. Each sound and song is held in their respective arrays and are of the datatype Clip, from the javax.sound.sampled API. All audio clips are loaded at the beginning of the program in the constructor of Game. The sound clips are the hit sound effects and are played when a player gets hit by an attack. The music is randomly chosen when starting a new game and are played until the game ends (the clip is stopped when returning to the menus).

All music is taken from this website, (khinsider, n.d.).

Testing

Most testing came from changing certain gameplay values so that the game would feel authentic and smooth.

First, movement speeds, in the x and y directions, were tested. X values too high would make the player fly across the screen when trying to walk towards the opponent, and values too low would barely move the player. Y values too high would make the player jump up and down too fast for the animation to finish, and values too low would make the game seem like it is using moon gravity.

Next, damage values of attacks were tested to balance the game. The damage value given to an attack had to fit with the length of the animation and the range of the attack. If a fast attack, such as forward light, had a high damage output, there would be no reason not to use it, making the game extremely unbalanced.

Similarly, stun/knockdown effects for each attack was tested make each attack feel like it had impact. If a long kick attack, such as backward heavy, did not knock the opponent over, the game would feel less exciting.

Finally, the speeds of attacks were tested, so that the player can observe the whole animation and the attacks feel authentic.

To implement all the sprites, I added method to the GUI class called hitbox(), which draws a red box around each player, so that I could see when the animations lined up correctly with the player’s hitbox. Below on the left is a frame of an animation that lines up with the hitbox, whereas, on the right, it hasn’t been lined up.

A screenshot of a video game

Description automatically generatedA screenshot of a video game

Description automatically generated

This allowed me to easily tweak the sprite configuration data in the Controls class, so that the animation would fit in the hitbox and would look good for users.

Project Planning

During the autumn term, I completed 4-6 hours of coding work a week. This lead up to the Interim Oral Interview, which went well as I was aptly prepared. My autumn term workload from other modules was manageable so the project wasn’t hindered.

During the spring term however, I had a larger workload from other modules, which slowed my progress. But I had foreseen this, as you can see by my risk register in appendix A, so spent more time on the project towards the beginning of the term, and then used the later half of the term for other assignments. I did make a risk in Jira to document this break in my capstone project work.

During the Easter break and the start of the summer term, I finished the code and began work on this technical documentation, which was finished closer to the deadline than I would have liked.

I made effective use of Jira to keep track of my project workflow. I made many issues, all of them relating to specific tasks that needed to be completed, which were primarily coding tasks. I could have included more detail in these tasks though, as to a outside viewer they may not be as easy to follow.

Conclusions

Now that I have finished the programming of this project, I believe that I have completed every one of my objectives to a high standard. I have recreated the Street Fighter gameplay loop, which involves two fighters facing off against each other in a “best of 3” format, where players can attack with a large set of moves and can dodge/block attacks to avoid damage. Players can choose from a full roster of fighters and play with retro graphics and sounds. I have implemented three game modes, which can be played player vs player or player vs AI, and a menu system to navigate between them. Users can customize their control schemes through an options menu.

My proudest achievements are the sprites and the collision detection. I am proud that I could use sprite sheets and the states of a player to display the correct animations in such a way that my game looks remarkably like the classic game. Also, I am proud that I was able to create attack hitboxes that would interact with a player’s hitbox, such that the attacks are satisfying to use.

Over the course of this project, I have learnt a few new skills and have gained useful knowledge for my future career and academics. My new skills are based around Java. I now know how to adapt keyboard key presses into controls for games, this is by using KeyAdapters/KeyListeners. I also now know how to load images and sounds into a Java program and can manipulate them with ease.

I gained knowledge and experience on how to plan a long project, and how to keep my momentum. I seem to work well in the Agile approach to project management. Finally, I have learned that Java is not well-suited for creating games and that the performance of a Java game is poor. In the future, I would prefer to use C#/C++ or a game engine to create games.

If I get the opportunity to work on this project again in the future, I already have a few plans for what could be implemented next. Firstly, each character has the same move set in the current game, and this could be changed so that each character is completely unique. This would include attacks with different effects and timings, and special moves like the Hadoken. These changes and additions would allow players to choose a character that suites their playstyle, for example, one player may prefer to attack from afar, whereas another player may prefer to get in close.

I would also improve the AI player, allowing them to make informed decisions based on the human player’s current position or state, for example, when the human player is far away, AI prioritises moving closer.

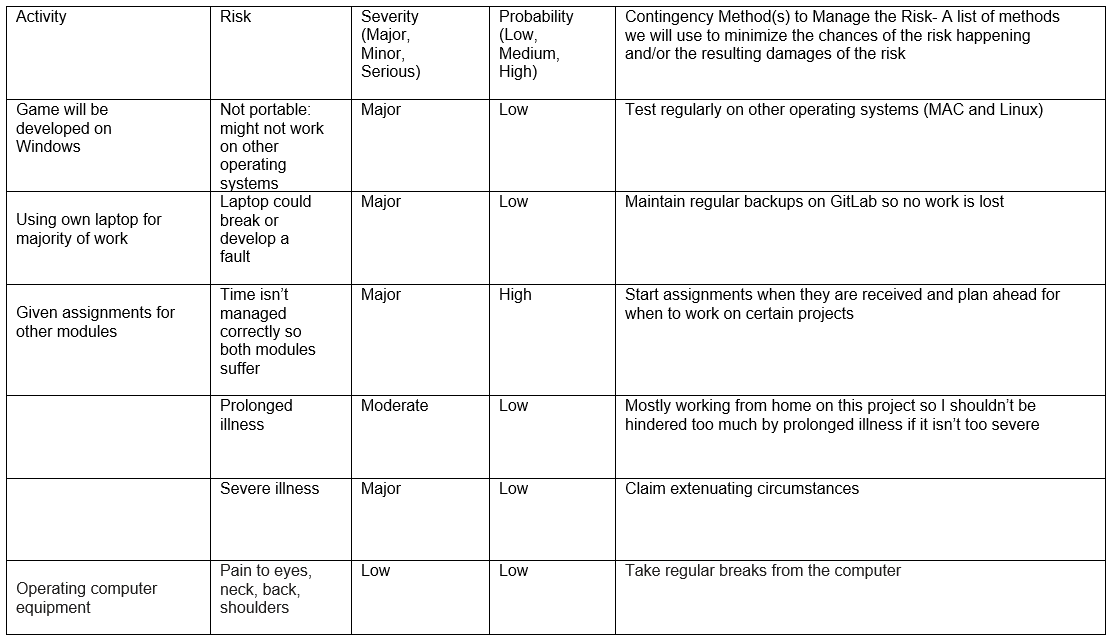
References

khinsider, n.d. *khinsider.* [Online]   
Available at: https://downloads.khinsider.com/game-soundtracks/album/street-fighter-ii-the-definitive-soundtrack

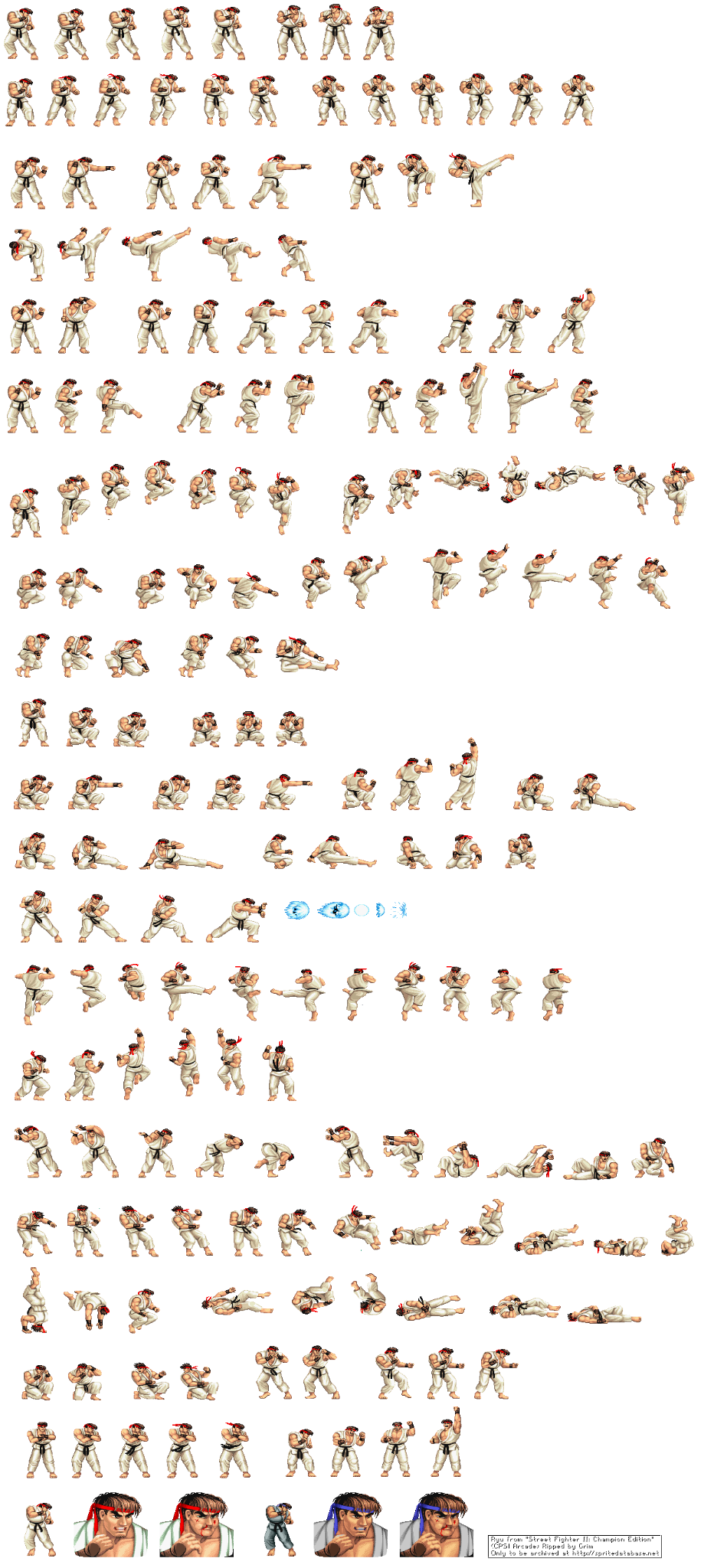
SpriteDatabase, n.d. *Sprite Database.* [Online]   
Available at: https://spritedatabase.net/search.php

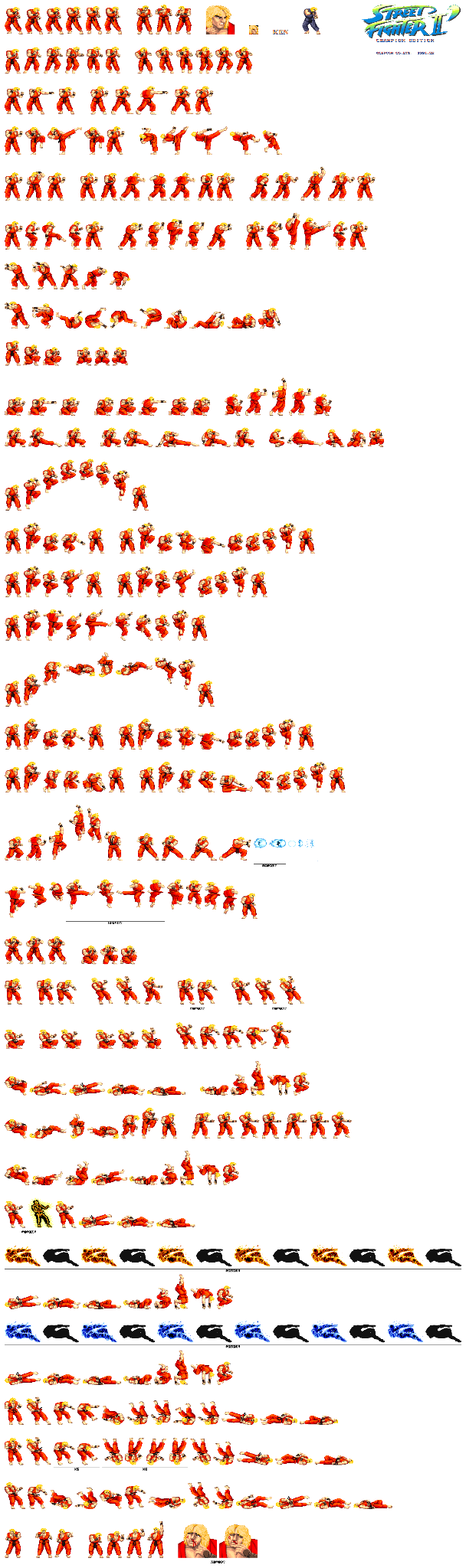
World of Longplays, 2013. *Arcade Longplay [370] Street Fighter II: The World Warrior.* [Online]   
Available at: https://www.youtube.com/watch?v=xI284D4y1q4

Appendix A: Risk Register

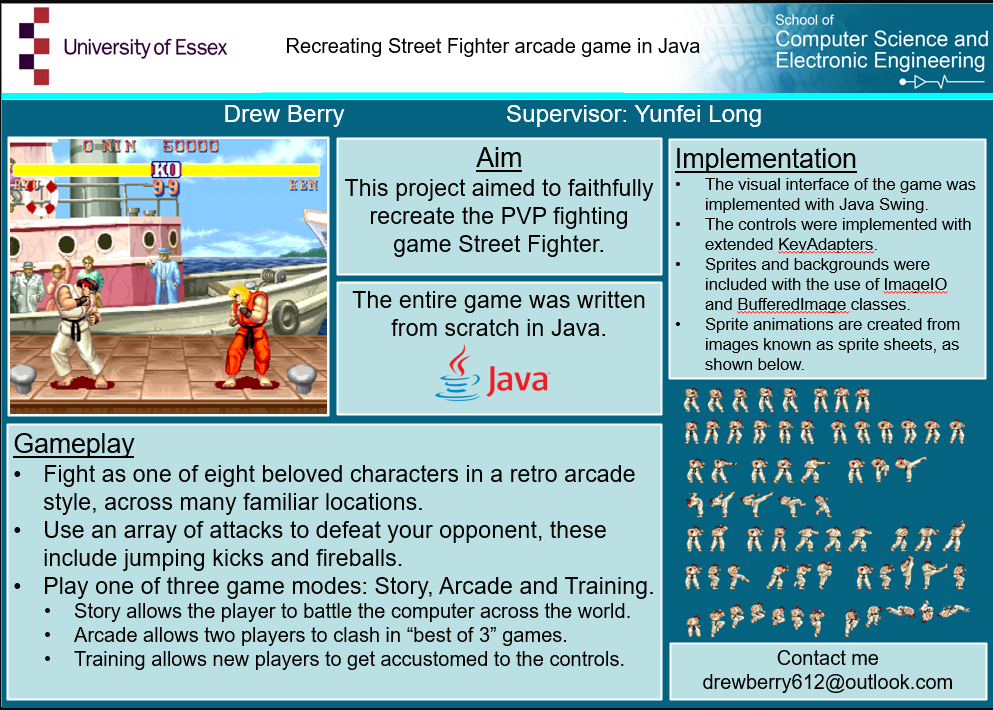


Appendix B: Character Sprite Sheets

Ryu Ken



Appendix C: Open Day Poster



Appendix D: Project Links

Jira: <https://cseejira.essex.ac.uk/secure/RapidBoard.jspa?rapidView=6850&projectKey=C301253>

Gitlab: <https://cseegit.essex.ac.uk/22-23-ce301/22-23_CE301_berry_drew_a_j>