Programming Project

Analysis **6**

Problem recognition 6

User requirements 6

Project Objectives 9

Research: Games which are similar to my final idea 9

Dragon Ball FighterZ 9

Tekken 3 11

Limitations 12

Hardware requirements 13

Design **14**

Prototype 1 **14**

Objectives 14

My_sprite class 14

The character class 16

Animate methods: 17

Set hitbox method: 17

Changes to the update method: 17

Modules needed for the prototype 18

Sprite sheets 20

Prototype 2 **21**

Objectives 21

Combination key recognition 21

The action class: 21

Character classes 23

The sub-character class: 23

Character Selection: 24

The Selection Sprite Class: 24

Character specific attack boxes and values: 25

Detecting collision: 27

The Health Bar Class 27

Modules needed for this prototype 28

Prototype 3 **30**

Objectives 30

Player reaction 30

Holding certain actions 31
Moving while in air 32
The level 1 enemy response 32
The enemy class: 33
The Raditz enemy class: 34
Character classes inheritance diagram: 35
Enemy action class: 35
The main menu screen 36
The select sprite parent class: 36
The menu select sprite class: 37
Setting up the timer 37
The number class: 37
Making the actual prototype 39
Organising the classes into modules: 39
Changes to the existing modules: 39
Prototype 4 **41**
Objectives: 41
Adding special moves into the game 41
Recognising a special move: 42
Including special moves in the character subclasses: 44
The ki_blasts class: 44
Kamehameha update method: 45
Death Ball update method: 45
Changing the check attack method 46
Creating the Practice mode 46
Making the pre-level screen 48
Making the end level screen 49
Creating the level 2 enemy response 50
Organising the level function 50
Making the actual prototype 51
Prototype 5 **52**
Objectives: 52
Saving and loading player data using a database 52
The save data class: 53
The unlockable character select screen 54
Adding special moves for Vegeta and Trunks 55
The Controls screen 56
Creating the level 3 enemy response 57
Adding background music and sound effect to the game 57

Making the actual prototype 57
Prototype 6 **58**
Objectives: 58
Looking at, and maybe fixing, mistakes 58
Methods in the character subclasses: 58
Simplifying the main code: 58
Checks with the queue: 59
Adding the character name to the save data object: 59
Fix the kamehameha and the galick gun special moves: 59
Including repeated characters in the player name: 59
Scrolling character selection: 60
The character select screen: 60
The enemies of the arcade mode 60
Development **61**
Prototype 1 **61**
Fixing the Animation methods 61
Frame numbers: 61
Default mode: 61
Directional movements: 61
Changing the update method: 63
Attacking a character 64
Added attributes 67
The remainder of the code 69
set_up.py: 69
sprite_interactions.py: 71
update_display.py: 73
main.py: 75
Prototype 2 **76**
Fixing the combination key recognition 76
Hitboxes and Attack-boxes 77
Collision Detection 79
Fixing the collision detection: 79
Testing the collision: 79
The remainder of the code 81
set_up.py: 81
sprite_interactions.py: 83
update_display.py: 84
main.py: 85

Prototype 3 **85**
Fixing the player reaction function 85
Holding actions methods and procedures 87
Why moving in air won't work 88
Developing the enemy response 89
Changes to the enemy class: 89
Testing the enemy set_queue method: 89
Making the main menu 92
Changes with displaying the timer on screen 92
The remainder of the code 93
set_up.py: 93
sprite_interactions.py: 93
update_display.py: 95
main_funcitons.py: 95
main.py: 97
Prototype 4 **98**
Fixing the special moves 98
Character reaction: 98
Special move check: 98
The two special moves: 99
Testing the effectiveness of changing the character values 102
Time taken to beat level 1: 103
Time taken to lose level 1: 103
Correcting the practice mode 104
The pre-level and end-level screens 104
Testing the level 2 enemy 105
Test 1: 105
Test 2: 106
The remainder of the code 107
set_up.py: 107
sprite_interactions.py: 110
update_display.py: 112
character_extras.py: 112
mian_function.py: 112
main.py: 113
Prototype 5 **115**
Changes made to loading and saving data 115
Making the unlockable character select screen 116
Adding the new special moves and the controls screen 117

Correcting the enemy response 117
Music and sound effects in the game 118
Comments on Curtis' feedback 119
Decreasing the amount of pixels moved: 119
Changes made to the choice of Frieza's attacks: 119
The wrap around screen: 119
"Infinite jumping" glitch: 120
The remainder of the code 120
set_up.py: 120
sprite_interactions.py: 120
update_display.py: 122
character.py: 122
character_extras.py: 123
main_functions.py: 124
data.py: 126
main.py: 127
Prototype 6 **128**
Any fixes that were made 128
Checking the "queue": 128
Trying to fix two of the special moves: 128
The arcade mode main function: 129
Fixing the set name function: 130
Fixing the scrolling selection: 130
The character select screen: 131
Evaluation **132**
Prototype 1 132
Prototype 2 132
Prototype 3 134
Prototype 4 135
Prototype 5 138
Prototype 6 139
Final Evaluation 140

Analysis

Problem recognition

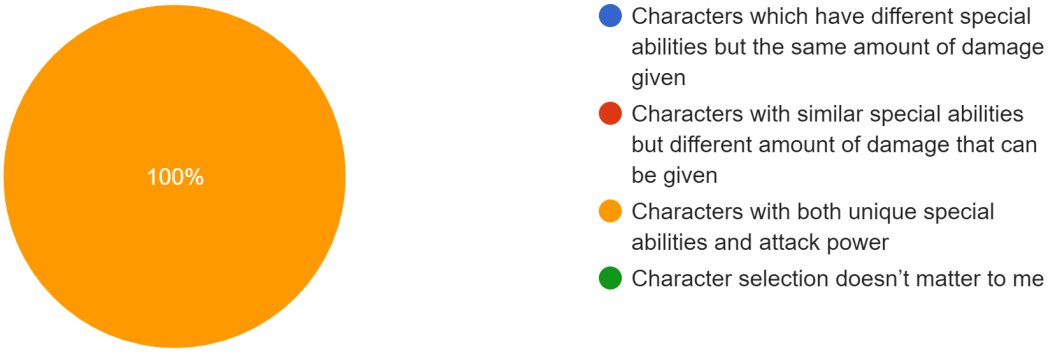
There are many aspects of life which can be stressful on young people and video games are one way to relieve that stress in a healthy way, as when you're playing a game, you're actively doing something and that has no effect on the real world - so if you are good or bad at the game, it has no meaning in reality. There are many other ways to deal with stress, but looking at people in my age group (year 12 and 13 students) I think that picking up a game to play would be the most accessible, requires little effort and most of the calming aspects of playing a game will be subconscious.

Personally, I think that games which are more challenging and require a certain strategy are the best type of solution to this as the player can get angry as they are playing but once they have completed that challenge it is satisfying. A genre of games which comes to my mind are the 2d fighters: they're simple enough to easily understand the dynamics of the game and also have an element of complex strategy which can differ from level-to-level.

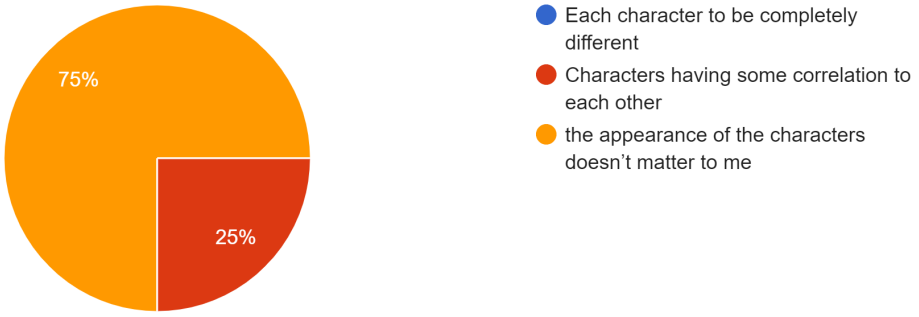
User requirements

As previously stated, I want my product to be aimed towards young people looking for a way to relieve stress, so I asked a few of my friends a selection of questions which aim to answer what specific aspects of a 2d fighter do they find the most fitting for a game which is overall satisfying to play:

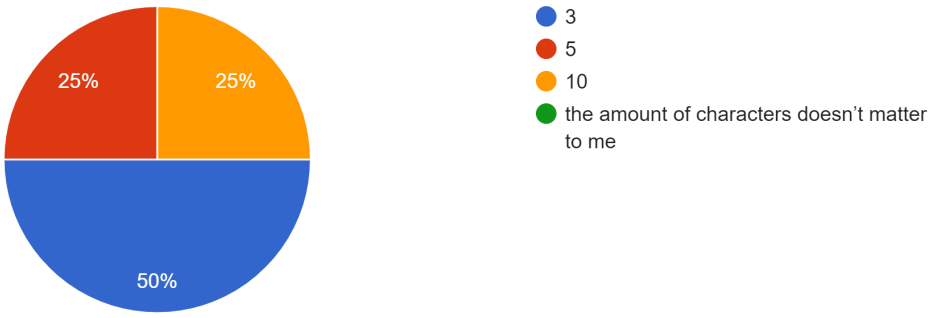
1. When it comes to character selection, which do you prefer:



2. Thinking about the appearance of the characters, would you prefer:



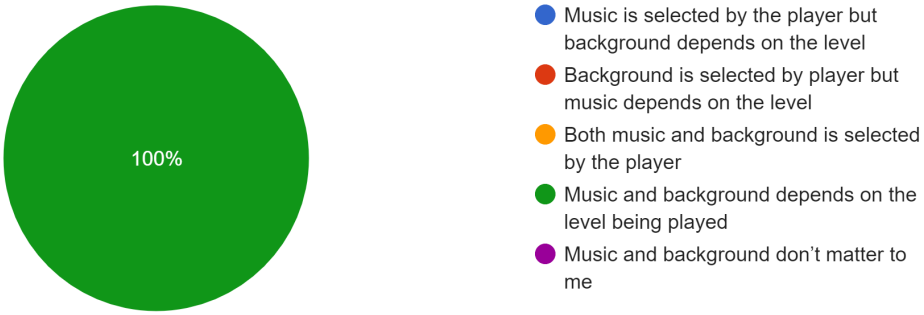
3. The preferable amount of characters that can be selected at beginner level would be: (keeping in mind characters might be able to be unlocked)



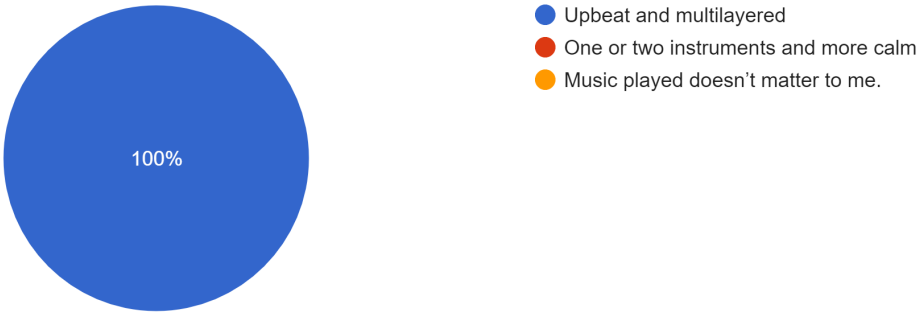
4. When it comes to enemies in the game, which do you prefer:

5. New characters can be unlocked by:

6. Thinking about the music and background of the game, which would you prefer:

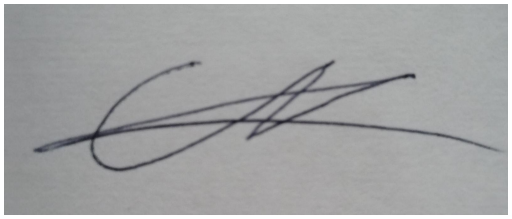


7. The music played while in battle is:



Two people agreed to test and give feedback on each prototype; their signatures agreeing to this are as follows:

Curtis Mitchell: Kanwal (Kanny) Rashid:



Where Curtis plays a lot of games and Kanny doesn't play games as often.

Project Objectives

- Allow the player to select a character, where each character has different move sets, strengths and weaknesses
- Allow the player to fight against enemies of different levels of difficulty, either by selecting the difficulty or playing in an arcade-mode type style
- Allow the player to play different modes like practice mode or local multiplayer mode (where the game mode is player v player using the same keyboard)
- The game should be close to the style and aesthetic of a 2d-fighter game

Research: Games which are similar to my final idea

I tried to look for games which I thought had gameplay mechanics which were easy to learn and had a simple game mode where you finish one level to move onto the next. The games that came to my mind were:

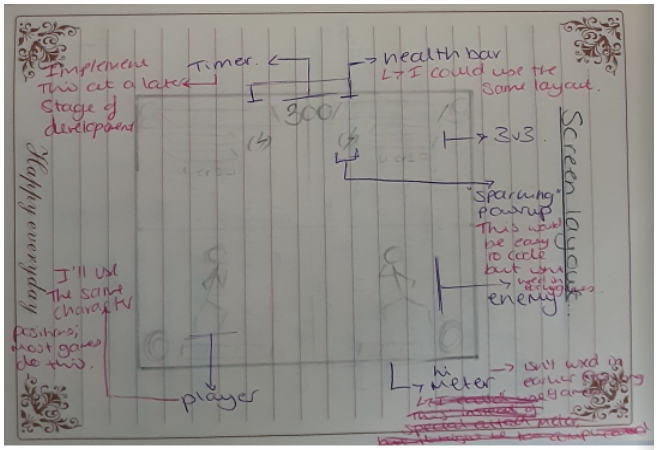
Dragon Ball FighterZ

Main controls:

X	-	special attack
O	-	weak attack combination
△	-	mid-strength attack combination
□	-	heavy attack combination

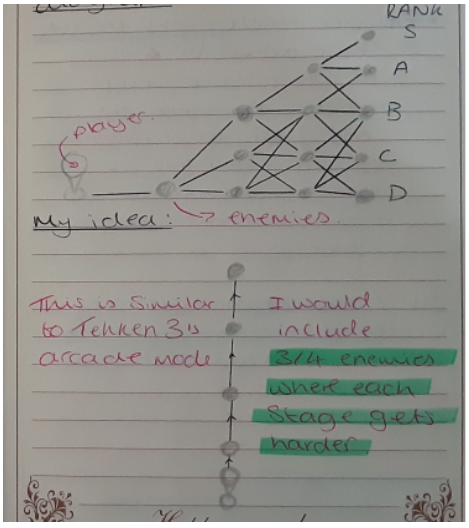
These attack controls are too complicated to add into my project

Screen layout:



Game modes:

- ➔ Arcade mode
  - ◆ Players go through a “tree” of enemy teams; the difficulty depends on how well the player performs in the previous battle. The higher the difficulty, the higher the points earned
  - ◆ I will probably create a more simple version of this: the difficulty of the enemy increases as the player moves up the tree but there will be only one tree. Points earned are from time taken to complete the level and other factors
  - ◆ diagram:



- ➔ Practice mode and PvP mode
  - ◆ These would be easy to include in my project but I would include these near the end development. For the practice mode, I wouldn’t include the fight tutorials

since the game would be simple to play and the player can just figure out attack combinations

➔ Other modes include: story mode, online multiplayer mode and team raid mode.

Tekken 3

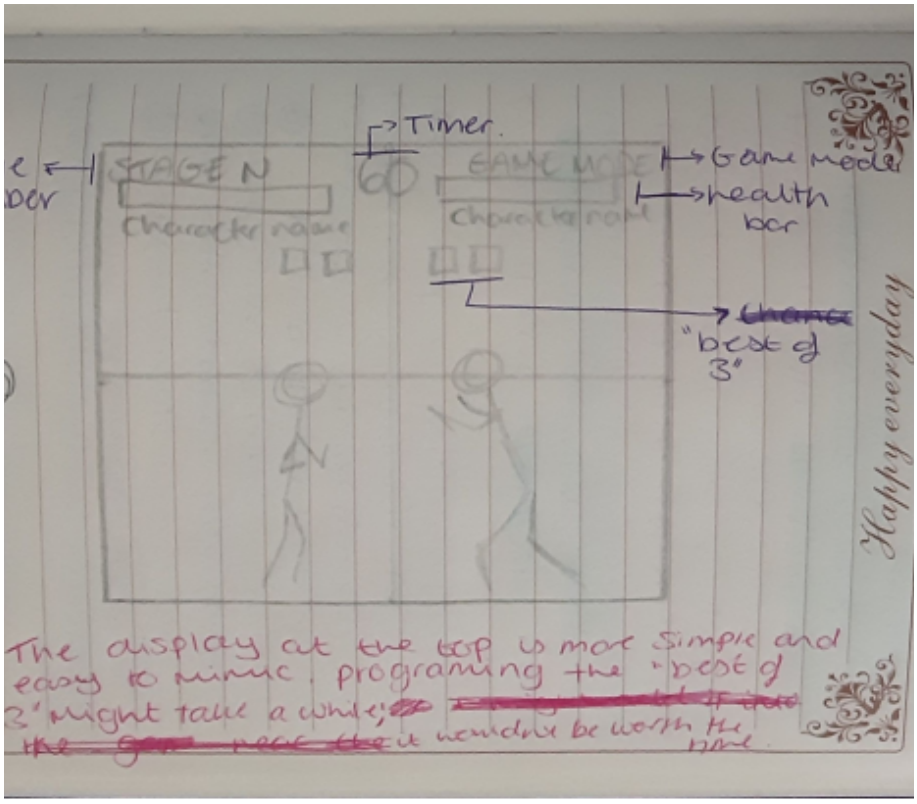
Main controls:

X	-	left kick
O	-	right kick
△	-	right punch
□	-	left punch

With using directional buttons to change to a low, mid or high attack

I will use attack buttons similar to this, maybe adding a block for one of the attack buttons, depending on what sprite sheets i can find/use. Also using the keyboard keys A,S,Z and X in place of the playstation buttons.

Screen layout:



Game modes:
(using the information given in the game manual)

- ➔ Arcade mode
  - ◆ “One player against the computer. Defeat all the opponents the computer throws at you and you win the game. There is no limit to how many times you can continue”
  - ◆ I would use this format but decrease the amount of enemies and the player has to restart when they lose instead of being able to continue; this will make the game more challenging. I also want to display the enemy tree between levels to show the player’s progress
- ➔ Practice mode
  - ◆ “This is where you can hone your fighting technique with any character and any opponent of your choice”
  - ◆ This would be the best way to implement a practice mode in a simple fighting game. But where this mode is customisable, i would set what can be done e.g one enemy who doesn’t respond (otherwise it would be like playing the main game) and the buttons pressed would be shown on screen
- ➔ Other modes include: survival mode, time attack mode and team battle mode.

Limitations

For specific users, a video game of this style might not be as successful at reducing stress. An example would be, if someone was colourblind it might be hard to distinguish between the several sprites displayed on screen and therefore more stressful to play; in an attempt to fix this, I’ve used sprites from a popular franchise and assumed that the color pallets used are already colourblind friendly. Another example could be if someone had trouble with movement in their hands or a slow reaction time, since the product will include real-time fights and to tackle this I will try to make the enemy’s attacks not too fast.

From the perspective of me programming the game, a limitation could be that I am working alone whereas normally on game development you would have a team of developers working on different aspects of the game. To try to make the product as successful as possible, and due to the reason I am one person, I will not include as many game modes as in the games researched.

Hardware requirements

This product will work on most PCs as python and pygame, the programming language used in this product, can work on most PCs; Pygamev1.9.6 can be used with Python versions 3.7.7 or greater and Pythonv3.7.7 can be used on Windows versions released after WindowsXP - note that I am using Python3.8 to program this product but it can be run on different versions of Python. There are no additional hardware requirements other than a computer which runs on Windows.

Design

Prototype 1

Objectives

- Display and animate characters on screen
- Let the player be able to hit an enemy
- Enemy should react
- The game should stop when the enemy is defeated
- Load a background

My_sprite class[1]

The pygame sprite class allows for images to represent sprites by having attributes such as x position, y position and a rectangle outlining the area around it. This class also allows for the sprites to be added to a pygame sprite group, which is a container class to hold and manage multiple sprite objects, allowing for them to be drawn on the screen and updated collectively The “my_sprite” class is an extension of the pygame sprite class allowing for sprites to be animated using a master sprite image. The image would need for the frames to be equally spaced for this to work; this means that the hit area and the area from the object get rect function would be different.

[1] *More Python Programming for the Absolute Beginner*
Chapter 7: Sprite animation demo
Author: Jonathan S Harbour
Publishers: Course Technology PTR
Published in 2011

```
import pygame
from pygame.locals import *

class my_sprite(pygame.sprite.Sprite):
    def __init__(self, target):
        pygame.sprite.Sprite.__init__(self) #extend the base Sprite class
        self.master_image = None
        self.image = None
        self.frame = 0
        self.old_frame = -1
        self.frame_width = 1
        self.frame_height = 1
        self.first_frame = 0
        self.last_frame = 0
        self.columns = 1
        self.last_time = 0

    #X property
    def _getx(self): return self.rect.x
    def _setx(self, value): self.rect.x = value
    X = property(_getx,_setx)

    #Y property
    def _gety(self): return self.rect.y
    def _sety(self,value): self.rect.y = value
    Y = property(_gety,_sety)

    #position property
    def _getpos(self): return self.rect.topleft
    def _setpos(self, pos): self.rect.topleft = pos
    position = property(_getpos, _setpos)

    def load(self, filename, width, height, columns):
        self.master_image = pygame.image.load(filename).convert_alpha()
        self.rect = Rect(0,0,width,height)
        self.image = self.master_image.subsurface(self.rect)
        self.frame_width = width
        self.frame_height = height
        self.columns = columns
```

```
def update(self, current_time, rate, x, y):
    #update the animation frame number
    if current_time > self.last_time + rate:
        self.frame += 1
        if self.frame > self.last_frame:
            self.frame = self.first_frame
        self.last_time = current_time

    #build on current frame only if it changed
    if self.frame != self.old_frame:
        #gets currenet frame by covering up master image
        frame_x = (self.frame % self.columns) * self.frame_width
        frame_y = (self.frame // self.columns) * self.frame_height
        self.rect = Rect(frame_x, frame_y, self.frame_width, self.frame_heig
        self.image = self.master_image.subsurface(self.rect)
        self.old_frame = self.frame

    self._setx(x)
    self._sety(y)
```

```
def __str__(self):
    return str(self.frame) + "," + str(self.first_frame) + \
           "," + str(self.last_frame) + "," + str(self.frame_width) + \
           "," + str(self.frame_height) + "," + str(self.columns) + \
           "," + str(self.rect)
```

The character class

This class will have to inherit from the animating sprite class and this class is made so that the characters used in the game will have values which a fighter would have: for example, a certain attack power or health value.

(as well as the attributes from the animating sprites class) Health (int) Attack_power (int) Defense (int) Hitbox (pygame rect) Animating (boolean) Name (string)
Get_health Set_health Get_attack_power Set_attack_power

Get_defense
Set_defense
move
Get_hitbox
Set_hitbox
Update (changed from parent class)
Animate methods

Since all characters (if their spritesheet follows the same pattern as the animate methods) can be initialised using the character class, extra characters can be added post-development

Animate methods:

I will make a method for each “move” the character has e.g:

```
METHOD punch():
        self.frame ← 26
        self.last_frame ← 31
        self.set_hitbox()
        self.update()
```

Used in game:

```
WHILE play
        FOR each keyboard event
                IF user input = key a THEN
                        Character object.punch()
```

Set hitbox method:

Since the player rect will be too big to be used as the hitbox, i created a text file with the dimensions of each frame. A hitbox will be used to check if one character has attacked another using a rect collision method, from the pygame sprite class.

```
METHOD set hitbox
        Open dimensions.txt in the correct character folder
        IF self.frame = the first number THEN
                Self.hitbox_width ← the second number
                Self.hitbox_height ← the third number
        END IF
```

Changes to the update method:

In the my_sprite class, the update method makes the animation continuous, whereas in the player class the animation sequence only needs to happen once. I have changed this update method in the character class in case I need the original update method from the parent class.

```
METHOD update (self, current_time, rate, x, y)
        self.animating ← True
        WHILE self.animating
                IF the frame needs to change THEN
                        self.frame += 1
                        IF self.first_frame > self.last frame THEN
                                Self.animating ← Fasle
                        END IF
                END IF
```

For when the animation needs to be continuous:

```
WHILE not animating
        character.default()
```

Modules needed for the prototype

- ➔ Animating_sprites
  - ◆ Which includes the my_sprite class
- ➔ Character
  - ◆ Which includes the character class
- ➔ Set up
  - ◆ Which sets up the display and loads the character sheets. functions / procedures include
    - Setup screen
    - Load character
    - Add group
    - Set positions
    - Print text on screen
- ➔ Sprite interactions
  - ◆ Which includes functions / procedures which controls what the sprites do
    - Player reaction
    - Enemy reaction (only reacting to being hit)
    - Stay on screen
    - Check attack (if the player has hit the enemy)

- ➔ Update display
  - ◆ Shows everything on screen which includes:
    - Update screen
    - Exit game
    - Check defeated

- defeated

➔ Main
- ◆ Where all the functions and procedures are placed
- ◆ Variables needed for this module:

Name	**Type**	**Description**
play	boolean	Stays as True until a character is defeated
screen	Pygame object	-
player	Character object	-
enemy	Character object	-
character	List of character object	Used in the “add group” function so that multiple characters can be added to the pygame sprite group
background	Pygame image object	-
Clock variables	integer	-
defeated	boolean	If either character is dead then defeated is set to True, and the game displays the winner and then stops

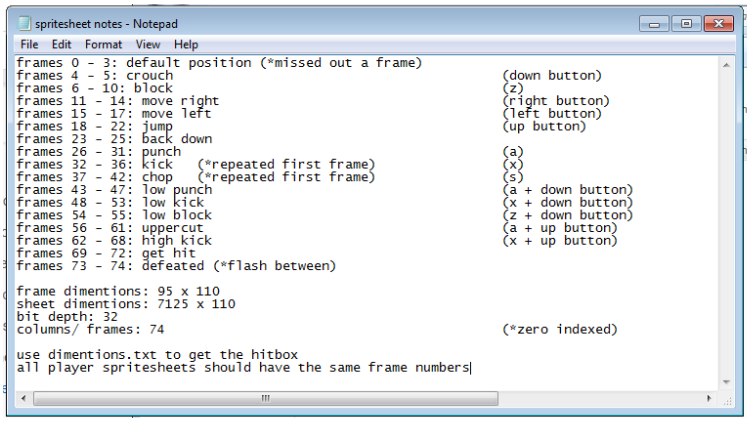
Sprite sheets

For this prototype, I need two sprite sheets: one for the enemy and one for the player. I have created a character folder for each character which contains:

The actual sprite sheet[2]

A zoomed in screenshot:

A text file describing which frame numbers go with which animation move



And a text file with the dimensions of each frame needed for the set-hitbox method

[2] The Spriters Resource
Authors: (no surname given)
Date Viewed: 06/07/2020
Year Published: 2006
<https://www.spriters-resource.com/>

Prototype 2

Objectives

- Fix the combination key recognition
- Character selection
- Health bars
- Fix the background

Combination key recognition

Since the pygame.key.get_pressed() booleans are set to False at every millisecond, the player reaction function won't recognise when two keys are pressed. This could be fixed by putting the action that the character is going to do in a queue and check with the previous item whether a combination move was intended and a queue is used as this abstract data type works by the first item to go into the array is the first to leave and items are therefore analysed in time order of when the key was pressed. In a set move function, the function will return instances of the action class for the 10 (including default) moves that can be done.The queue will be a dynamic queue since not many action objects will be added and the limit of how many keys can be pressed in the time it takes to remove and set the queue is unknown.

The action class:

To compare each character action to check if there is a combination move, various attributes of that move need to be checked.

name (string) mode (string - for the character update method) movement (boolean) has_mode (boolean)
get_name get_movement get_mode get_has_mode

```
FUNCTION set_move()
        punch ← action object
        block ← action object
        chop ← action object
        kick ← action object
        jump ← action object
        crouch ← action object
        right ← action object
        left ← action object
        down ← action object
        default ← action object
        RETURN a tuple off all action objects which are separated in the set queue function

FUNCTION set_queue(moves ← tuple of action objects, queue)
        reverse queue
        action_taken ← default (moves[9])

        FOR each key pressed
                IF key = a THEN
                        action_taken ← punch (moves[0])
                [check each key using if statements]
        NEXT key

        IF the player is too high THEN
                action_taken ← down (moves[8])

        IF the action isn't in the queue THEN
                Add action to the back of the queue

        reverse queue
        RETURN queue
```

And then in the player reaction function:

```
FUNCTION player reaction (player, queue)
        current_move ← queue[0]
        check_move ← queue[1]

        IF current_move.get_name() = “punch” THEN
                player.punch()
        [if statement for each move which requires one key pressed]

        IF current_move isn’t a movement action AND check_move is a movement action OR
        The other way around THEN
                IF current_move.get_name() == “punch” THEN
                        IF check_move.get_name() == “jump” THEN
                                player.uppercut()

                        IF check_move.get_name() == “crouch” THEN
                                player.low_punch()

        [if statement for each possible combination]

        mode ← action_taken.get_mode()
        clear queue
        RETURN mode,queue
```

Character classes

The sub-character class:

For each character, I will create a class which will set the master image, name, defense, attack power and health. I also edited the main character class to include an attribute called self.values which is a list which includes the character’s starting y position, highest y position etc and this is set in the sub-character class’ initialise method. Another attribute is the self.attack_box which is set as None in the super character class and then set in a method in the sub-character class. (pseudocode for the attack boxes explained later)

health original health attack_power defense path default_y highest_y down_value select_x
get_default_y get_highest_y get_down_value get_select_x set_box_lists [any special move specific to that character]

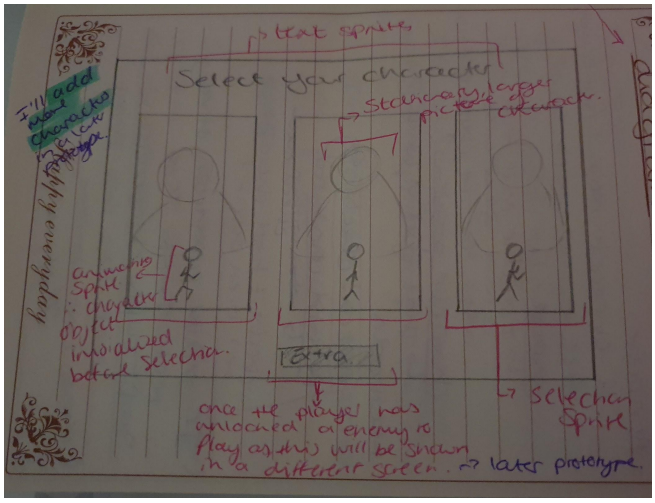
Character Selection:

I want the player to, initially, be able to select between 3 playable characters and they might be able to select an enemy to play as if they have defeated that enemy. Therefore for each enemy, there needs to be a character class where the player can select them and a character class as an enemy.

The Selection Sprite Class:

The selection sprite class which represents each character on the character select screen. When the player clicks on the selection sprite of a certain character, they then play as that character

Diagram:



character (a character object) rect
check_click

```
METHOD check_click
        IF mousedown in range(rect.x) THEN
                RETURN character
        END IF
```

In later prototypes when there is another character select screen with characters sharing y values, I will add a check for the range of rext.y also.

In main code:

```
IF player not selected THEN
        option1 ← select sprite object
        option2 ← select sprite object
        option3 ← select sprite object
        [each for a different character]

        option1.check_click()
        [for all options]

        IF any option != None THEN
                IF option1 != None THEN
                        player ← option1
                [if statements for all options]
                END IF
        END IF
END IF
```

Character specific attack boxes and values:

Since each character has a different height and pygame measures y values increasing from top to bottom, values like default y and highest/smallest y are unique to each character. I am going to first find these values for the character Goku and then test other character's values in relation to that. And for the hitboxes, I'll try to work it out by drawing diagrams and counting pixels.

```
METHOD set box lists(x,y)
        Set each box list (for each attack move) in relation to the x and y coordinates passed in
        the update method

        self.attack_boxes ← a list of the attack box lists
```

An example of a box list:

```
self.punch_boxes ← [pygame.Rect(x,(y-25),10,10), None, None, pygame.Rect((x+15),
(y-25),10,10), None]
```

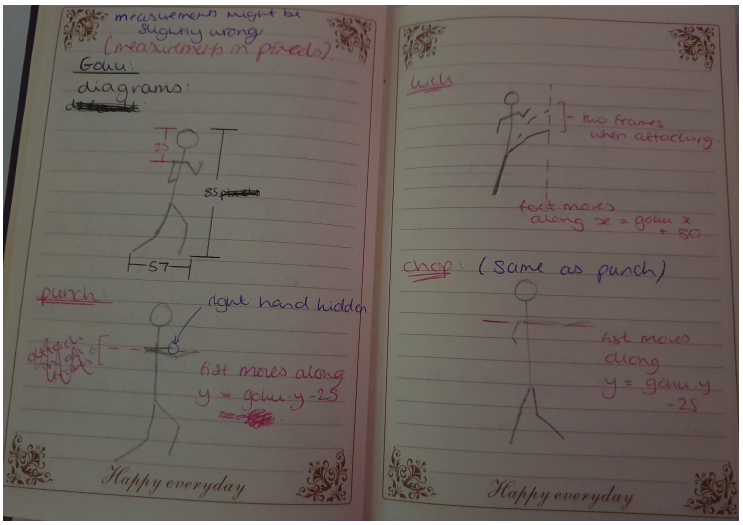
where the box is set to None is where the character is in the attack animation but isn't actually attacking

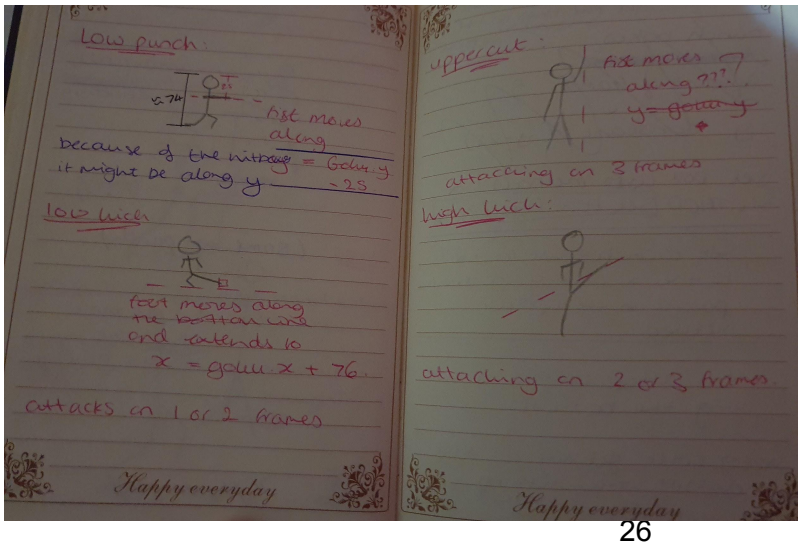
Set attack box method (in the super character class)

```
METHOD set attack box(x,y)
        Set box lists(x,y)
        box_list ← attack_boxes[attack_row]
        current_box ← box_list[frame_number]

        IF current_box != None THEN
                self.attack_box ← current_box
        END IF
```

where attack_row is an attribute of the super class which is set in the move method and it corresponds to which attack box row the attack they are doing is on, and frame_number is the frame number of the current animation cycle, set in the super class's update method

Diagrams used to predict x and y values:





Detecting collision:

I will move the original check_attack function from the sprite interaction module to a method in the character class just so that code isn't repeated

```
METHOD check_attack(opposition ← character object)
        IF opposition.attack_box != None AND rects collide AND NOT blocking AND NOT being
        hit THEN
                deduct health
                self.hit()
        END IF
```

In sprite interaction module:

```
IF player is attacking THEN
        enemy.check_attack(player)
END IF

IF enemy is attacking THEN
        player.check_attack(enemy)
END IF
```

The Health Bar Class

character (character object) enemy (boolean) added_x (integer) starting _width width height x y colour (tuple) bar_rect health_rect image (Nonetype; added in a later prototype)
get_health_rect get_bar_rect

set_rectangles (sets both rects)
character_hit
update
remove_health_rect

The character hit method is used to update the size of the health rect after the character that the health bar represents has been hit. If the character is an enemy then their health bar will be displayed at the opposite side of the screen and therefore their rect will have to move its position to look like it is moving mirrored to the player's health bar.

```
METHOD character_hit()
        value ← character health
        total ← character original health

        percentage_left ← value / total
        self.width ← integer(starting_width * percentage_left)

        IF enemy = True THEN
                percentage_taken ← 1 - percentage_left
                self.added_x ← self.starting_width * percentage_taken
        END IF

        IF percentage left is between 0.25 and 0.5 AND colour != orange THEN
                colour ← orange
        END IF

        IF percentage left is less than 0.25 AND colour != redTHEN
                colour ← red
        END IF
```

Modules needed for this prototype

Since I will have to switch between character selection and the first level, I will make a module called main_functions which has the main code (like main in prototype 1) separated into functions. I will use the same modules from prototype one but edit them a bit to fit this prototype e.g the set_positions procedure will need to be different for the character select and level one so i will have character_select_positions and set_level_positions

main functions will include:
[each line is a function or procedure]

```
FUNCTION character select
        ---setup---
```

```
        clear screen
        set_positions
        add_group

        ---sprite interactions---
        WHILE NOT player added
                player = check_click
        ---update---
                update screen

        IF player != None THEN
                RETURN player


PROCEDURE level_1(player)
        ---setup---
        clear screen
        load backgrounds
        initialise enemy
        initialise health bars
        play ← True
        defeated ← False

        ---sprite interactions---
        IF play THEN
                WHILE NOT defeated
                        event check
                        player response
                        enemy response
                        stay on screen
                        check attack

        ---update---
                        update screen
                        defeated ← check defeated
                        start time

                WHILE defeated
                        play ← defeated update
                        IF NOT play THEN
                                defeated ← False
```

Prototype 3

Objectives

- fix the character select screen
- create an easy-level enemy response
- include a main menu screen
- amend the player movement to Curtis' requirements
- create the health bar / timer (top screen) display to Kanny's requirements

Player reaction

In Prototype 2, the player reaction function was one big function; this may have been the reason why I found it hard to find how to fix the combination key error. (That an action which required a combination of two keys was only recognised if the keys were pressed while the player was not in default). So I will split the player reaction function into around 4 separate functions.

```
FUNCTION check queue
        IF len(queue) > 1 THEN
                set both current move and check move in another function

        ELSE IF len(queue) <= 1 THEN
                [in another function]
                current_move ← first item in the queue
                check_move ← None

        ELSE
                RETURN "", queue
        END IF

        RETURN current_move, check_move
```

Then combination_move_check(), single_move_check(), and defualt_check() are the same as previous, all returning the queue and mode

```
FUNCTION player reaction
        current_move, check_move ← check_queue(queue)

        IF current_move = "" THEN
                RETURN "", queue
        ELSE IF check_move != None THEN
                queue, mode ← combination_check()
```

```
        ELSE IF current_move != None AND check_move != None THEN
                queue, mode ← single_move_check()

        END IF
        queue ← default_check()

        RETURN mode, queue
```

Holding certain actions

As required from both clients, actions like moving left/right, blocking or crouching should be able to be kept on that action if the key for that action is still being pressed down.
In the character update method:

```
IF frame > last_frame THEN
        IF animating and NOT hold THEN
                animating ← False
                frame_number ← 0

        ELSE IF hold AND frame = last frame THEN
                self.hold_postion(keydown)
        END IF
```

where hold is a boolean attribute of the character class and is set to true in a move method where that move can hold a frame and keydown is a boolean which is passed into update and set in a keydown_check function

```
FUNCTION keydown_check
        IF the item at the front of the queue = block THEN
                k1 ← True
        ELSE
                k1 ← False
        END IF
        [if statements like this for crouch, move left and move right]
        RETURN k1, k2, k3, k4
```

where a list of 4 false's are passed in to the function and if any of the returned variables are true, true will be passed into update

```
FUNCTION check_true(list of booleans)
        IF any of the booleans in the list are True THEN
                RETURN True
        ELSE
                RETURN False
        END IF
```

and the hold_position method would be:

```
METHOD hold_position
        IF keydown = True THEN
                self.frame = last frame
        END IF
```

Moving while in air

To get the player to be able to move left/right while jumping, I have added a boolean attribute called highest_point where it will be set to True when the character's y position over a certain point.

```
METHOD set highest point
        IF y-coordinate < highest y coordinate AND y-coordinate < default y + 10 THEN
                highest point ← True
        ELSE
                highest point ← Fasle
        END IF
```

The method is called in the movement method when mode = "jump"; this is so that the program can check the queue when the highest point = True if the player wants to move.

In the combination move check:

```
IF current move.get_name() = "jump" THEN
        [all of the previous checks]

        IF player.get_highest_point = True THEN
                IF check_move.get_name() = "right" THEN
                        player.move_right()
                [the same with left and the reverse checks]
                END IF
        END IF
```

The level 1 enemy response

I will need to make two classes: and enemy class (a subclass of character) which has the methods which all enemies will need and a class which inherits from the enemy class and the character which is the enemy class, in this prototype the enemy will be Raditz and I'll add Vegeta as a playable character. In the Radtiz_enemy class, there will be an enemy reaction method which will be unique to Raditz's character, this is so the enemies can increase in difficulty as the levels progress, having a different character to fight against in the next level.

The enemy class:

queue detectbox_height detectbox_width detectbox value - used in the detectbox width/height
update set_detectbox get_detectbox check_player_position

The queue will be a static queue so that action objects can be inserted according to their priority, and therefore index values need to be known. This will be set at 4 spaces.

Where the detectbox is set using the hitbox width/height + a value which will be different for each enemy. The detectbox will be used to see if the player is close to the enemy and different enemies will respond differently to whether the player is in the box or not.

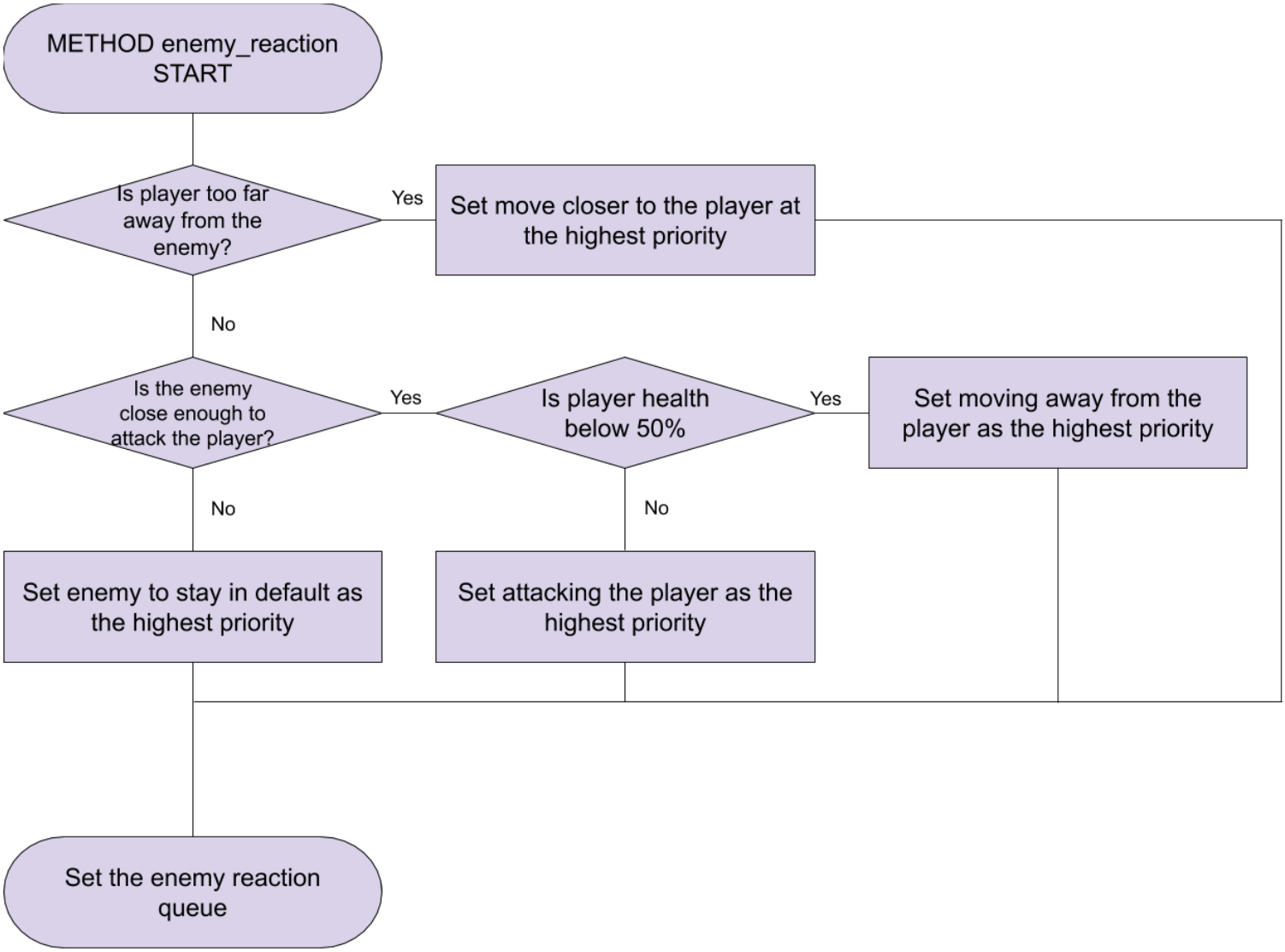
```
METHOD update
        super.update()
        self.set_detectbox(x,y)

METHOD set detectbox
        [using x and y from update]
        detectbox_height ← self.hitbox_hieght + value
        detectbox_width ← self.hitbox_width + value

        x ← x - n
        y ← y - m

        detectbox ← Rect(x,y,detectbox_width,detectbox_hieght)
```

I will use trial and error to find n and m; numbers used to try to make sure the enemy is in the centre of the detectbox.

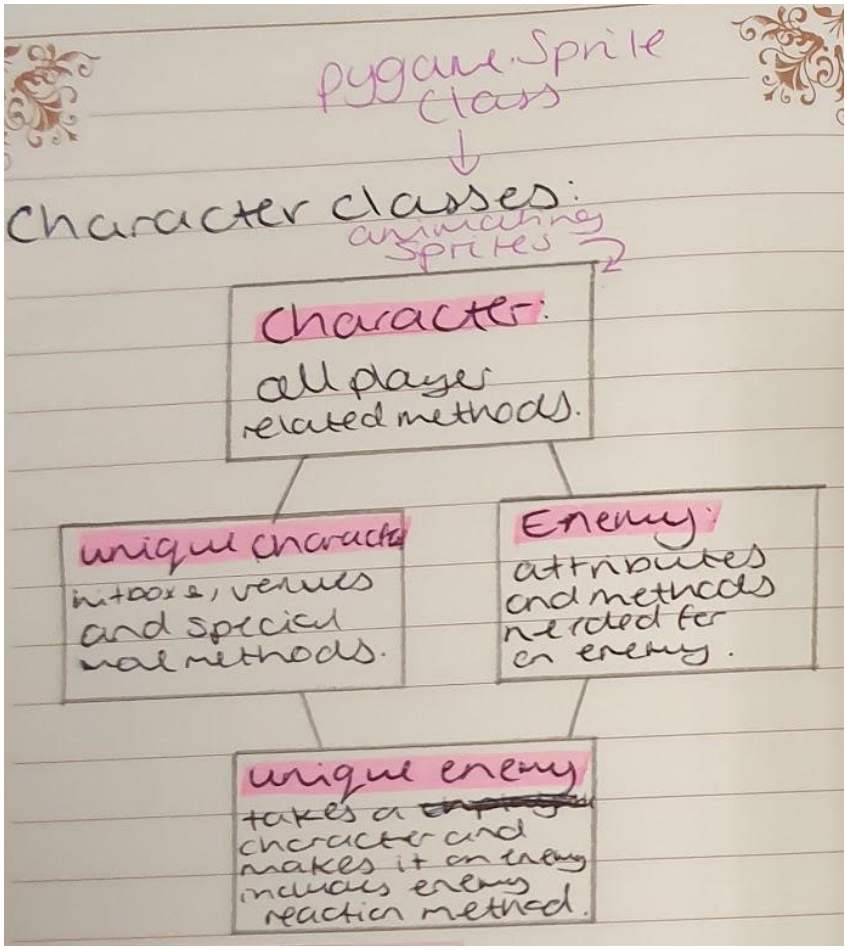
The check_player position method will be another version of the rect collision method for pygame sprites, where the player is passed into the method and the rectangles which are being checked are the detectbox and the player hitbox.

The Raditz enemy class:

The class will have no extra attributes and the initialise method will initialise the parent classes and set the value for the size of the detectbox. It will have 2 additional methods: a get_queue and a set_queue method, where the set queue method will be the basis for how the enemy will respond to the player; once the queue is set, it will be passes through the player_reaction function (in the finished prototype I will rename this function to character_reaction), which will allow for the enemy to be set to the right animation frames and respond to the player.



Character classes inheritance diagram:



Enemy action class:

The enemy needs action objects, like the player does, but since there needs to be an attribute for the priority of the move, so I will make a subclass of action called action_enemy.

priority - initialised as none then changed in the set_queue method
get_priority set_priority reset_prioritiy - sets priority to none; used at the end of the set_queue method

The main menu screen

For the user to be able to click on a text image and for that to lead onto what is described in that text (e.g arcade mode) I will need to re-design the select sprite class so that I can make select sprite classes inheriting from the select sprite class (e.g menu_select_sprite and chaarcter_select_sprite classes)

The select sprite parent class:

x y width height - both are initialised as 0 then set in the subclasses Rect image
check_click update

Where the check_click method has been changed to:

```
METHOD check_click
IF mouse position in range of rect THEN
    RETURN True
END IF
```

This will then be overwritten in the character select sprite class to return the character and the menu select sprite class won't change this method.
And the update method is where the image displayed changes because the player's mouse is on that image

```
METHOD update
    IF mouse position is in range of rect AND NOT self.check_clikc THEN
        self.image ← 2nd image in sheet
    END IF
```

To set the second image, I will have to add an attribute called image rect which will move to the second image by changing the x coordinate of the rect and using `self.image ← master_image.subsurface(image_rect)`.

The menu select sprite class:

The only thing that is different from this class to the parent class is the width and height needing to be set

Setting up the timer

Since mose 2D-Fighter games have a time limit for how long the level should last, I should put that in my game. This will be easy as the whole loop in the level procedures runs IF play = True, so the function which checks the timer will be similar to check_defeated but this will change the play boolean.

```
FUNCTION set timer
    level_start_time ← ticks
    RETURN level_start_time
```

Where ticks are from the pygame.clock.get_ticks() function which return the amount of time in milliseconds since pygame as initialised. So the time passes since the level started would be ticks or current_time - level_start_time.

```
FUNCTION check_timer
    IF current_time - level_start_time < 300 seconds THEN
        RETURN True
    ELSE
        RETURN False
    END IF
```

Where play is set to the returned boolean and I have used 300 seconds since that is the time limit for the game Dragon Ball FighterZ

The number class:

So that the time remaining can be shown on the screen, there needs to be a class which inherits from the pygame sprite class. The image of which number is shown can also be done in this class

number_displayed digit x y rect image
get_digit set_digit

update - the image of which number is displayed

set as 0, 1 or 2 so that the number can be positioned correctly according to which digit it is in the number (0 being the hundreds column if time remaining > 100)

```
METHOD update
        time_remaining ← str(time_remaining)
        [set as a string so that it can be indexed]

        number_displayed ← time_remaining[digit]
        path ← path to image of number displayed, using the variable number displayed
        self.image ← load image with path
```

Since I need time remaining and not time passed, I need a function for that in the main code

```
FUNCTION get_time_remaining
        time_remaining ← 300 - (current_time - level_start_time in seconds)
        RETURN time_remaining
```

I also need a function which will remove a digit from the group once it is not needed (e.g the third digit when time remaining < 100)

```
FUNCTION check_number_group
        IF time_remaining < 100 AND digit_1.alive() THEN
                group.remove(digit_1)
        ELSE IF time_remaining < 10 AND digit_2.alive THEN
                group.remove(digit_2”)
        END IF
        RETURN group
```

Where the .alive() is a pygame function which returns a boolean whether the sprite belongs to a sprite group.

The sprites that I am using:[3]

Where I’ve separated them into individual images and named them “1.png” etc so that it is easy for the class to load the image

[3] *King of Fighters* (1998) published by SNK for the PlayStation

Making the actual prototype

Organising the classes into modules:

If there was an error with any of these classes, it would be a bad idea to have all the classes in one module since it would be hard/take more time to find the correct class and fix the error.

- ➔ character.py
  - ◆ Contains the character classes and subclasses
- ➔ action_classes.py
  - ◆ Contains the action class and subclass
- ➔ level_top_display.py
  - ◆ Contains the number class and the health bar class
- ➔ select_sprites.py
  - ◆ Contains the select sprite class and subclasses

Changes to the existing modules:

- ➔ set_up.py
  - ◆ Add a set timer function
  - ◆ Add a menu setup function which returns a list of the main menu select sprites
  - ◆ Add a function which sets up the number sprites to return a list
    - With both lists of sprites, the add_group function will be used to return a sprite group, where I need to change the variable name characters → sprites

    - And the setup functions would be:

```
FUNCTION menu_setup
        path ← image path for arcade image to be used in class initialise
        method
        [path variable for each sprite]

        arcade ← menu select sprite
        practice ← menu select sprite
        controls ← menu select sprite
        exit_select ← menu select sprite

        sprites ← [arcade,practice,controls,exit_select]
        RETURN sprites
```

  - ◆ Set up the win, lose or out of time text images

➔ sprite_interactions.py
- ◆ Rename “player reaction” to “character reaction” and include the new player reaction functions
- ◆ Set enemy action sprites
- ◆ check click function for the main menu screen

➔ update_display.py
- ◆ Add check_timer, get_time_remaining and check_number_group procedures/functions
- ◆ Main menu update procedure
- ◆ “Time ran out” update
- ◆ Change the character update to include the keydown variable at the end of the character’s update method and to pass in the enemy’s mode

➔ main_functions.py
- ◆ Changing the level procedure into a function where it returns a boolean whether the player has won or lost the level
- ◆ Main menu function which returns 4 booleans; a rough idea would be:
  - FUNCTION main_menu
    list ← menu_setup()
    selected ← False
    group ← add_group(list)
    WHILE NOT selected
    x,y ← selct_event_check
    [check which option is true with x and y]
    IF op1 OR op2 OR op3 OR op4 THEN
    RETURN [op1,op2,op3,op4]
    END IF

➔ main.py
- ◆ screen ← setup_screen()
  arcade,practice,controls,exit_select ← main menu(screen)

  IF arcade THEN
  player ← character_select(screen)
  win ← level1(screen, player)
  exit_game()

  ELSE IF practice THEN
  pass

  ELSE IF controls THEN
  pass

```
ELSE IF exit_select THEN
    exit_game()
```

Where controls and practice mode will be added in a later prototype

Prototype 4

Objectives:

- Make the level 2 enemy response, which will be harder than level 1.
- Include a practice mode, where the player can fight against a responseless character and the keys they are pressing are displayed onscreen.
- Include a “special move” for one playable character and the level 2 enemy.
- Pre-level screen and “player advances to the next level” screen.
- Organise the level functions into smaller functions and procedures.
- Change each of the character’s attack power, health and defense values so each is unique, and ,with that, re-design the check attack method so that the character’s defense is included.

Adding special moves into the game

In all fighter games there is the inclusion of “special moves”, which are attacks unique to a specific character and which also require a unique set of keys to be pressed in a certain order to activate it; this will make it so that there is a different advantage to playing as each character. Another way to create advantages to play as each character would be by changing their attack power, defense and health values, which I will design later in this prototype. In my game, I’ll make it so that the player doesn’t know their characters special move, and therefore the player will need to figure this out either by testing different key combinations in practice mode or while playing the levels in arcade mode; this is similar to the play style of Tekken 3 and the idea was taken from that game.

Recognising a special move:

A function, similar to the combination_move_check, will have to be made, where this function will require more “check moves” (variables holding different items in the queue) so the check queue function and the player reaction function will need to be amended.

```
FUNCTION check_queue
        current_move ← queue[0]
        IF len(queue) > 2 THEN
                next_in1 ← 1
                next_in2 ← 2
                TRY
                        WHILE queue[next_in1] = default
                                next_in1 ← next-in1 + 1
                        WHILE queue[next_in2] = default
                                next_in2 ← next-in1 + 2
                EXCEPT if the queue has no moves != default
                        RETURN current_move, None, None
`       END IF

        check_move1 ← queue[next_in1]
        check-move2 ← queue[next_in2]
        RETURN current_move, check_move1, check_move2
```

And in character reaction:

```
current-move, ch1, ch2 ← check_queue()
IF ch2 != None THEN
        queue_mode ← special_move_check()
        IF queue_mode != None THEN
                queue, mode ← queue_mode
        ELSE
                [continue with code from previous]
        END IF
END IF
```

And then a special move check function needs to be created, which will be similar to the combination_move_check function but it

- ➔ Will need to check a combination of 3 keys which need to be pressed
- ➔ And the keys that need to be pressed and the corresponding move method will need to be returned from the character
    - ◆ I will need to know how to call a class method from a string variable set; if this works, I should also change both the combination move check and the single move check using this feature as this will increase the efficiency as there are much less if statements being used.

"Methods in a class can also be called using the string representation of its name: call getattr(object,name) using a method name in string form as name the class as object. Assign the result to a variable and use it to call the method with an instance of the class as an argument"[4]

```
class C:
    def m(self):
        return "result"

    def n(self):
        print("result")


an_object = C()

class_method = getattr(C, "m")
result = class_method(an_object)
print(result)
```

```
>>> ================================ RESTART ==============
==================
>>>
result
```

```
FUNCTION single_move_check
        move_taken ← False
        IF current_move = default THEN
                RETURN queue, ""
        END IF

        action_method ← getattr(character, current_move.get_name())
        action_method()
        mode ← current_move.get_mode()
        move_taken ← True

        IF NOT move-taken THEN
                [the same code as previous but with check_move1]
```

[4] Kite - documentation lookup for python, date viewed: 20/1/2021
<link to website used>

Including special moves in the character subclasses:

I'll add an attribute called special moves, which will be a dictionary where the key will be the name of the special move method as a string and the value will be a list of strings of the action object name that the keys return.
Example: {"kamehameha": ["punch", "chop", "right"]}
Where a kamehameha is an attack that is used in all of Dragon Ball, consisting of a blue energy blast and is mainly used by Goku.

```
METHOD kamehameha
        self.frame ← frame number
        self.last-frame ← last frame number
        self.ki_blast ← kamehameha()
        self.attacking ← True
        self.blast ← True
        self.frame_number ← 0
```

Where kamehameha() is a subclass of the ki_blast class; design for the ki blast class included later. And self.blast is used in the character update method to make the blast's hitbox the character's attack box.

```
FUNCTION special move check
        FOR special_move IN character special moves
                IF the three moves are equal to the ones listed in the value list THEN
                        action_taken ← getattr(character, special_move)
                        action_taken()
                        RETURN queue, ""
                END IF
        NEXT special_move
        RETURN None
```

The ki_blasts class:

Since most special moves in Dragon Ball will involve some form of ki blast, I will need to make a class so that the blast can be animated, drawn on screen and return their hitbox to the character to set as their attack box. The ki blast class will inherit from the pygame sprite class so that the images can be drawn on screen and a subclass of the ki blast will be called in the special move method, with taking the character as a parameter; the spritesheet for the blast will be located in the character folder for the character which has that special move. Subclasses are included from ki blasts as each blast will have a different update method e.g Goku's Kamehameha will project horizontally, growing in size and Frieza's Death Ball will take some time to grow, then move with projectile motion.

x y rect image master image hitbox character
update (blank) _getx _gety _setx _sety get_character

Changed in the update method

Kamehameha update method:

The easiest way would be to use a spritesheet and then change its hitbox rect by the width being the smallest width multiplied by the frame number and the rect being the largest hitbox rect, similar to how the character hitboxes and rects work but this time, the hitbox dimensions are being calculated instead of being read from a text file.

In the character update method:

```
IF self.blast THEN
        self.ki_blast .update()
END IF
```

I might need to pass in a sprite group for the ki blast object to be part of so that it can be drawn on the screen, but I'm not sure how to do that

Death Ball update method:

The structure of the subclass will be the same as the kamehameha, as most of the subclasses will be, and the only difference is the update method. The ball will start off by growing in size, and therefore a similar method of obtaining the hitbox from the kamehameha update method will be used. Once at full size, the ball will be projected in a curve motion which can be calculated by getting the x and y components of the velocity and the ball can also be rotated using this; the idea for this was taken from a book[5] and the only thing taken directly from the code was how to rotate an image using pygame.transform. I might also make it so that the death ball does more

[5] *More Python Programming for the Absolute Beginner,* Chapter 6: Spaceship Demo,Jonathan S Harbour, Published by Course Technology PTR in 2011

damage than a regular attack by increasing the character's attack power in the death ball method then changing the attack power back to its original by adding another attribute for each character called original_attack_power, which will let the attack power go back to its original value in the default move method.

Changing the check attack method

In my user requirements , the majority opinion was that playable characters should have unique special moves and have different values for their attack power, defense and health, and for this to also have an effect on the game. Therefore I need to change the check attack method to also include defense and vary the character's health. The change in health value won't be immediately noticed by the player as only the health bar is shown and the player will also not know the differences in attack power or defense and will find out by playing the arcade mode; this is so that the players can try out different characters and decide which one they prefer.

I could also make it so that when a character is blocking, a bit of health is still deducted
(after the main if statement in the check attack method)

```
ELSE IF  [all the other conditions] AND self.blocking = True THEN
        self.set_health(self.get_health() - (opposition.get_attack() // self.get_defense()))
```

I'll make the characters values be:
Goku - balanced; the original attack power and defense of 5 and health 100
Vegeta - increased attack power, and decreased health and defense
Trunks - Increased health, and decreased attack power and defense

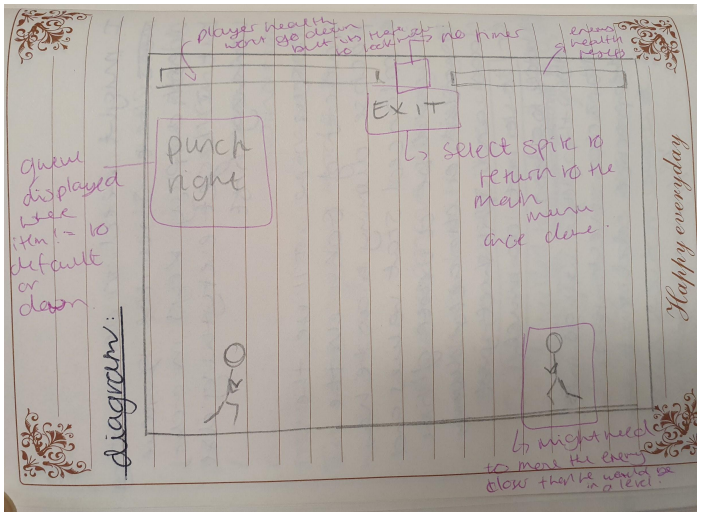
Raditz - Increased health and defense, and decreased health
Frieza - Increased health and attack power, and decreased defense

Creating the Practice mode

Most fighting games will have a practice mode so that the player can figure out special moves and different combination moves with their character. In this practice mode, I will include:

- ➔ An unresponsive "dummy" with its health displayed on screen and when it reaches zero the dummy's health is reset
- ➔ The player's attack queue displayed on screen so that the player can see which keys result in which move
- ➔ The player will go from character select then to the practice mode, and when they want to exit the practice mode, there will be an exit button placed on screen which the player will click to exit.

To save time, I'll use the character of Goku as the dummy as he has balanced values for his attack, defense and health as it would be a waste of time to make a sprite sheet and class for a character which won't do much. The dummy will update by a dummy_reaction function, which will be just the dummy in default - so the default check from the character reaction, but without the in air check



The only new aspect here would be displaying the queue on the screen:

```
FUNCTION get_queue_images
        item_list ← empty list
        FOR item IN queue
                IF item != default AND item != down THEN
                        image ← load image using the action object name to get the path
                END IF
        item_list.append(image)
        NEXT item

        RETURN image_list

PROCEDURE display_queue
        image_list ← get_queue_images()
        FOR image IN image_list
                screen.blit(x, (y + (height * index)), image)
        NEXT image
```

This is so that the images are shown with the first button pressed at the top and the next button below that, and so on. I also need to test for the right x and y values.

The event check for the exit button will have to be included in the set queue function as there can't be two for loops for the same variable in one while loop; the second loop won't run. So

there will have to be a function called setqueue_eventcheck which returns variables for both the player queue and checking if the exit button has been pressed.

Making the pre-level screen

To give the player time to think before the enemy starts attacking, I will include a pre-level screen which will display the text “stage[level number]” then “ready” and then “fight” one after the other for a set amount of time. So in the main pre_level function there will be functions/procedures for:

- ➔ Set up text
- ➔ Normal event check
- ➔ No enemy or player reaction, just default; I can use the dummy reaction from the practice mode
- ➔ Update display
- ➔ Check which text image needs to be displayed according to how much time has passed since the start of the level
- ➔ Loop with a variable set to True or False when there are no images left to display

```
level_start ← False
WHILE NOT level_start
    code for the pre level function

FUNCTION display_text
    IF time_passed < 10s THEN
        display “stage 1”
        RETURN False

    ELSE IF 10s < time_passed < 20s THEN
        display “ready”
        RETURN False

    ELSE IF 20s < time_passed < 30s THEN
        Display “fight!”
        RETURN False

    ELSE
        RETURN True
    END IF
```

And this will be placed in the update screen procedure so it will be in the while loop

Making the end level screen

In the end level screen, a picture of the character will be shown moving up the “enemy tree” (in this prototype, there will only be two enemies) if the player has won the previous level. If the player has lost, then the character icon will be shown discoloured grey and falling off the tree. So therefore, I will need the character icon to have two different ways of updating depending on whether the player has won or lost. The character icon should also be an attribute of the character class and the image will be in the corresponding character folder. And so that the icon can be animated, I should make an icon class, where an instance of the class is made in the character initialise method and the class inherits from the pygame sprite class so that it can be added to a sprite group and drawn on screen.

x y width height master image rect frame outcome
get/setx get/sety get/set outcome set image update

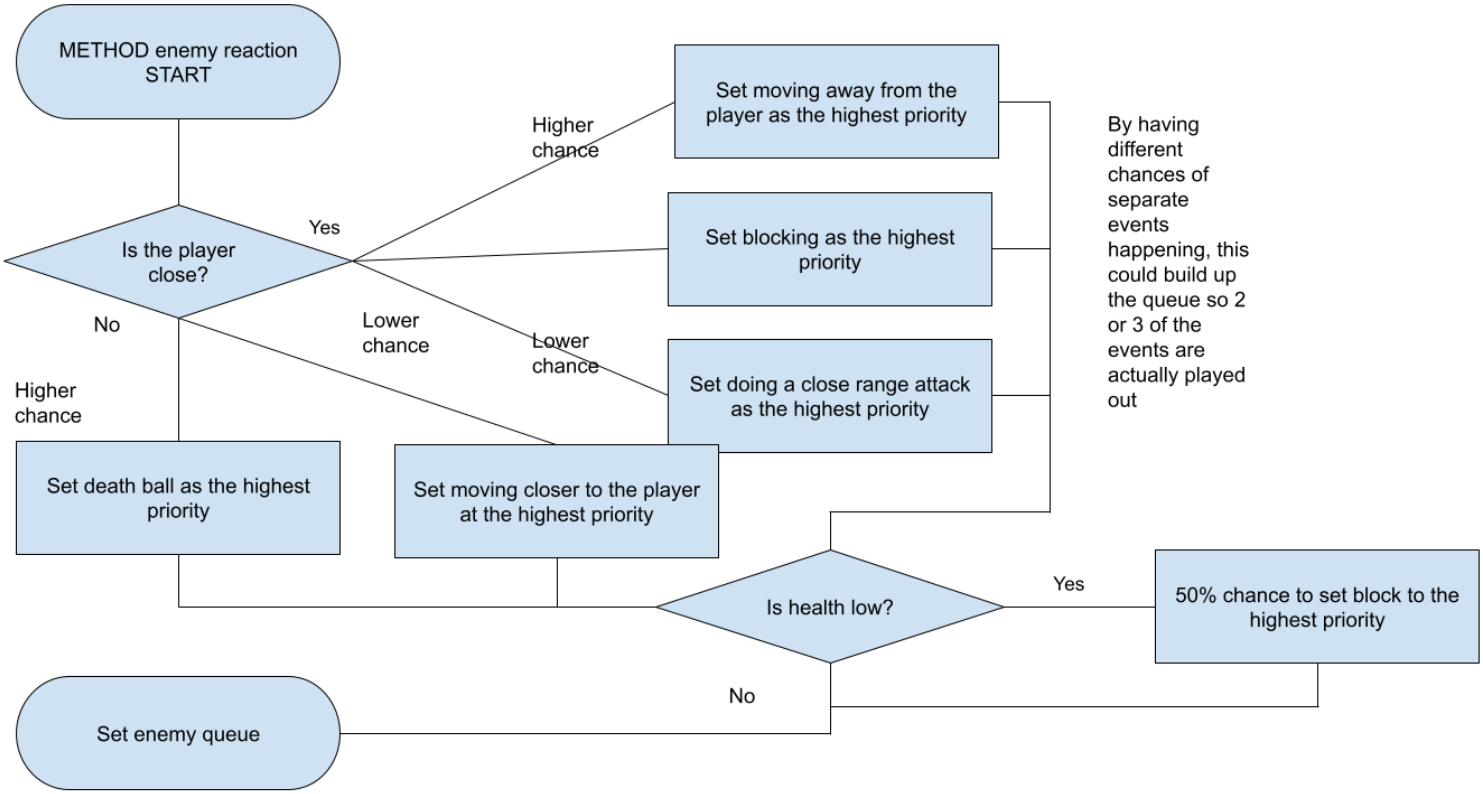
Where outcome if either true or False depending of the player has won or lost the level; for enemy icons, this will be set to None and the update method is only called for the player’s icon

```
METHOD update
    IF outcome = True THEN
        IF a certain amount of time has passed THEN
            move image up
            IF y = next enemy y THEN
                RETURN True
            RETURN False

    ELSE IF outcome = False THEN
        IF a certain amount of time has passed THEN
            move image down
            IF the image has gone off the screen THEN
                RETURN True
            RETURN False
```

The returned booleans show if the end level screen has finished animating

Creating the level 2 enemy response

The enemy for this level is Frieza, following the season-end boss battles from Dragon Ball Z. Frieza is a small character and therefore won't move close to the player as frequently as Raditz; Frieza will make this level harder than the previous as he will have an attack power between 10-20 and his Death Ball special attack takes a lot of damage if the player doesn't black or dodge the attack. To balance this out, Frieza has the lowest defense



I will need to change the enemy set detect box and set smaller detect box methods, so that in the parent enemy class it sets out the box, then the sub enemy classes will call the super method then realign the boxes after that.

Organising the level function

Since there is now a 2nd level where large sections of code will be repeated, I need to section out the level1 function and then use that same format with level2 functions/procedures. Inside each level function will include:

- ➔ Set up which returns the sprite lists
- ➔ Main game which returns booleans

- ➔ Time out and defeated which also return booleans

Also, some variables will need to be set inside the level function to avoid returning too many variables in level setup, like sprite groups and variables. I could also add a set enemy function which will take the level number as a parameter and return the enemy on that level by looking through a list of enemies and adding an attribute called level number.

```
FUNCTION set_enemy
        level ← current level number
        Initialise each enemy
        enemies ← list of enemies
        FOR enemy IN enemies
                IF enemy.get_level() = level THEN
                        RETURN enemy
                ELSE
                        NEXT enemy
                END IF
```

I also need to utilise the get_healthbar methods and methods similar so that I am not setting variables which I do not need, and therefore reducing the amount of variables needing to be returned.

Making the actual prototype

There's a lot of little changes/additions to be made in this prototype, so I won't list each individual one but rather group them together:

- ➔ Any changes to the character classes
- ➔ Set up, interaction and update functions for the new modes
  - ◆ Practice mode
  - ◆ End level screen
  - ◆ Pre level screen
- ➔ Level sub procedures and functions
- ➔ Level1 and level2 level functions
- ➔ Organise new changes into the existing modules
- ➔ Put the new modes into the main code.

Prototype 5

Objectives:

- Saving and loading player data so that enemy characters can be unlocked to play as once that specific level has been completed
- The unlockable character select screen
- Add background music and sound effects to the game
- Level 3 enemy response
- Special moves for Vegeta and Trunks
- Controls screen
- Client improvements:
    - Decreasing the pixels moved when right/left is pressed
    - Changing the chances on Frieza's choice of attack
    - Looking and fixing the jump bug and the wrap-around screen; the player should stop at the screen boundary.

Saving and loading player data using a database

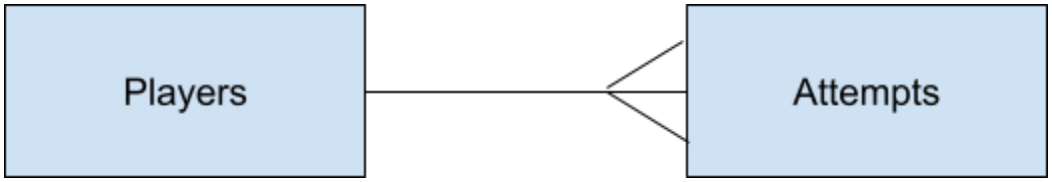
From the user requirements, most people said that they wanted to be able to play as the enemy they have defeated, and in order to do that data on the player must be stored and then retrieved to check if the player has unlocked any of the enemy characters.

The data that would need to be saved would be:

- The player's name, so that when that name is entered, the data associated with that name will be retrieved
- How many levels the player has beaten, and therefore which character's they have unlocked
- Maybe which character they have used when playing, so that in later versions of the game, trends on each playable character can be mapped
- Each attempt a player has done with all this information included

So therefore I will include two tables in this database: 1 main table including the player names and a unique player ID which will link to a table called attempts, which have all the attempts of the arcade mode by all players and with all the data stated prior except for the player name. The reason for separating this into two tables is so that the program can quickly find if, when a player name is entered, a new player row needs to be created or if data needs to

be retrieved for that player. Databases can be created and manipulated in python using the SQLite library.



I will make a save data class, where the set and get methods will mostly be database queries, with exceptions being some of the set methods. An instance of the save data class is created at the beginning of the main code and data is loaded into the save data object if the player clicks “Load save data” in the arcade mode and if “new game” is selected then the save data object is updated with the player name and character. At the end of each level, the level number is updated and when the arcade is complete, by either losing or winning the game, data is saved. When data is loaded for a player, the player can’t continue from their last level completed; loading the levels completed will only allow for the enemy characters to be unlocked to play as.

When the player clicks either load data or new game at the beginning of the arcade mode, the player will enter their name and this will be displayed on screen using the pygame text function and so there has to be set up, interaction and update display functions for the “start arcade” screen.

The save data class:

name character_name level_number (ID isn’t needed as this will autoincrement and can be retrieved using a get method query) raditz_unlocked frieza_unlocked cell_unlocked (variable needed for the database to work)
get/set_player_name set_character_name (won’t need get yet) get/set_level_number create_tables check_data

Booleans set through set characters unlocked method using the level number attribute
There will also be a check data function to create the tables if they aren't already there

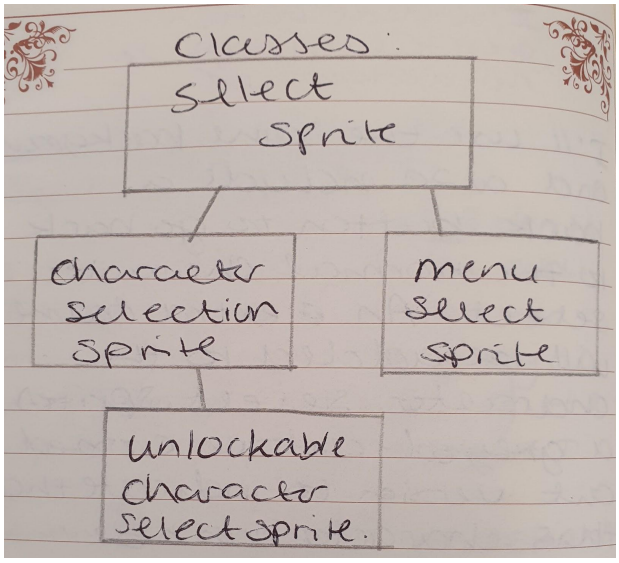
```
FUNCTION check_data
        data ← save_data()
        TRY
                data.check_data()
        EXCEPT
                data.create_tables ()
        RETURN data
```

Where the check data method perform a simple query and if the database doesn't exist, the query will error

The unlockable character select screen

When a player defeats a level, they will be able to select the character they defeated to play as in the next attempt of the arcade mode. I will create a main function called unlockable character select which will be called within the original character select function if a select sprite for more characters is clicked. Since there are 3 unlockable characters, I can use functions from the character select, but the update function will have to be different. I'll use the same background and also include a back select sprite to go back to the normal character select. Since the unlockable character's select sprites will need an extra frame to show that the player can't select them, a new subclass will be made inheriting from the normal character select sprite. The only changes will be the extra frame, and therefore a polymorphed method of set frame, and a boolean for whether the character represented has been unlocked by the player.

Class diagram:



```
METHOD set frame
        IF NOT self.character_unlocked THEN
                self.frame ← 2
        ELSE IF self.check_click() THEN
                self.frame ← 1
        ELSE
                self.frame ← 0
        END IF
```

Where the character unlocked attribute is passed into the initialise and is retrieved from the save data select_[character name] method

This therefore means that data will have to be loaded before the practice mode as well, to make the character select work and so that the player can practice as the characters they have unlocked.

Adding special moves for Vegeta and Trunks

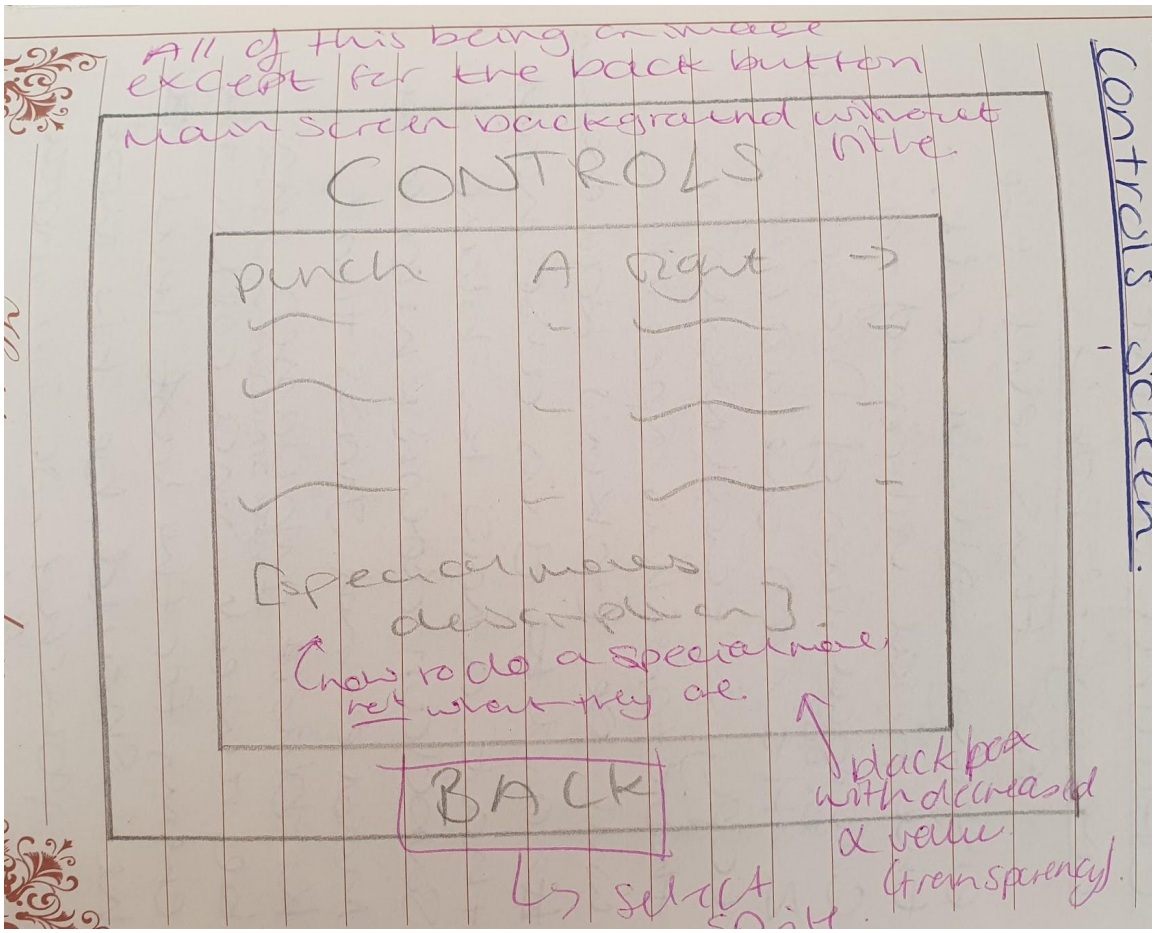
For Vegeta I'll add his "galick gun" as his special move, which is blast similar to the kamehameha but purple and it's the first special move that Vegeta performs in the show. Since it is similar to the kamehameha, most of the code can be copied into a galick_gun class - inheriting from the ki_blasts class - and then frame numbers can be changed.

And for Trunks I will use "shining slash" which is also the first special move Trunks displays. The shining slash has two parts: one where Trunks is attacking with his sword and the next where he shoots a small close range blast. The hitboxes and movement for Trunks, as there is a portion of animation where Trunks needs to jump and then fall back down, will have to be done as part of the ki blast update method in order to not overcomplicate the character update method; this also means that the blast can't be killed once is has hit as the special move has a few opportunities to hit the enemy, and ending the ki blast will stop the movement in the animation and the hitboxes. For the portion of animation where Trunks needs to jump and fall back down, the movement method from the character class is called directly with the correct parameters as if it was called in character update like it normally would. I will also change the attack power to 10 when this special attack is called, since Trunks has the lowest attack power.

The Controls screen

In the main menu, I have already included the menu select sprite to view the controls screen but I haven't added this screen in yet; this is so that the player knows the basic controls of the game. There will be a little bit of programming involved with this screen, as it is just there to display information, but I will need a button event check which I have already programmed and I will need to program the setup to load the image of the controls screen

Diagram of the controls screen:



Creating the level 3 enemy response

For this level, the enemy will be Cell - the final villain of the original run of Dragon Ball Z. In the story, Cell has the ability to copy special moves, so therefore his special move will be Goku's Kamehameha, and this also means that a new subclass doesn't have to be made and this also shows how the subclasses can be reused if the frames of animation are in the same format. In the comics, there are three forms of Cell and I'll be using his first form, and that is why his attack will be pretty low but have high defense and health values; from a game perspective, this makes it so that Cell isn't hard to defeat but it will take a lot of time, putting time pressure on the player as there is a time limit of 300 seconds. Cell won't move as quickly as Frieza, so he will have a few different sets of close range attacks, making it so that some of his attacks can be predicted by the player, and also including blocking in some of the close attack combinations. Once Cell has a health value below 25% of his original health, he will move away from the player and have a higher percent chance to block, and a percent chance to attack using the kamehameha - this is the same style used in the comics as when Cell (first form) was about to get defeated by the main characters, he would perform a special move and exit the situation.

Adding background music and sound effect to the game

Adding background music to the game will be pretty easy as I can use the music related pygame functions and procedures to load and play music. In the user requirements it was a majority opinion to have upbeat music which changes depending on the level, so therefore I will need a function which returns the music needed by taking the level number as a parameter - which will also mean that music will be organised either in different files or have their level number in the file name. The background music for the select screens will just be placed in the main functions with a play --- music functions in the set up module. I will use music from the Dragon Ball Xenoverse series of games or Dragon Ball Fighterz. With adding sound effects in the pygame music procedures, sound files have to be loaded and played in different channels to be able to be played at the same time, so in the methods for the select sprites, sound effects have to be loaded into a different challen and played in the methods which require sounds effects.

Making the actual prototype

Functions added/changed in order to make this prototype work:

- setup , interaction and update methods for:
    - Controls screen
    - Loading data
    - Second character select
- Insert loading data into practice and arcade mode
- Add music and sounds effects play functions/methods and add to main functions
- Add Cell to the end level screen

Prototype 6

Objectives:

In this prototype, there won't be any new features added into the game as I think everything that can be included is included. The previous prototype showed a complete product, but there are a few things that can be fixed and/or changed:

- Fix previous errors in code where possible
- Make sure that the enemies are easy enough to beat and aren't awkward
- Any final client requirements:
    - Fix the errors Curtis found

Looking at, and maybe fixing, mistakes

Methods in the character subclasses:

Some of the get methods for characters are repeated in each subclass, so I should move these to the parent class. And the values attribute (the list of attributes like the default y position, for example) isn't actually used a lot in the code and can be replaced with the get methods, and so therefore the values attribute should be removed for the program to be more space efficient.

Simplifying the main code:

Since the code for the arcade mode is quite long and repetitive, I should simplify this; this can be done by creating a recursive function which returns the player data at the end of the function

```
FUNCTION arcade_mode(screen, player, player_data, level number)
        win ← level(screen, player, level_number)
        IF win = True THEN
                IF level_number = 3 THEN
                        player_data.set_level(3)
                        player_data.save()
                        RETURN player_data
                ELSE:
                        player_data ← arcade_mode(sc, player, player_data, (level_number + 1))
                END IF
        ELSE
                player_data.set_level(level_number)
                player_data.save()
                RETURN player_data
        END IF
```

This was my initial idea but- looking at this again - this can be simplified again by combining some of the if statements which will decrease the execution time:

```
FUNCTION arcade_mode(screen, player, player_data, level_number)
        win ← level(screen, player, level_number)
        IF win = True THEN
                player_data ← arcade_mode(screen, player, player_data, level_number+1)
        ELSE IF win = False AND level_number = 3 THEN
                player_data.set_level(level_number)
                player_data.save()
                RETURN player_data
```

Checks with the queue:

Since there are two reverse() lines called, this is probably redundant or causes the queue not to work properly, so i should remove that and adjust how the functions depending on how this works.

Adding the character name to the save data object:

I forgot to include this in the previous prototype, but the character name isn't used in any of the prototypes of this game. In a later version of the game, the data for each character can be analysed to see if there is a trend correlating a character and how successful or unsuccessful they are at beating the arcade mode, and then adjustments can be made to balance out characters if this seems like an unfair advantage/disadvantage.

Fix the kamehameha and the galick gun special moves:

Both of these special moves don't work at a close range when the character is reversed, but work properly when the character is not reversed; this is probably a problem with the hitboxes that needs to be corrected. And, sometimes when activating a special move, the move will repeat with the player not pressing any buttons; I could fix this by clearing the queue after a special move has been activated, like how it is done with the combination moves.

Including repeated characters in the player name:

When I was programming the set_name function, the player_name string would repeat letters added to the letters list, so I changed it so that letters couldn't be repeated to solve this issue. But I think I can make repeated letters work if i use the pygame key pressed booleans:

```
FOR letter IN letters
        FOR event IN pygame events
                letter _bools ← pygame key booleans

        [same code as before until letter if statement]
                IF True IN letter_bools THEN
                        Add the letter to player name
                END IF
[same code as before]
```

Scrolling character selection:

A tester found that a select sprite can be clicked using the mouse scroll-wheel, but when scrolling too fast the game errors, saying that more than the correct number of booleans were returned. I should change all the check click related functions to only respond to the mouse buttons and not the scroll wheel

The character select screen:

If a character is selected on the second screen, the program would return to the first screen before returning the character. This can be fixed by moving the return statement in the character select main function, to be in the loop “if player_added” so that the function is ended exactly when a character is selected

The enemies of the arcade mode

The tester said that it was weird for the two characters to go into a corner and continuously activate their special moves, so I thought I should fix this. For Frieza, I won’t fix this because he is an enemy who is already hard to beat and an attack strategy against Frieza would be to make him throw the death ball, but while he is charging it, the player can attack the enemy. For Cell - I do still want to include this in the game because it would be much more easier for the player to beat the level once unlocking Frieza, creating replayability - but I will try to fix this by adding:

- An addition to the chance variable if Cell is at x values less than 100 and greater than 900
- An option where Cell moves closer to the player in a higher chance value bracket
    - And this can still be chosen even if Cell is in the correct x value range.

Development

Prototype 1

Fixing the Animation methods

Frame numbers:

When testing the animation, the character would miss out the last frame of animation so I set the move animation's last frame to the previous move animation's first frame; this made the character animate properly.

Default mode:

By writing WHILE NOT player.get_animating(): player.default() the character would be continuously set to the first frame of the default animation and therefore wouldn't animate properly. So I changed it to:

```
#if the player isn't doing anything:
if not player.get_animating():
    if player.get_default_mode():
        #^ to make sure that the first frame isn't reset
        pass
    else:
        player.default()
```

I added an attribute called default mode to the character class which is set to True when the character is not moving, attacking or blocking and it is set to False if it is doing something.

Directional movements:

When a key is pressed and a character moves, the animation happens while they are at one point instead of animating while they are moving. To fix this, I added another variable called "mode" which is used in the update method

In the sprite interactions module, the player reaction function now returns the mode variable - which is a string being "", "right", "left", "down" or "jump".

```
#directional keys
if user_input[K_UP]:
    player.jump()
    mode = "jump"

if user_input[K_DOWN]:
    player.crouch()

if user_input[K_RIGHT]:
    if player.get_reversed():
        player.move_left()
    else:
        player.move_right()
    mode = "right"

if user_input[K_LEFT]:
    if player.get_reversed():
        player.move_right()
    else:
        player.move_left()
    mode = "left"
```

I also added a method in the character class which checked if the character sprite needed to turn around because they were at the other side of the screen:

```
def set_reversed(self, oposition): #character object
    if self._getx() >= oposition._getx():
        self.reversed = True
    else:
        self.reversed = False
```

In the update character method, mode is used to decide which way the character is moving if mode != “”

```
            if mode != "":
                pos = self.movement(mode, x, y)
                x = pos[0]
                y = pos[1]

            else:
               self._setx(x)
               self._sety(y)

            self.set_hitbox(x, y)

    def move(self, dx, dy, x, y):
        self._setx(x + dx)
        self._sety(y + dy)
        return (self._getx(), self._gety())
        #and the hitbox will be updated in the update() method

    def movement(self, mode, x, y):
        #making sure the character moves gradually when animating
        if mode == "right":
            return self.move(7.5,0, x, y)

        if mode == "left":
            return self.move(-7.5,0, x, y)

        if mode == "jump":
            return self.move(0, -3, x, y)

        if mode == "down":
            return self.move(0, 4.18, x, y)
```

When the character had mode = “jump” I couldn’t get the gravity working within the method as different characters are different heights so this is included in the player reaction function:

```
#need to test for the number
    if player._gety() <= 234:
        player.back_down()
        mode = "down"
```

Changing the update method:

The update method was changed so that the “default” and “defeated” methods would be continuous and every other method would go through once

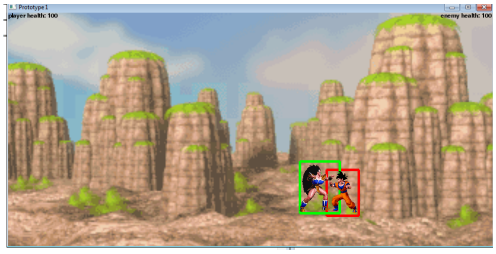
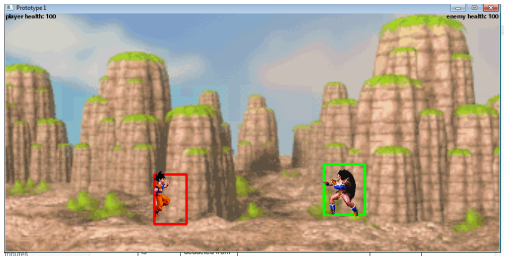
```
def update(self,current_time, rate, x, y, mode): #x and y resets the movemen
    #update the animation frame number
    self.old_frame = self.frame - 1
    if current_time > self.last_time + rate:
        self.frame += 1
        if self.frame >= self.last_frame:
            if self.animating:
                self.animating = False
            else:
                #default and defeated animations:
                if self.frame > self.last_frame:
                    self.frame = self.first_frame
        self.last_time = current_time
```

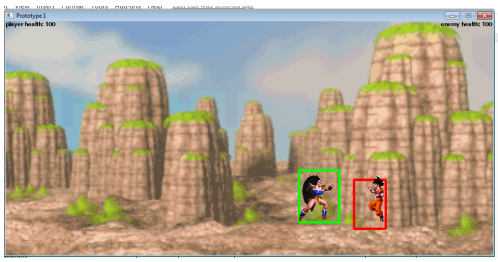
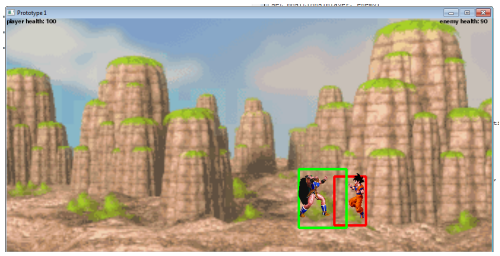
Attacking a character

```
def check_attack(player, enemy, group): #not sure if i need the spite group
    #player attack
    if player.get_attacking() and pygame.sprite.collide_rect_ratio(0.9)(player,enemy) and not enemy.get_attacked():
        enemy.set_health(enemy.get_health() - player.get_attack_power())
        enemy.hit()

    #enemy attack
    if enemy.get_attacking() and pygame.sprite.colide_rect_ratio(0.9)(player,enemy) and not player.get_attacked():
        player.set_health(player.get_health() - enemy.get_attack_power())
        player.hit()
```

This is the code I wrote for how health would be deducted when a character is attacking; the attack power and health is set in the “set positions” procedure and in later prototypes I want each character to have a unique attack power, health and defense values.

Tests for characters attacking:

Condition	Expected outcome	Screenshot	Successful?
The characters are on screen; one is moving but not attacking and the other is in default	No health is deducted from either character	 Text on screen: Player health: 100 Enemy health:100	Yes
The player and the enemy's hitboxes overlap and neither are attacking	No health is deducted from either character	 Text on screen: Player health: 100 Enemy health: 100	Yes
The player is attacking but hitboxes aren't close	No health is deducted from either character	 Text on screen: Player health: 100 Enemy health: 100	Yes

The player is attacking and hitboxes are close but not touching	No health is deducted from either character	 Text on screen: Player health: 100 Enemy health: 100	Yes
The player is attacking the enemy with hitboxes touching	The enemy will lose health and animate “hit”	 Text on screen: Player health: 100 Enemy health: 90	Partially: the enemy does lose health but it acts as if it got attacked twice, so 10 health is removed instead of 5
The player has a series of successful hits to the enemy	When the enemy’s health decreases to below 0, the “defeated” animation is on the enemy and the player is stuck on default. “You won” is displayed on the screen	 Video showing this working	Yes

Added attributes

I added a few boolean attributes to the character class that the game would run properly:

Name	Description	In code

Blocking	When either character is blocking, health can't be subtracted	`def block(self):`
		`  self.frame = 6`
		`  self.last_frame = 10`
		`  self.animating = True`
		`  self.default_mode = False`
		`  self.blocking = True`

Attacking	When one character is attacking, health can be subtracted	`def punch(self):`
		`  self.frame = 25`
		`  self.last_frame = 31`
		`  self.animating = True`
		`  self.default_mode = False`
		`  self.attacking = True`
		
		`def check_attack(player, enemy, group): #not sure if i need the spite group`
		`  #player attack`
		`  if player.get_attacking() and pygame.sprite.collide_rect_ratio(0.9)(player,enemy)`
		`  enemy.set_health(enemy.get_health() - player.get_attack_power())`
		`  enemy.hit()`
		
		`  #enemy attack`
		`  if enemy.get_attacking() and pygame.sprite.colide_rect_ratio(0.9)(player,enemy) a`
		`  player.set_health(player.get_health() - enemy.get_attack_power())`
		`  player.hit()`

dead	When a character dies, dead is set to True to allow for "defeated" to be set to True	`def set_dead(self):`
		`  if self.health < 0:`
		`  self.dead = True`
		`  else:`
		`  self.dead = False`

```
def check_defeated(player, enemy):
    player.set_dead()
    enemy.set_dead()

    if player.get_dead():
        player.defeated()
        enemy.default()
        return True

    if enemy.get_dead():
        enemy.defeated()
        player.default()
        return True

    else:
        return False
```

(character object.defeated being a “move” method)

```
        defeated = check_defeated(player, enemy)
        start_time = get_start_time(ticks, defeated)

    while defeated:
        ticks = pygame.time.get_ticks()
        play = defeated_update(screen, background, pl
        if not play:
            defeated = False

if not play:
    exit_game()
```

(this is in the while play loop)

The remainder of the code

set_up.py:

```
def setup_screen():
    pygame.init()
    screen = pygame.display.set_mode((997,473),0,32)
    pygame.display.set_caption("Prototype 1")
    return screen

def load_background(name):
    path = os.path.join("spritesheets", "Background1", name)
    background = pygame.image.load(path)
    return background
```

In future prototypes the background should change depending on the level the player is on and the folder “Background1” being for the first level, so I will have to update the function to allow for that to be checked.

```
def load_character(name, screen):
    character1 = character(name, screen)
    filename = "%sSheet.png" %(name)
    path = os.path.join("spritesheets", name, filename)
    dimentions = get_dimentions(name)
    width = int(dimentions[0])
    height = int(dimentions[1])
    character1.load(path, width, height, 75)
    return character1

def get_dimentions(name):
    #like the get_hitbox() method
    path = os.path.join("spritesheets", name, "dimensions.txt")
    file = open(path, "r")
    for line in file:
        line = line.split(",")
        if int(line[0]) == 75:
            dimentions = (line[1], line[2])
            #^ width, height
    return dimentions
```

All the characters have a file named with their character name and their sprite sheet is called [Their name]Sheet.png, so that makes getting the sheet quicker as only the name needs to be used as a parameter. Each character folder also has a text file called “dimensions.txt” which has the dimensions of each frame for the hitbox, but the last line - line 75 as all the characters have a total of 74 frames - has the equally spaced squares dimensions so that the my_sprite class can easily divide the sheet and get the frames.

```
def set_positions(player, enemy):
    #player to enemy height difference = 15 at default

    player._setx(300)
    player._sety(295)
    player.set_health(100)
    player.set_attack_power(5)
    player.default()

    enemy._setx(600)
    enemy._sety(275)
    enemy.set_health(100)
    enemy.set_attack_power(5)
    enemy.default()
```

```
def add_group(characters):
    #characters = list of the characters on screen
    group = pygame.sprite.Group()
    for item in characters:
        group.add(item)
    return group
```

In this prototype I used trial and error to figure out where the characters should be placed for their y positions, since the characters are different heights. In the next prototype I will create sub-classes of the character class for each of the characters in the game with attributes like their highest y point . The characters are then added to a sprite group so that they can be drawn on the screen at the same time instead of individually.

```
def print_text(screen,  x, y, text,size, colour = (0,0,0)):
    font = pygame.font.Font(None, size)
    imgText = font.render(text, True, colour)
    screen.blit(imgText, (x,y))
```

I also included a print text procedure for displaying the character's health and whether you have lost or won. In later prototypes I will include a health bar for the character's health and maybe an image of text saying if you have won or lost because that would look better.

sprite_interactions.py:

```
def player_reaction(player):
    mode = ""
    for event in pygame.event.get():
        if event.type == QUIT:
            exit_game()

        user_input = pygame.key.get_pressed()

        #combination keys first - these don't work
        if user_input[K_a] and user_input[K_DOWN]:
            player.low_punch()

        if user_input[K_x] and user_input[K_DOWN]:
            player.low_kick()

        if user_input[K_z] and user_input[K_DOWN]:
            player.low_block()

        if user_input[K_a] and user_input[K_UP]:
            player.uppercut()

        if user_input[K_x] and user_input[K_UP]:
            player.high_kick()

        #single keys
        if user_input[K_a]:
            player.punch()

        if user_input[K_z]:
            player.block()

        if user_input[K_s]:
            player.chop()

        if user_input[K_x]:
            player.kick()

        #directional keys
        if user_input[K_UP]:
            player.jump()
            mode = "jump"
```

```
            if user_input[K_DOWN]:
                player.crouch()

            if user_input[K_RIGHT]:
                if player.get_reversed():
                    player.move_left()
                else:
                    player.move_right()
                mode = "right"

            if user_input[K_LEFT]:
                if player.get_reversed():
                    player.move_right()
                else:
                    player.move_left()
                mode = "left"

        #if the player isn't doing anything:
        if not player.get_animating():
            if player.get_default_mode():
                #^ to make sure that the first frame isn't reset
                pass
            else:
                player.default()

    #need to test for the number
        if player._gety() <= 234:
            player.back_down()
            mode = "down"

        mode = (mode, "") #"" for the enemy
        return mode
```

The player reaction functions controls what animation the player is doing and what “mode” the player is in and since the enemy won’t be moving in this prototype, the mode for the enemy stays at “” - but in a future prototype I will put the enemy’s mode into the enemy_reaction procedure. The enemy reaction procedure just sets the enemy to default when not being attacked.

```
def stay_on_screen(player, enemy):
    #player first:
    if player._gety() < 0:
        player._sety(275)

    if player._getx() > 980 or player._getx() < 0:
        player._setx(0)

    #enemy:
    if enemy._gety() < 0:
        enemy._sety(275)

    if enemy._getx() > 980 or enemy._getx() < 0:
        enemy._setx(0)
```

This just makes sure that the characters stay on the screen

And check_attack() is also included in this module

update_display.py:

```
def update_screen(screen, background, player, enemy, ticks, group, mode):
    screen.blit(background, (0,0))
    pygame.draw.rect(screen, (255,0,0), player.get_hitbox(), 5)
    pygame.draw.rect(screen, (0,255,0), enemy.get_hitbox(), 5)
    player.set_reversed(enemy)
    player.update(ticks, 180, player._getx(), player._gety(), mode[0])
    enemy.set_reversed(player)
    enemy.update(ticks, 180, enemy._getx(), enemy._gety(), mode[1])
    group.draw(screen)
    print_text(screen, 0, 0, "player health: %d" %(player.get_health()), 18)
    print_text(screen, 890,0, "enemy health: %d" %(enemy.get_health()), 18)

    pygame.display.update()
```

The update screen procedure just makes sure that everything is displayed on the screen. When either character is defeated, the game uses a different screen update method:

```
def defeated_update(screen, background, player, enemy, ticks, group, start_time)
    screen.blit(background, (0,0))
    pygame.draw.rect(screen, (255,0,0), player.get_hitbox(), 5)
    pygame.draw.rect(screen, (0,255,0), enemy.get_hitbox(), 5)

    if enemy.get_dead():
        print_text(screen, 498.5, 236.5, "YOU WON", 40)
    if player.get_dead():
        print_text(screen, 498.5, 236.5, "PLAYER DEFEATED", 40)

    player.set_reversed(enemy)
    player.update(ticks, 180, player._getx(), player._gety(), "")
    enemy.set_reversed(player)
    enemy.update(ticks, 180, enemy._getx(), enemy._gety(), "")
    group.draw(screen)
    pygame.display.update()

    if ticks > start_time + 4000:
        return False
    else:
        return True
```

```
def get_start_time(ticks, defeated):
    if defeated:
        return ticks
    else:
        return 0
```

This is different to the normal update method as the player can't move (because mode is set to "" and it is in a separate loop which doesn't include player_reaction) and after 4000 ticks after the character has been defeated, "defeated" in the main code will be set to False, which will cause the game to exit using a exit_game() procedure

main.py:

After importing everything:

```
play = True
defeated = False

screen = setup_screen()
background = load_background("background.png")
framerate = pygame.time.Clock()
player = load_character("Goku", screen)
enemy = load_character("Raditz", screen)
characters = [player, enemy]
group = add_group(characters)
set_positions(player, enemy)


if play:
    while not defeated:
        framerate.tick(30)
        ticks = pygame.time.get_ticks()

        if not player.get_animating():
            mode = player_reaction(player)
        enemy_reaction(enemy)
        stay_on_screen(player, enemy)
        check_attack(player, enemy, group)
        if player.get_default_mode():
            player._sety(295)
        update_screen(screen, background, player, enemy, ticks, group, mode)

        defeated = check_defeated(player, enemy)
        start_time = get_start_time(ticks, defeated)

    while defeated:
        ticks = pygame.time.get_ticks()
        play = defeated_update(screen, background, player, enemy, ticks, group,
        if not play:
            defeated = False

if not play:
    exit_game()
```

The “if player.get_default_mode(): player._sety(295)” is there because I couldn't get the character to return to the exact same y coordinate after jumping, so i tried to get it as close to 295 as possible then moved the character

Prototype 2

Fixing the combination key recognition

When my pseudocode was translated into python code, an action which required only one key needed to be pressed twice for it to be recognised (since the length of the queue has to be >= 2 to assign a current move and a check move). I changed this to:

```
if len(action_queue) > 1:
    current_move = action_queue[0]

    #so that the program skips default:
    next_in = 1
    try:
        while action_queue[next_in].get_name() == "default":
            next_in += 1
    except: #list is too small
        next_in = 1
    check_move = action_queue[next_in]


elif len(action_queue) <= 1:
    current_move = action_queue[0]
    check_move = None

else: #nothing in the queue
    return "", action_queue
```

The first section of code in the player_reaction function sets the current_move to the first action in the queue and the check_move to the next item which isn't default if the queue length is greater than 1. The try and except statement is there in case the list index is out of range, so the queue could be - for example [punch, default, default, default] and the check_move will be set to default (which will be later blocked by an if statement). If the length of the queue is less than or equal to 1 the check_move is set to None; this will be used later in the if statements checking the combination moves. And the else returns no mode and the queue that was passed into the function, since the queue will be too small to check. In a later prototype, I should separate the player reaction function into separate functions.

```
if check_move != None:
    if not current_move.get_movement() and check_move.get_movement() or
        if current_move.get_name() != "default" or check_move.get_name()
            if check_move.get_name() == "jump":
                if current_move.get_name() == "punch":
                    player.uppercut()
```

[continues to check all combinations]

```
#if there isnt a second move to check
if current_move != None and check_move == None:
    if not current_move.get_movement():
        if current_move.get_name() == "punch":
            player.punch()
```

The function then checks through for which action was intended by the player, like planned in the pseudocode.

Hitboxes and Attack-boxes

First, I needed to fix the original hitbox which is around the character, so I added a few boolean attributes and if statements to move the box to fit the character as closely as possible.

```
self.in_air = False
self.low = False
self.high_attack = False
#^to fix the original hitboxes
```

And then in the set_hitbox method:

```
if self.reversed:
    x += self.hitbox_width // 2

if self.in_air:
    self.hitbox_height -= self.hitbox_height * 0.2

if self.low:
    y += self.hitbox_height // 2
    self.hitbox_height -= self.hitbox_height * 0.2

if self.high_attack:
    y -= self.hitbox_height * 0.2

if self.name == "Trunks":
    y -= 15

self.hitbox = pygame.Rect(x,(y+(self.hitbox_height // 4)) ,
```

I decided the addition and subtraction values through trial and error.
And secondly with the attack boxes, the measurements I predicted in the diagrams I had drawn for the design weren't completely correct. So, I entered the values I predicted for the character

of Goku and then changed it using trial and error and for the other characters, I entered Goku's attack boxes and then changed them, again using trial and error, to fix that character. Also, I needed to make seperate hitboxes if the character is reversed since the image is flipped and moved x-self.width.

```
def set_box_lists(self,x,y):
    if not self.reversed:
        self.punch_boxes = [None,None, None, pygame.R
        self.kick_boxes = [None,pygame.Rect((x+80),(y
        self.chop_boxes = [None,None,None,pygame.Rect
        self.low_punch_boxes = [None,None,pygame.Rect
        self.low_kick_boxes = [None,None,pygame.Rect(
        self.uppercut_boxes = [None,None,None,pygame.
        self.high_kick_boxes = [None,None,None,pygame

    if self.reversed:
        x -= self.hitbox_width

        self.punch_boxes = [None,None, None, pygame.R
        self.kick_boxes = [None,pygame.Rect((x+70),(y
        self.chop_boxes = [None,None,pygame.Rect((x+7
        self.low_punch_boxes = [None,None,pygame.Rect
        self.low_kick_boxes = [None,None,pygame.Rect(
        self.uppercut_boxes = [None,None,None,pygame.
        self.high_kick_boxes = [None,None,None,pygame


    self.attack_boxes = [self.punch_boxes,
                         self.kick_boxes,
                         self.chop_boxes,
                         self.low_punch_boxes,
                         self.low_kick_boxes,
                         self.uppercut_boxes,
                         self.high_kick_boxes]
```

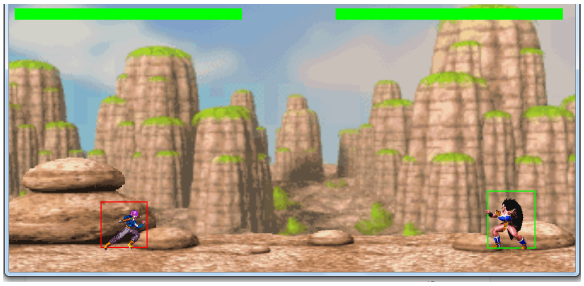
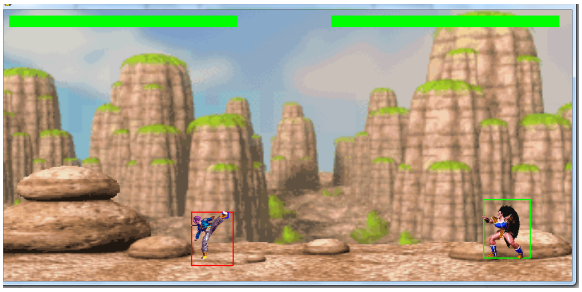
Collision Detection

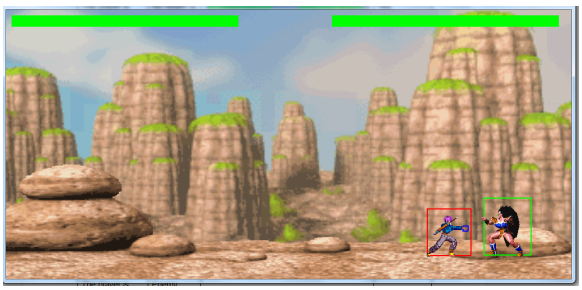
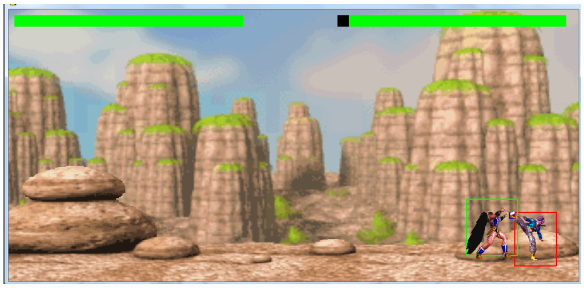
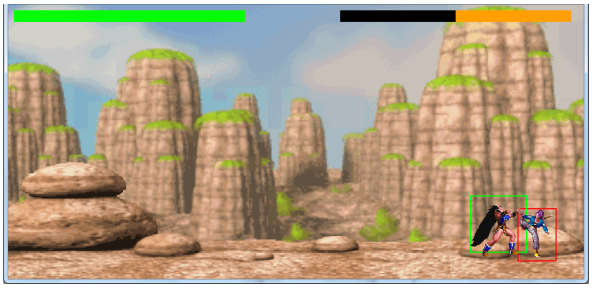
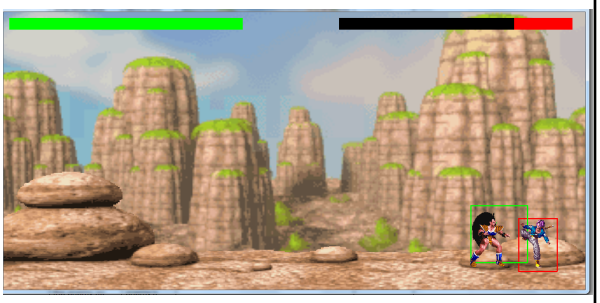
Fixing the collision detection:

Using the pygame collide_rect function doesn't work since it doesn't take a specific rect as a parameter; the object is passed and the function uses object.get_rect(). I tried to set the hitbox and attack_box as the character's rect at certain points (e.g IF self.attacking THEN self.rect ← self.attack_box) but this didn't work as it moved the image of the character to the specified x and y value. So I changed the collide_rect function to:

```
def collide_rect(self, opposition):
    #overwrite sprite collide rect
    return opposition.get_attack_box().colliderect(self.get_hitbox())
```

Testing the collision:

Condition	Expected Outcome	Screenshot	Successful?
The characters are on screen; one is moving but not attacking and the other is in default	No health is deducted from either character		Yes
The player is attacking but hitboxes aren't close	No health is deducted from either character		Yes

The player is attacking and hitboxes are close but not touching	No health is deducted from either character		Yes
The player is attacking the enemy with hitboxes touching	Enemy attacked once and health bar is decreased due to the decrease in health		Yes, this has improved from prototype 1 where the enemy was attacked twice
The player hits the enemy so that the enemy's health is below 50%	The enemy's health bar turns orange		Yes
The player hits the enemy so that the enemy's health is below 25%	The enemy's health bar turns red		Yes
The player has a series of successful hits to the enemy causing the enemy's health to reduce to 0%	The screen displays "You Won" and the enemy does a defeated animation		Yes, but for a second you can see the coloured health bar drawn going backwards.

The remainder of the code

set_up.py:

For the setup of the character select screen, the three characters and select sprites are initialised, their positions are set, and the lists of characters and select sprites are returned to make the sprite groups.

```
#------------------character selection---------------

def character_select_positions(screen):
    goku = Goku(screen)
    raditz = Raditz(screen)
    trunks = Trunks(screen)

    goku.default()
    raditz.default()
    trunks.default()

    gx, gy =(goku.get_select_x(), goku.get_default_y())
    rx, ry = (raditz.get_select_x(), raditz.get_default_y())
    tx, ty = (trunks.get_select_x(), trunks.get_default_y())

    goku._setpos((gx, gy))
    raditz._setpos((rx, ry))
    trunks._setpos((tx, ty))

    image_path = os.path.join("spritesheets", "select_sprites", "select_sprite.png")
    #^change image within class when sprite is made
    select_goku = selection_sprite(screen, goku,image_path, 25,110,2)
    select_raditz = selection_sprite(screen, raditz,image_path, 350, 110,1)
    select_trunks = selection_sprite(screen, trunks,image_path, 675,110,0)

    characters = [goku, raditz, trunks]
    select_sprites = [select_goku, select_raditz, select_trunks]


    return (characters, select_sprites)
```

The setup for the level screen sets the position of the character the player has selected, initialises the enemy and sets their position, initialises the health bars and returns the two health bars to be drawn on screen in update.

```
def set_level_positions(screen,player, enemy):
    player.default()
    enemy.default()

    player._setx(100)
    enemy._setx(800)

    player._sety(player.get_default_y())
    enemy._sety(enemy.get_default_y())

    player_healthbar = health_bar(player,screen,False)
    enemy_healthbar = health_bar(enemy,screen,True)
    player.set_health_bar(player_healthbar)
    enemy.set_health_bar(enemy_healthbar)

    return player_healthbar, enemy_healthbar
```

And setting up the background is slightly different from Prototype 1; it returns a second image to make the background look like it has a bit more depth. In a later prototype I would enter the level number as a parameter and then load the correct background.

```
#------------------levels-------------------------
def load_level_background():
    path = os.path.join("spritesheets", "level1", "background.png")
    background = pygame.image.load(path)

    path1 = os.path.join("spritesheets", "level1", "background2.png")
    background2 = pygame.image.load(path1)

    return background, background2
```

sprite_interactions.py:

With the functions for the character select, I separated getting the coordinates and checking which character has been selected to make it more organised.

```
#------------------character selection---------------

def select_eventcheck():
    x = y = 0
    for event in pygame.event.get():
        if event.type == QUIT:
            exit_game()

        if event.type == MOUSEBUTTONUP:
            x,y = pygame.mouse.get_pos()


    return x,y


def check_click(select,x,y):
    player = None
    option1 = select[0].check_click(x,y)
    option2 = select[1].check_click(x,y)
    option3 = select[2].check_click(x,y)

    if option1 != None or option2 != None or option3 != None:
        if option1 != None:
            player = option1
        if option2 != None:
            player = option2
        if option3 != None:
            player = option3

    return  player
```

Most of the functions/procedures from Prototype 1 are the same, like enemy_reaction and stay_on_screen. Set_move, set_queue and player_reaction are in the sprite interaction module.

update_display.py:

With the procedures for the level screen update, it is the same as the procedures from Prototype 1. The only new procedure is the update for the character select screen, where the parameters include the sprite groups made in the character select function and the list of characters so that each can be updated individually

```
#------------------character selection---------------

def character_select_update(screen, character_group,select_group, ticks, characters):
    screen.fill((255,255,255))

    gx, gy =(characters[0].get_select_x(), characters[0].get_default_y())
    rx, ry = (characters[1].get_select_x(), characters[1].get_default_y())
    tx, ty = (characters[2].get_select_x(), characters[2].get_default_y())

    characters[0].update(ticks, 180,gx,gy, "")
    characters[1].update(ticks, 180,rx,ry, "")
    characters[2].update(ticks, 180, tx, ty, "")

    select_group.draw(screen)
    character_group.draw(screen)

    pygame.display.update()
```

I have also changed the player default check section of code from Prototype 1 as when the player was set to its default y coordinate, the program wouldn't allow for the player to fall down slowly. So I used the boolean “in_air” which is used to fix the hitboxes to check if the character was in the air and if it isn’t, then the character is set to its default y coordinate. In the next prototype I need to make a get method for the booleans used in the main program.

```
if not player.get_animating() or current_move.get_name() == "default":
    if player.get_default_mode():
        #^ to make sure that the first frame isn't reset
        pass
    else:
        player.default()
        if player.in_air:
            pass
        else:
            player._sety(player.get_default_y())
```

main.py:

In main_functions.py, the level1 and character_select procedure/function were written the same as the pseudocode. The main file includes:

```
import pygame
from main_functions import *
from set_up import *
from sprite_interactions import *
from update_display import *

screen = setup_screen()
player = character_select(screen)
level1(screen,player)
exit_game()
```

Prototype 3

Fixing the player reaction function

The section of code where the combination_move_check and the single_move_check functions were called didn't work because there were times where the combination check function would return "None" as two keys were pressed but that didn't result in a combination move attack. I changed this to:

```
if check_move != None:
    actionqueue_mode = combination_move_check(current_move, check_move, player, action_queue)

if actionqueue_mode != None:
    action_queue, mode = actionqueue_mode
else:
    action_queue, mode = single_move_check(current_move,check_move, player, action_queue)

action_queue = default_check(player, action_queue)
```

Where queue_mode is set to "None" at the beginning of the function

There was also an error where the character would be stuck animating the same move over and over again, blocking any new inputs from being recognised. To stop this, I added this bit of code

after the default check:

```
    if len(action_queue) > 4:
        action_queue = []
        #^failsafe: to stop the queue from being stuck
```

The code is a lot neater now but the combination key recognition still doesn't work when the character is in default, and I think this is because of how the action object is removed from the queue once the action starts, not at the end of the action; I don't know how to fix that and I've spent too much time trying to. Having to do a single move then a combination move isn't that much of an error and it can be included as a feature of the game. At least now all the actions in the queue are executed where as before, only the time at the front of the queue would be executed.

I also needed to change the check_queue function for the enemy reaction: since all 4 slots in the enemy's queue needs to be filled for the set_queue function to work, the length of the queue will always be > 1. So, the try and except statement IF len(queue) > 1 THEN has been changed to:

```
def check_queue(action_queue):
    if len(action_queue) > 1:
        current_move = action_queue[0]

        #so that the program skips default:
        next_in = 1
        try:
            while action_queue[next_in].get_name() == "default":
                next_in += 1
        except: #list is too small
            return current_move, None
        check_move = action_queue[next_in]
```

Holding actions methods and procedures

```
f keydown_check(user_input, keydown):
  k1, k2, k3, k4 = keydown
  if user_input[K_z]:
      ke1 = True
  else:
      k1 = False

  if user_input[K_DOWN]:
      k2 = True
  else:
      k2 = False

  if user_input[K_RIGHT]:
      k3 = True
  else:
      k3 = False

  if user_input[K_LEFT]:
      k4 = True
  else:
      k4 = False

  return k1, k2, k3, k4
```

I changed the function to look at if the key was still set to True because looking at the action object at the front of the queue had the same effect as previous: the action would stop even if the key was still being pressed.

Note: I didn't notice, and fix, the spelling error until prototype 5

I also had to change two aspects of the character methods, first the section of code in the update method which changes the frame

```
def update(self,current_time, rate, x, y, mode, keydown):
    #update the animation frame number
    self.old_frame = self.frame - 1
    if current_time > self.last_time + rate:
        if self.hold and self.frame == self.hold_frame:
        #freezes onto one frame if the move requires it
            self.hold_position(keydown)
        else:
            self.frame += 1
            self.frame_number += 1
```

I added in an attribute called hold_frame since if the character was on the last frame of animation, the character would just switch to default; hold_frame is set in the initialise method as 0 then changed in the move methods where a hold_frame is required.

Secondly, I had to change the hold_position method as the hold attribute wasn't set to False causing the character to stay in that frame regardless of inputs.

```
def hold_position(self, keydown):
    if keydown:
        self.frame = self.hold_frame
    else:
        self.hold = False
```

Hold is then set to true again when there is a move where the player can hold down a key for the move to continue.

Why moving in air won't work

The set_highest_point method worked in giving a window where highest_point ← True for the player then to press a movement button then move in that direction while in the air, but in order for that to work the combination move check needs to work without buttons being repeatedly pressed.

Developing the enemy response

Changes to the enemy class:

I added a smaller detect box and close_contact_check method which uses a smaller detect box, which is slightly larger than the enemy's hitbox, to check if it has collided with the player hitbox to decide whether the enemy is close enough to attack the player.

Testing the enemy set_queue method:

01. Make the enemy move towards the player and if it is close enough, it attacks the player
    Successful: No
    Reason: The enemy did move towards the player and start attacking when it was close but the problem lies with when the player moves away: the queue freezes on [left, kick, crouch, empty] and the enemy doesn't move. Even if the player moves back into the enemy smaller detect box, the enemy stays in default

```
#set moves and repeat moves
punch,block,chop,kick,jump,crouch,right,left,down,default = moves
kick_repeat = None
punch_repeat = None

#move towards player
if not self.check_player_position(player):
    if not self.reversed:
        right.set_priority(0)
    else:
        left.set_priority(0)

#respond to the player
if self.close_contact_check(player):
    kick_repeat = kick
    kick.set_priority(0)
    kick_repeat.set_priority(1)
    crouch.set_priority(2)



#after all the moves have had their priority set
moves_ = [punch,block,chop,kick,jump,crouch,right,left,down,default]
if punch_repeat != None:
    moves_.append(punch_repeat)
if kick_repeat != None:
    moves_.append(kick_repeat)
```

Fix: The issue was with the single_move_check function; once it had checked the current_move it would check the check_move regardless if the current_move resulted in a move method being called. To fix this, I added a boolean which was set to True if the current_move had resulted in an action taken, then the check for check_move only occurred if the boolean was still False

```
def single_move_check(current_move,check_move, player, action_queue):
    move_taken = False
    if not current_move.get_movement():
        if current_move.get_name() == "punch":
            player.punch()
            move_taken = True
```

[if statements for all moves following this format]

```
if check_move != None and not move_taken:
    if not check_move.get_movement():
        if check_move.get_name() == "punch":
            player.punch()
```

02. After the player's health is below 50% the enemy moves close enough for the player to attack but the enemy doesn't attack for a set amount of time
Successful: No
Reason: In order to do this properly, I needed to use a while loop, but that would be within another while loop, causing the game to crash. I tried using a for loop, but that didn't work either: the enemy continues to attack until the player has left the detect box then doesn't move away when needed, but still moves towards the player

```
#stop attacking for a set time - partially works
    if player.get_health() <= (player.get_original_health() // 2) and not self.attack_break_done:
        stop_attacking = True
        kick_repeat.reset_priority()
        kick.reset_priority()
        crouch.reset_priority()

        if self.start_time == 0:
            self.start_time = current_time
            print("----------------starttime set-----------------")

        if self.stop_attacking:
            for time in range(stop_time):
                #move away from player
                if self.reversed:
                    right.set_priority(0)
                else:
                    left.set_priority(0)

                if current_time - self.start_time > self.stop_time:
                    self.attack_break_done = True
                    self.stop_attacking = False
                    print("---------------continue attack--------------------")
```

Fixes:
I changed it to when the enemy's health is below 25% the enemy moves away from the player and this works

```
#respond to the player
if self.close_contact_check(player):
    right.reset_priority()
    left.reset_priority()
    kick_repeat = kick
    kick.set_priority(0)
    kick_repeat.set_priority(1)
    crouch.set_priority(2)


    if self.get_health() <= (self.get_original_health() // 4):
        kick_repeat.reset_priority()
        kick.reset_priority()
        crouch.reset_priority()

        #move away from player
        if self.reversed:
            right.set_priority(0)
        else:
            left.set_priority(0)
```

03. Neaten and fix everything before moving on
    a. The enemy movement is a bit off
       I needed to add that the enemy should be set to default when the player is in the detect box

```
#to stop the enemy from attacking when the player-
#- has left the smaller detection box
if self.check_player_position(player):
    default.set_priority(0)
```

    b. When the player/enemy is hit, make sure that they move a little bit back

```
if self.being_attacked and not self.attacking:
    if self.reversed:
        self.move(1,0,x,y)
    else:
        self.move(-1,0,x,y)
```

Video of the enemy response working

Making the main menu

For the planned pseudocode to work, the mouse_x and mouse_y needs to be updated at every tick; previously, mouse_x and mouse_y were only updated only if the mouse button was released. Taking this out of the event loop causes the program to crash. Putting getting the mouse position in the while loop to run at every tick doesn't work as the position isn't updated. I fixed this by using the mousemotion event in pygame events; I didn't know that was an event.

```
for event in pygame.event.get():
    if event.type == MOUSEBUTTONUP:
        option1 = arcade_mode.check_click(x,y)
        option2 = controls.check_click(x,y)
        option3 = exit_select.check_click(x,y)
        option4 = practice_mode.check_click(x,y)

    if event.type == MOUSEMOTION:
        x,y = pygame.mouse.get_pos()
```

Changes with displaying the timer on screen

The only thing that was changed was the check-number_group function. This is now a procedure since you can't remove items from a group by using .remove() but by using .kill(), the sprite is removed from all groups, meaning that the group didn't need to be returned.

```
def check_number_group(time_remaining, digit_1, digit_2, digit_3):
    if time_remaining < 100 and digit_1.alive():
        digit_1.kill()
        digit_2.set_digit(0)
        digit_3.set_digit(1)
        digit_2.set_x_value((digit_2.get_x_value() + 7))
        digit_3.set_x_value((digit_3.get_x_value() + 7))
    elif time_remaining < 10 and digit_2.alive():
        digit_2.kill()
        digit_3.set_digit(0)
        digit_3.set_x_value((digit_3.get_x_value() + 7))
```

The remainder of the code

set_up.py:

Nothing major was changed since the last prototype in set_up, just added functions which returned tuples of images for the different ending screens

sprite_interactions.py:

The menu check click function returns a list of booleans for if that option has been selected. This is returned to the main_menu function in main_functions and is only returned to the main code if there is a True in the list, which improves efficiency as if there wasn't a True in the list, the program would waste time trying to find a True which isn't there.

```
def menu_checkclick(option1,option2,option3,option4):
# return booleans(arcade,practice,controls,exit)
    if option1 or option2 or option3 or option4:
        if option1:
            return (True,False,False,False)
        if option2:
            return (False,True,False,False)
        if option3:
            return (False,False,True,False)
        if option4:
            return (False,False,False,True)
    else:
        return (False,False,False,False)
```

I've also changed the select_event_check which now returns x, y and a list of booleans (or for character select, the character or None) for each select sprite's check click method.

```
def select_eventcheck(x,y,sprites):
    item_list = []

    for event in pygame.event.get():
        if event.type == QUIT:
            exit_game()

        if event.type == MOUSEMOTION:
            x,y = pygame.mouse.get_pos()

        if event.type == MOUSEBUTTONUP:
            for sprite in sprites:
                item = sprite.check_click(x,y)
                item_list.append(item)



    return x,y, item_list
```

Then to further check for the player selected in the character select main function, a character_select_checkclick function is called, which uses the 3 options as a parameter (where two will be equal to None and one an instance of character) and returns the character.

```
def character_select_checkclick(option1,option2,option3):
    player = None

    if option1 != None or option2 != None or option3 != None:
        if option1 != None:
            player = option1
        if option2 != None:
            player = option2
        if option3 != None:
            player = option3

    return  player
```

update_display.py:

I've added to the defeated and time out update functions where a text image is displayed on screen instead of pygame text.

main_funcitons.py:

There is the main_menu main function which is new to this prototype which works the same way the character select function works but returns a tuple of booleans to be checked in the main code. Both the main menu function and the character select function have been changed so that when the mouse is on the select sprite, it changes image.

```
def main_menu(screen):
    path = os.path.join("spritesheets", "backgrounds", "menu_background.png")
    background = pygame.image.load(path)
    select_sprites = menu_setup(screen)
    select_group = add_group(select_sprites)

    framerate = pygame.time.Clock()
    x = y = 0

    while True:
        #^return statement stops the loop
        ticks = pygame.time.get_ticks()
        framerate.tick(30)

        x,y, option_list = select_eventcheck(x,y,select_sprites)

        if len(option_list) != 0:
            op1,op2,op3,op4 = option_list
            boolean_list = menu_checkclick(op1,op2,op3,op4)
            option_selected = check_true(boolean_list)
        else:
            option_selected = False

        if option_selected:
            return boolean_list

        menu_update(screen,select_group, background, x, y)
```

And the character select function:

```
def character_select(screen):
    screen.fill((255,255,255))

    path = os.path.join("spritesheets", "backgrounds", "char_select_background.png")
    background = pygame.image.load(path)

    lists = character_select_positions(screen)
    character_group = add_group(lists[0])
    select_group = add_group(lists[1])
    framerate = pygame.time.Clock()

    player_added = False
    x = y = 0

    while not player_added:
        framerate.tick(30)
        ticks = pygame.time.get_ticks()

        x,y, player_selected = select_eventcheck(x,y,lists[1])

        if len(player_selected) != 0:
            op1,op2,op3 = player_selected

            player = character_select_checkclick(op1,op2,op3)
            if player != None:
                player_added = True

        character_select_update(screen,background,character_group,select_group , ticks,lists[0],x,y)

    return player
```

The level function was also changed so that the check timer function would set a variable called time_left and so there's 3 different branches of the level function:

```
while not defeated and time_left and play:
```

The main game code follows

```
while not time left and play:
```

The out of time update is shown for a set amount of time, then play is set to False, and False is returned as the player lost

```
while defeated and time_left and play:
```

The defeated update is shown for a set amount of time , then play is set to false and if the player won, True is returned and if the player lost, False is returned.

main.py:

I changed the code from the pseudocode planned; the main code is in a while loop so that when the arcade mode level has finished, the user will be returned to the main menu screen.

```
import pygame
from set_up import *
from main_functions import *

screen = setup_screen()

while True:
    arcade,practice,controls,exit_select = main_menu(screen)
    if arcade:
        player = character_select(screen)
        win = level1(screen,player)
        if win:
            print("player moves on to the next level")
        else:
            print("player has to restart")
    if practice:
        print("added in a a later prototype")
    if controls:
        print("added in a a later prototype")
    if exit_select:
        exit_game()
```

Video showing the main menu, character select and the 3 ending screens[6]

[6] Images for the main menu screen and character select screen:
Dragon Ball z (1989) Toei Animation, Funanimation, 26th April
Dragon Ball Super (2015) Toei Animation, Funimation, 5th July
Tekken 3 (1998) published by Namco Hometek SCEE for the PlayStation
Dragon Ball Taiketsu (2003) published by Atari for the Game Boy Advance
Dragon Ball FighterZ (2018) published by Bandai Namco for the PS4 and XboxOne

Prototype 4

Fixing the special moves

Character reaction:

I added a small change to character reaction from the pseudocode, just so that code wasn't being repeated as much or going through a section when it didn't need to:

```
def character_reaction(character, action_queue):
    mode = ""
    current_move, check_move1, check_move2 = check_queue(action_queue)
    actionqueue_mode = None

    if current_move == "":
        return "", action_queue

    actionqueue_mode = special_move_check(current_move, check_move1, check_move2, character, action_queue)
    if check_move1 != None and actionqueue_mode == None:
        actionqueue_mode = combination_move_check(current_move, check_move1, character, action_queue)

    if actionqueue_mode != None:
        action_queue, mode = actionqueue_mode
    else:
        action_queue, mode = single_move_check(current_move,check_move1, character, action_queue)

    action_queue = default_check(character, action_queue)

    if len(action_queue) > 4:
        action_queue = []
        #^failsafe: to stop the queue from being stuck on one move, freezing the game

    return mode, action_queue
```

First the function checks for a special move then, if returned None, a combination move is checked if there is a check move. And if that returns None, a single move is checked and set as the action queue and mode; if either the special move check or the combination move check returned something, then the action queue and mode will be set to the values returned.

Special move check:

Most of the time, the move list (which is a list of action objects taken from the front of the action queue) has repeated moves in the list. For this, I needed to change the check queue function.

Section of code changed:

```
#so that the program skips default:
next_in1 = 1
next_in2 = 2
try:
    while action_queue[next_in1].get_name() == "default" and action_queue[next_in1].get_name() 
        next_in1 += 1
    check_move1 = action_queue[next_in1]
except: #list is too small
    return current_move, None, None

try:
    while action_queue[next_in2].get_name() == "default" and action_queue[next_in2].get_name() 
        next_in2 += 2
    check_move2 = action_queue[next_in2]
except:
    return current_move, check_move1, None
```

I changed it so that the setting of the check move variables are set within the “try” section which lets the code error if the next action object is default or a repeated (cut off from the if statement screenshot), and that the two check moves are set in different try and except statements; this also makes sure that if there isn’t a second check move but there is one check move, then that move will be returned. In the next prototype, I should make a set check moves function so that code isn’t repeated.

The two special moves:

With getting the sprite group, I wasn’t sure on how I was going to do that - so I first tried to include the sprite group as an attribute set in the initialise of the parent class, then the subclass initialise method would add it to the group. Then in the update method, if the blast is being drawn on the screen, at the end I added self.group.draw(screen) and if the blast needed to be removed: self.kill() which is a method from the pygame sprite class where the sprite is removed from all sprite groups that it is a part of. I also added setting the character boolean of blast to false when the sprite is killed so that the character stops calling the ki blast update method. And this all worked.

Common aspects used in both the special moves would be in the initialise method for each subclass, I’ve used trial and error to find it’s x and y coordinates in relation to where the character is, and I’ve also added a image box to the parent class which allows for the hitbox to be moved without moving the sprite as the rect is now equal to the image box and not the hitbox. Also the code for finding the correct frame is basically the same as the code from the character class.

Kamehameha update method:

```
def update(self, screen):
    if self.get_character().get_frame() > 77:
    #^previous frames is getting the attack ready
        self._setx(self.character._getx() + self.character.hitbox_width)
        if not self.character.get_default_mode() and self.frame == 3:
            self.frame = 3
            if self.get_character().get_reversed():
                self.hitbox = pygame.Rect((self.x - 70), self.y, -(120 * 4), 45)
            else:
                self.hitbox = pygame.Rect(self.x, self.y, (120 * 4), 45)
        else:
            self.old_frame = self.frame - 1
            self.frame += 1
            if self.frame > self.last_frame:
                self.kill()
                self.get_character().set_blast(False)

        if self.frame != self.old_frame:
            frame_x = (self.frame % 4) * 480

            self.rect = pygame.Rect(frame_x, 0, 480,45)
            self.image = self.master_image.subsurface(self.rect)
            if self.get_character().get_reversed():
                self.image = pygame.transform.flip(self.image, True, False)
                self._setx(self._getx() - 550)
            self.hitbox = pygame.Rect(self.x, self.y, (120 * self.frame), 45)
            self.rect = self.hitbox
            self.old_frame = self.frame

        self.group.draw(screen)
```

The update waits until the character is above frame 77 since the frames previous is the character getting the blast ready, then the blast will grow in size until frame 3 of the blast is reached, and then the blast stays at frame 3 until the character goes into default mode (finished the attack)

Video of the player as Goku activating the kamehameha[7]

[7] Artwork for the special moves:
Deviant Art
Authors: spied (username)
User: saiyagami
Date viewed: 26/01/2021
Year published: 200
<Link to artwork>

The death ball update method:

```
def update(self,screen):
    self.get_character().set_attack_power(50)
    self.set_projectile()
    self.set_draw()

    if 78 <= self.get_character().get_frame() <= 83:
        self.current_charframe += 1
        #let the ball grow in size...
        if self.frame >= self.last_frame:
            self.frame = 4
            #...if it hasn't reached full size

        if self.current_charframe == (self.prev_charframe + 3):
            #^ to add a delay in frame changes
            self.frame += 1
            self.prev_charframe = self.current_charframe

    elif self.projectile:
        #ball moves with projectile motion
        self.frame = 4

        self.angle = (self.angle - 1) % 360
        dx = math.sin(math.radians(self.angle)) * 15
        dy = math.cos(math.radians(self.angle)) * 15

        if self.get_character().get_reversed():
            dx = -dx

        self.rotation_angle = (-math.degrees(math.atan2(dy,dx))) % 360

        self._setx(self._getx() + dx)
        self._sety(self._gety() + dy)
```

(with the frame changing code and group.draw at the bottom; mostly the same as the kamehameha but the values for width and height are different)

I've also added here the methods of set projectile and set draw which are boolean attributes of the death ball class and are set to true when their conditions are met: for draw, it is if the character is above a certain frame and for projectile it is if the boolean hasn't already been set to true and the character is above a different frame. The character's attack power is set to 50 while the blast is updating and then reset to the character's original attack power in the default method; this sometimes doesn't work and the attack power is left at 50 for longer than it should but this is rare and I will try to fix this in the next prototype.

The ball first grows in size as the images used get larger as the frame number increases then, once the ball is at full size, the ball should freeze on that frame - but while testing, the ball would switch between the smallest size and the largest size, but then again it looks better than having the ball freeze on one frame so I decided to keep that. When the character animates to throw the ball, the ball moves with projectile motion, using 15 as the velocity and the angle set to 90 to begin with. The method then finds the vertical and horizontal components of the velocity to change the x and y coordinates by, and then inverse tan is used to rotate the image of the ball as it moves along the screen.

At first I wanted the ball to continue moving whilst the character had gone back into default but that didn't work, so I tried to make it so that the character would freeze on the frame of throwing the ball, then reset to default and kill the blast once the blast had left the screen. But the character looks a bit awkward as trying to freeze a character on their last frame doesn't work because of the order of checks in the update method (which I will try to fix in the next prototype) the character switches between the last two frame in order to stay in the animation cycle for the special move; this does look awkward but everything else worked, so I stuck with this method.

Video of the player as Frieza activating the death ball attack

I also needed to change the set-attack_box method in the character class so that the method is called every time update is called so that when I get frieza's death ball working when the character is in default, the attack box is still added to the blast.

```
def set_attack_box(self, x, y):
    if self.blast:
        self.attack_box = self.ki_blast.get_hitbox()
    elif self.attacking:
        self.set_box_lists(x, y)
        box_list = self.attack_boxes[self.attack_box_row]
        try:
            current_box = box_list[self.frame_number]
            self.attack_box = current_box
        except: #sometimes the program is a bit behind to what is going on, so index out of range error
            pass
```

The attack boxes are also sometimes behind what is going on in the program so I added a try and except statement here and in the default method, the attack box is set to None.

Testing the effectiveness of changing the character values

Since I wasn't sure if the changes to each character's attack, defense and health values would have an effect on the game, I thought I should do two timed tests: how long it takes each character to beat level 1 (which tests the attack power value) and how long it takes for each character to get beaten by level 1, while actively trying to get hit (which will test the health and defense values).

Time taken to beat level 1:

The results which are expected from this is that Vegeta takes the shortest amount of time, then Goku and then Trunks as this is the order of highest attack power to lowest attack power. Their defense and health values might have an effect on this, but only by a really small amount, so that can be ignored. With each character, I did the same attack combinations to try to make this a fair test and each of the time values will have an uncertainty of +/-0.4s for average human reaction time for both starting the timer and stopping the timer. A full set of characters were tested each time to make sure that the time taken wouldn't improve as i got used to playing the level with that character

	Test 1	**Test 2**	**Test 3**
Goku	1:33	1:50	2:01
Vegeta	0:46	0:57	1:02
Trunks	3:01	4:10	3:07

The time values for each look accurate as it doesn’t look like there is a large difference between the character's time taken for each test, when removing clear outliers and each character stays in their own section of time . These results show the results expected given the order of attack power values. So therefore, the results show the trend that was expected and the change of the attack value does make a difference to the game.

Time taken to lose level 1:

In this test, which each character, I have tried to actively lose the level by moving towards the enemy and letting him attack the player; this will test both the defense and health values to see if they have any effect on the game. The expected result would be for Vegeta to lose the game the quickest because he has the lowest defense and health values, then Goku and then Trunks.

	Test 1	**Test 2**	**Test 3**
Goku	1:11	1:06	1:12
Vegeta	0:52	0:47	0:53
Trunks	1:19	1:24	1:18

There isn't that much of a difference between the time taken to lose the level, but there is enough of a difference to see that this does have an effect on the game, and the results are as expected given the character's health and defense values

Correcting the practice mode

The layout of the practice was mostly kept the same, but the exit button was placed where the timer would have been placed. The get queue images function worked but with displaying the queue items on the screen, the images only appeared for less than a second because of how the screen updates at every millisecond.

```
def display_queue(screen, queue, previous_list):
    image_list = get_queue_images(queue)
    image_list.reverse()
    if len(image_list) == 0:
        image_list = previous_list
    for image in image_list:
        screen.blit(image, (10, (40 +(20* image_list.index(image)))))
    return image_list
```

So I changed it so that the display queue function returns the image list, and in the main code this is set as previous list, which is passed into the display queue function at the next iteration of the while loop. So if the current list is empty, the function will display the previous list and this all worked. This function is called in the update display function, where that returns the previous list to the main procedure.

The pre-level and end-level screens

Not much was changed from the planned code for both the pre-level and end-level screens

```
def display_prelevel_text(screen, text, current_time, level_start_time):
    if ((current_time - level_start_time) // 1000)   <= 3:
        screen.blit(text[0], (430, 220))
        return False
    elif 3 < ((current_time - level_start_time) // 1000) <= 6:
        screen.blit(text[1], (435,220))
        return False
    elif 6 < ((current_time - level_start_time) // 1000) <= 9:
        screen.blit(text[2], (430,220))
        return False
    else:
        return True
```

For the pre-level screen, this is the main function which makes the screen work. I changed the time interval to 3 seconds, and for the main code for the pre-level screen:

```
level_start = False
level_start_time = pygame.time.get_ticks()
while not level_start:
    ticks = pygame.time.get_ticks()
    level_start = prelevel_update(screen,ticks,backgrounds,
```

This is placed in the level function

And for the end level screen, the only thing I needed to add was a set rect for the enemy icons method, because of how only the player icon will update, only their rect is set. Also since the enemy tree isn't complete, I added an If statement so that the end level screen wouldn't animate if the player had completed it, because there wasn't a next enemy to go to..

```
def end_level_screen(screen,player,win,current_level):
    level1,level2 = setup_enemy_icons(screen)
    current_enemy = get_current_enemy([level1,level2],current_level)
    next_enemy = get_next_enemy([level1,level2],current_level)
    setup_player_icon(player,win,current_enemy)
    icon_group = add_group([player.get_icon(), level1.get_icon(),level2.get_icon()])
    background = pygame.image.load(os.path.join("spritesheets", "backgrounds", "end_le
    framerate, stop_animating = endlevel_values()

    if next_enemy != None and win or next_enemy == None and not win:
        while not stop_animating:
            framerate.tick(30)
            ticks = pygame.time.get_ticks()

            event_check()

            screen.blit(background,(0,0))
            stop_animating = player.get_icon().update(ticks,180,next_enemy)
            icon_group.draw(screen)
            pygame.display.update()


    else:
        print("enemy tree complete")
```

Testing the level 2 enemy

Test 1:

- ➔ The enemy repeats the special move when it is meant to move closer
- ➔ Sometimes the ki blast would disappear
- ➔ It looks awkward

To fix this I then made it so that

- ➔ Frieza would freeze on the last frame of animation while the ki blast sprite is still alive, and then go to default when the ki blast has hit the player or the blast has left the screen
    - ◆ This didn't work, since when a character is on their last frame of animation, it changes to default. So I made it switch between the last two frames; this looks a bit weird but it made the ki blasts work
- ➔ And to make the enemy less awkward
    - ◆ The character only blocks if the player is near them
    - ◆ Reduced the chance of the character doing a special move, and this is only done if the player isn't close to the enemy
    - ◆ The character doesn't move away if the player is inside the smaller detection box

Test 2:

- ➔ There's a glitch where the ki blast disappears but isn't killed, so Frieza's attack power stays high
- ➔ I might need to decrease Frieza's normal attack power since it is hard to beat the level

And to fix this:

- ➔ Since Frieza's frame is now frozen, the switch in attack power can be moved to the hit method since that can stop frieza from using his special move halfway through.
- ➔ Changed the attack power of Frieza from 15 to 12, and now the level is easier to beat.

For this prototype, I think that the enemy response is ok: it still moves a bit awkwardly and is still quite challenging to beat, but it is still an improvement from the previous enemy.

The remainder of the code

These are the sections of code changed from each module from the previous prototype and not all the code for the game will be included here, as some of it wasn't changed.

set_up.py:

The functions for the level set up:

```
def level_values():
    #in order of the original level1 function
    return pygame.time.Clock(), [], set_move(), set_move_enemy(),True, F

def set_enemy(screen,level):
    enemies = [Raditz_enemy(screen), Frieza_enemy(screen)]
    for enemy in enemies:
        if enemy.get_level() == level:
            return enemy


def level_setup(screen,player,level):
    screen.fill((255,255,255))
    backgrounds = load_level_background(level)
    enemy = set_enemy(screen,level)

    enemy.set_health_bar(health_bar(enemy,screen,True))
    player.set_health_bar(health_bar(player,screen,False))

    timer_numbers = set_digits()

    return enemy,timer_numbers
```

I added a few more level setup functions, where in level_values, the tuple is separated before the while loop for the level and each mode in the game (like practice mode or the end level screen) will have a "values" function which will return a tuple of booleans, lists and other things needed for the certain game mode to start; the set enemy function is used to get the right enemy for the level by looking at an attribute called level,which is the level number that the enemy appears on and the level setup returns the enemy and the timer numbers, and not the health bars as I can use the character get health bar method.

Pre-level and game over setup:

```
def get_prelevel_images(level):
    #returns a tuple of images which are seperated in sprite interactions
    stage_path = os.path.join("spritesheets", "pre_level", "stage%d.png" %(level))
    ready_path = os.path.join("spritesheets", "pre_level", "ready.png")
    fight_path = os.path.join("spritesheets", "pre_level", "fight!.png")

    stage = pygame.image.load(stage_path)
    ready = pygame.image.load( ready_path)
    fight = pygame.image.load(fight_path)

    return (stage, ready, fight)

def get_endlevel_images():
    lose_path = os.path.join("spritesheets", "end_level", "lose.png")
    win_path = os.path.join("spritesheets", "end_level", "win.png")
    time_path = os.path.join("spritesheets", "end_level", "time.png")

    lose = pygame.image.load(lose_path)
    win = pygame.image.load(win_path)
    time = pygame.image.load(time_path)

    return (lose,win,time)
```

I've included two functions which return the images needed for the game over and pre-level screens - where in the pre-level images, the level number is taken in a parameter so that the correct stage image is used.

Practice mode setup:

```
def practicemode_game_setup():
    #in order: moves, action_queue, keydown, framerate, previous_list, exit_game, mouse_x, mouse_y
    moves = set_move()
    framerate = pygame.time.Clock()
    return moves, [],[False, False, False, False], framerate, [], False, 0 ,0

def exitbutton_setup(screen):
    exit_path = os.path.join("spritesheets", "select_sprites", "exit.png")
    exit_select = menu_select_sprite(screen, exit_path, 455, 10, 83, 38)

    group = pygame.sprite.Group()
    group.add(exit_select)

    return exit_select, group
```

For the practice mode, the setup functions are used to set up the exit button by creating an instance of the menu select sprite class, and to return the variables needed for the mode to work.

End level setup:

```
def setup_enemy_icons(screen):

    level1 = Raditz_enemy(screen)
    level2 = Frieza_enemy(screen)

    level1.get_icon().set_image()
    level2.get_icon().set_image()

    level1.get_icon()._setpos((500,300))
    level2.get_icon()._setpos((500,150))

    level1.get_icon().set_rect_enemy()
    level2.get_icon().set_rect_enemy()

    return level1,level2

def setup_player_icon(player,win, current_enemy):
    player.get_icon().set_outcome(win)
    player.get_icon().set_image()
    player.get_icon()._setpos((400, current_enemy.get_icon()._gety()))
```

The first two functions are used to set up the character icons according to whether the player won the level played and which level the enemies are on; the icons don't need to be returned as I can use the get_icon() character method.

```
def get_current_enemy(enemies, current_level):
    for enemy in enemies:
        if enemy.get_level() == current_level:
            return enemy

def get_next_enemy(enemies, current_level):
    for enemy in enemies:
        try:
            if enemy.get_level() == current_level + 1:
                return enemy
        except: #end of enemy tree
            return None

def endlevel_values():
    return pygame.time.Clock(), False
```

Then I also used functions to get the current enemy, where the y coordinate of the current enemy will be equal to the y coordinate of the player, and the next enemy, where the y coordinate will be equal to the final y coordinate the player icon reaches, if the player has won the previous level. And there is also a values function for the end level screen.

sprite_interactions.py:

For the practice mode:

```
def dummy_reaction(dummy):
    if not dummy.get_animating():
        if dummy.get_default_mode():
            pass
        else:
            dummy.default()
```

I included a dummy reaction function for the practice mode, and then it is also used in the pre-level screen since the player and the enemy aren't moving during that section

```
def check_attack_dummy(player, dummy):
    if player.get_attacking():
        dummy.check_attack(player)

    if dummy.get_health() <= 0:
        dummy.set_health(dummy.get_original_health())
```

The check attack procedure checks only if the player is attacking the dummy, as the dummy can't attack and the dummy's health resets if its health is below 0.

```
def setqueue_eventcheck(action_queue, moves, player, keydown, exit_ga
#punch, block, chop, kick, jump, crouch, right, left, down, default

    action_queue.reverse()
    action_taken = moves[9]

    for event in pygame.event.get():
        user_input = pygame.key.get_pressed()
        if event.type == QUIT:
            pygame.quit()
            sys.exit()
        if event.type == MOUSEMOTION:
            mouse_x, mouse_y = pygame.mouse.get_pos()

        if event.type == MOUSEBUTTONUP:
            exit_game = exit_select.check_click(mouse_x, mouse_y)

        #single keys
        elif user_input[K_a]:
            action_taken = moves[0]

        elif user_input[K_z]:
            action_taken = moves[1]
```

I also needed to include a function which checked both key inputs and the mouse button presses for the exit button, since two for loops for the same variable can't be done in the same while loop

Character reaction:

```
def single_move_check(current_move,check_move, character, action_queue):
    move_taken = False

    if current_move.get_name() == "default":
        return action_queue, ""

    if current_move.get_name() == "move_right" and character.get_reversed():
        action_method = getattr(character, "move_left")
        action_method()
    elif current_move.get_name() == "move_left" and character.get_reversed():
        action_method = getattr(character, "move_right")
        action_method()
    else:
        action_method = getattr(character, current_move.get_name())
        action_method()

    mode = current_move.get_mode()
    move_taken = True

    if not move_taken:
        #the check move won't be default, but the current move can be
```

I also changed the single move check to use the getattr inbuilt method, and also used checks to see if the character was reversed to make sure that the right animation was being used. The after "if not move taken", the code is repeated and the action_queue and mode

update_display.py:

All of the code has been covered previously and there are no changes to existing functions and procedures

character_extras.py:

The only extra module to change would be the action classes module, which I have changed to character_extras, and this includes the action classes, the icon class and the special move classes and subclasses.

mian_function.py:

The level function:
I've made it so that there is only one level function, instead of having a separate level 1 and level2 function, as it takes in the number of the level as a parameter to load the background and choose the enemy.

```
level_start = False
level_start_time = pygame.time.get_ticks()
while not level_start:
    ticks = pygame.time.get_ticks()
    level_start = prelevel_update(screen,ticks,backgrounds,player

framerate,action_queue,moves,counter_moves,play,defeated,time_lef

if not defeated and time_left:
    defeated,time_left,start_time = main_game(defeated,time_left,

if not time_left and play:
    while play:
        framerate.tick(30)
        ticks = pygame.time.get_ticks()

        play = time_out_update(screen, backgrounds, player, enemy
        if not play:
            return False

if defeated and time_left:
    while play:
        framerate.tick(30)
        ticks = pygame.time.get_ticks()
        play = character_defeated_update(screen, backgrounds, pla

        if not play and player.get_dead():
            return False
        elif not play and enemy.get_dead():
            return True
```

First the pre level screen is displayed and once that has finished, I call a function called main game which has the main while loop for the game and once play is False, the booleans and start_time are returned. The bit that is cut off before the main game starts, the variables for the game are returned from the get_level_values function. I'm not sure if I should split this function into even smaller functions and procedures because that would make it easier to find errors in the code but too many variables are being passed in and returned already.

Practice mode:

```
def practice_mode(screen,player):
    dummy = Goku(screen)
    set_level_positions(screen, player, dummy)
    character_group = add_group([player,dummy])
    healthbar_group = add_group([player.get_health_bar(), dummy.get_health_bar()])
    backgrounds = load_level_background(1)
    moves, action_queue, keydown, framerate, previous_list, exit_game, mouse_x, mou
    exit_button, button_group = exitbutton_setup(screen)

    while not exit_game:
        framerate.tick(30)
        ticks = pygame.time.get_ticks()

        action_queue, keydown, exit_game, mouse_x, mouse_y = setqueue_eventcheck(ac
        keypressed = check_true(keydown)

        if not player.get_animating():
            mode, action_queue = character_reaction(player, action_queue)

        dummy_reaction(dummy)
        check_attack_dummy(player, dummy)
        stay_on_screen(player,dummy)

        previous_list = practice_update(screen,backgrounds,player,(mode,keypressed,
```

The practice mode procedure works the same as the level function but has a check for if the exit button is pressed and the list of images appearing on screen are returned from the update function.

main.py:

And this is the main code:

```
import pygame
from set_up import *
from main_functions import *

screen = setup_screen()

while True:
    arcade,practice,controls,exit_select = main_menu(screen)
    if arcade:
        player = character_select(screen)
        win = level(screen,player,1)
        end_level_screen(screen,player,win,1)
        if win:
            win = level(screen,player,2)
            end_level_screen(screen,player,win,2)


    if practice:
        player = character_select(screen)
        practice_mode(screen,player)

    if controls:
        print("added in a a later prototype")

    if exit_select:
        exit_game()
```

Prototype 5

Changes made to loading and saving data

There are a few changes I made to do with loading and saving data while programming it. The first would be that data is updated and saved once the player has lost the level instead of updating the level number as the player beats each level and this is so that less lines of code are used while getting the same outcome in the end. So in the mian code it would look like this:

```
17          if win:
18              win = level(screen,player,2)
19              end_level_screen(screen,player,win,2)
20              if win:
21                  win = level(screen,player,3)
22                  end_level_screen(screen,player,win,3)
23                  if win:
24                      print("game complete")
25                      player_data.set_level_number(3)
26                      player_data.save()
27                  else:
28                      player_data.set_level_number(2)
29                      player_data.save()
30              else:
31                  player_data.set_level_number(1)
32                  player_data.save()
33          else:
34              player_data.save()
35
```

And I might try to simplify this even more in the next prototype since this code is still a bit repetitive.

To test if this had worked before adding it to the main code I copied the previous prototype and checked if saving data worked there by adding in the save data class and input prompts into the shell for player name and also outputs into the shell to show if data had been retrieved as it should have been. In the actual prototype, inputs will be called in the GUI with the name being displayed while it is being typed and there won't be anything to say whether data was loaded correctly or if a new file was created because I'm hoping that this would work correctly in the actual prototype.

Video of choosing to load data, or not to load data, when using the practice mode

Making the unlockable character select screen

There was two little changes that I had made to the character select: first would be that I needed to add a polymorphed check click method for the enemy characters since when they aren't unlocked by the player, if the select sprite is clicked, then the player shouldn't be returned, so I changed this to:

```
def check_click(self, mouse_x, mouse_y):
    if self.x < mouse_x < (self.x + self.width) and self.y < mouse_y < (self.y + self.height) and self.character_unlocked:
        return self.character
```

Where the boolean character unlocked is from the save data object select_[character name] method

The second change that I made was that, since the only change between the two character select screens was the select sprites displayed, the function called itself when the extra or back button was clicked but passed in a boolean called first screen, to which if it were True the already unlocked characters would be displayed and if it were False then it would be the unlockable character select sprites would be displayed. So when character select is first called in the main code, True is passed in, and then in the function of the character select:

```
14      if first_screen:
15          characters,select = character_select_positions(screen)
16      else:
17          characters,select = extra_characters_positions(screen,player_data)
```

And then in the section where the extra button or the back button is checked:

```
31          if len(player_selected) != 0:
32              op1,op2,op3,op4 = player_selected
33              #option 4 being to switch screens
34
35              player,switch = character_select_checkclick(op1,op2,op3,op4)
36              if switch:
37                  if characters[0].get_name() == "Goku" and not player_added:
38                      #if goku then switch to second screen
39                      player = character_select(screen, player_data, False)
40                  elif not player_added:
41                      #switch to first screen
42                      player = character_select(screen, player_data, True)
43
```

Adding the new special moves and the controls screen

There wasn't much change to this section, but I changed the type of attack Trunks does. I thought the ending bit of the "shining slash" would, most of the time, not hit the enemy (since the enemy has been pushed back due to the previous attacks) so I changed this more to be like the "burning attack" attack that Trunks has where he shoots a ki blast really fast.

[Video of Trunks and Vegeta's special attacks displayed in the practice mode]

And for the controls screen, I made it as planned but this might be a bit too undescriptive and so I might need to change this in the next prototype:

Correcting the enemy response

After correcting small errors the only thing I changed was the chances on the final attacks and I added a chance of default when he is selecting between which close attack combination to do, but when he goes into default and stays there until the booleans for the close contact check and the check player position have changed; I think this is better so that it gives a chance for the player to launch their special moves onto cell while he won't do anything about them and when his health decreases to 50, he'll start attacking harshly anyway.

Music and sound effects in the game

Most of this worked as planned but, because of how the character select is a recursive algorithm, the play_char_select_music() procedure was called outside of the character select main function (unlike the other main functions) and is called a line before the character select function is called in the main code.

Example of a play music procedure:

```
36  def play_level_music(level):
37      pygame.mixer.music.load(os.path.join("sfx", "level%d.mp3"%(level)))
38      pygame.mixer.music.play(-1)
```

Example of playing the sound effects in the select sprite classes:

```
18      def set_frame(self,mouse_x,mouse_y):
19          prev_frame = self.frame
20          if self.check_click(mouse_x,mouse_y):
21              self.frame = 1
22              if prev_frame != self.frame:
23                  self.play_music()
24          else:
25              self.frame = 0
```

Where the method play_music is like the play --- music procedures

[Video showing the music and sound effects added to the game][8]

[8] *Dragon Ball FighterZ* (2018) published by Bandai Namco for the PS4 and XboxOne
Tekken 3 (1998) published by Namco Hometek SCEE for the PlayStation

Comments on Curtis' feedback

Decreasing the amount of pixels moved:

I changed the pixels moved from 7 to 4 and for close range, this is really good, but for when the enemy is at the other side of the screen, the character moves really slowly. So I made it so that when the key is held down the character moves 7 pixels and when it is pressed once the character moves 4 pixels

```
236
237     def movement(self, mode, x, y, keydown):
238         #making sure the character moves gradually when animating
239         if mode == "right":
240             if keydown:
241                 return self.move(7,0,x,y)
242             else:
243                 return self.move(4,0, x, y)
244
245         if mode == "left":
246             if keydown:
247                 return self.move(-7,0,x,y)
248             else:
249                 return self.move(-4,0, x, y)
250
```

Where keydown is passed in from update and is the boolean used to check if the frame of animation should change or not.

Changes made to the choice of Frieza's attacks:

I have changed Frieza's attack choice so that when the player is close to fireza, it is much less likely for Frieza to move away and try to launch his death ball. And I think that this has made the level easier to beat, but that will be decided by the testers

The wrap around screen:

I added another if statement to the stay on screen procedure so that doesn't allow characters to wrap around the screen:

```
27  def stay_on_screen(player, enemy):
28      if player._getx() < 0:
29          player._setx(0)
30
31      if enemy._getx() < 0:
32          enemy._setx(0)
33
34      if player._getx() > 900:
35          player._setx(900)
36
37      if enemy._getx() > 900:
38          enemy._setx(900)
```

“Infinite jumping” glitch:

I tried this and the character only went one level above where it shouldn't be and the glitch didn’t occur for the second time trying, while the character was one level above. I know this is due to the down action being left out because an attack was used, and I don’t know how to fix this but I’ll have another look at the queue in the next prototype - also since the queue has been slightly wrong in other aspects of the game.

The remainder of the code

set_up.py:

The additions to the set_up module are the additions of the play music procedures and the set up functions for start arcade and start practice mode, which are two new main functions that allow the player to choose to load data or not; I will show this in the main functions section of the remainder of the code section. The two set up functions return the select sprites for each option.

sprite_interactions.py:

I created a check move function which takes in “next in” and the queue as parameters and returns the check move for the character reaction function to work. The code is the same as previous so I won’t repeat that here, and now the check_queue function is a lot more smaller:

```
def check_queue(action_queue):
    if len(action_queue) > 1:
        current_move = action_queue[0]

        check_move1 = get_check_move(1, action_queue)
        check_move2 = get_check_move(2,action_queue)

        return current_move, check_move1, check_move2

    elif len(action_queue) == 1:
        current_move = action_queue[0]
        check_move1 = None
        check_move2 = None

    else: #nothing in the queue
        return "", action_queue

    return current_move, check_move1, check_move2
```

This reduces repeated code and therefore makes the program slightly more efficient.

And the last change to the sprite_interactions functions to do with the levels would be corrected spelling allowing for the player to hold block and perform a low block.

Also in this module is the enter name function, which uses the pygame key functions and procedures to let the player enter their name. One problem with this is that, to make sure that letters aren't added onto the string "player_name" continuously, I've made it so that in the set name function letters can't be repeated and that might be annoying for some players.

```
def enter_name_event_check(enter, letters):
    #using pygame key and event functions
    for event in pygame.event.get():
        if event.type == QUIT:
            exit_game()

        if event.type == KEYDOWN:
            if event.key == pygame.K_RETURN:
                pygame.mixer.Channel(2).play(pygame.mixer.Sound(os.path.join("sfx", "select.wav")))
                enter = True

            elif len(pygame.key.name(event.key)) > 1:
                #check backspace
                if event.key == pygame.K_BACKSPACE:
                    try:
                        letters.pop()
                    except IndexError:
                        pass

            else:
                pygame.mixer.Channel(1).play(pygame.mixer.Sound(os.path.join("sfx", "mouse_on.wav")))
                letters.append(pygame.key.name(event.key))

    return enter,letters
```

And the set name function:

```
def set_name(letters, player_name):
    for letter in letters:
        if len(player_name) >= 1:
            if letter not in player_name:
                player_name += letter
        else:
            player_name += letter

    return player_name
```

update_display.py:

The only procedures that were added are the update procedures for the start arcade, start practice mode and the controls screen, which are procedures which are similar to previous procedures, so i'll leave that out

character.py:

The only change I haven't mentioned from this module is that I changed the highest y values to be 10 pixels higher than the default y because some characters were inconsistent with how high they went when jumping, so this makes sure that they do fall down after jumping.

character_extras.py:

Firstly, I added a set_image_enemy method in the icon class so that if the player had defeated the enemy their icon would appear greyed out

```
def set_image_enemy(self, current_level,win):
    if self.character.get_level() < current_level:
        self.frame = 1
    elif self.character.get_level() == current_level and win:
        self.frame = 1
    elif self.character.get_level() == current_level and not win:
        self.frame = 0
    else:
        self.frame = 0

    self.rect = pygame.Rect((self.width * self.frame),0,self.width,self.height)
    self.image = self.master_image.subsurface(self.rect)
```

And this method is called in the setup function for the enemy icons. The set rect method still needs to be used as in the rect, the icon's x and y positions aren't included so the rect needs to be set with its positions

And then there's the ki_blast subclasses for the galick gun and shining slash; the update method for the galick gun is very similar to the kamehameha but the shining slash was a bit more complicated as the character does a lot of different things during this attack and needs normal attack boxes on the tip of his sword:

```
def update(self, screen):
    if 83 < self.get_character().get_frame() < 88:
        #move character up
        self.get_character().movement("jump", self.get_character()._getx(), self.get_character()._gety(),False)

    elif 88 <= self.get_character().get_frame() < 90:
        #move character back down
        self.get_character().movement("down", self.get_character()._getx(), self.get_character()._gety(),False)

    elif self.get_character().get_default_mode() or self.frame > self.last_frame:
        #the blast is removed
        self.kill()
        self.get_character().set_blast(False)

    #drawing the blast onscreen
    elif self.get_character().get_frame() >= 103:


        #the blast gets updated
        if not self.get_character().get_reversed():
            self._setx(self._getx() + 20)
        else:
            self._setx(self._getx() - 20)

        if self.frame != self.last_frame:
            self.old_frame = self.frame
            self.frame += 1

        if self.frame != self.old_frame:
            self.rect = pygame.Rect((self.frame * 85), 0, 85,75)
            self.image = self.master_image.subsurface(self.rect)
            if self.get_character().get_reversed():
                self.image = pygame.transform.flip(self.image, True, False)
            self.hitbox = pygame.Rect(self.x, self.y, 85,75)
            self.rect = self.hitbox
        self.group.draw(screen)

    else:
        #set the hitbox on the tip of the sword - needs updated x and y values
        #list index = character frame number
```

```
        if not self.get_character().get_reversed():
            sword_hitboxes = [None,None,None,pygame.Rect((self.get_character()._getx() + 90),(self.ge
        else:
            sword_hitboxes = [None,None,None,pygame.Rect((self.get_character()._getx() + 10),(self.ge
        self.hitbox = sword_hitboxes[self.get_character().frame_number]
```

main_functions.py:

The main functions that I have added into this prototype are the controls screen, enter name start arcade and start practice mode
Starting with the controls screen:

```
def controls_screen(screen):
    background, back_button, menu, framerate = controls_screen_setup(screen)
    back_group = add_group([back_button])
    x = y = 0

    while not menu:
        framerate.tick(30)
        ticks = pygame.time.get_ticks()

        x,y,menu = back_button_click_check(x,y,back_button)

        menu_update(screen, back_group, background, x, y)
    pygame.mixer.Channel(1).play(pygame.mixer.Sound(os.path.join("sfx", "select.wav")))
```

There's not much here since the controls screen just needs to check if the back button is clicked and update it

Then there is the enter name function, which is called in both the start arcade and the start practice mode:

```
def enter_name(screen, background):
    text_rect, enter_name = enter_name_setup()
    enter, letters, framerate = get_enter_name_values()

    while not enter:
        framerate.tick(30)
        ticks = pygame.time.get_ticks()
        player_name = ""
        #^reset so removes letters can be removed from the string

        enter,letters = enter_name_event_check(enter,letters)
        player_name = set_name(letters, player_name)

        enter_name_update(screen,background, text_rect, enter_name,player_name)

    return  True, player_name
```

And the boolean returned is so that once the player has pressed enter to enter their name, the program can either create a new save file or load save data.

The start arcade and start practice mode functions are similar but in the start practice mode function, which select sprite is clicked is important whereas in the start arcade this isn't important. So the start arcade function:

```
def start_arcade(screen, player_data):
    background = pygame.image.load(os.path.join("spritesheets", "backgrounds", "menu_background.png"))
    load_data, new_game = start_arcade_setup(screen)
    button_group = add_group([load_data, new_game])
    select, load, new, framerate = get_load_data_values()
    x = y = 0

    while not select:
        framerate.tick(30)
        ticks = pygame.time.get_ticks()

        x,y, options = select_eventcheck(x,y, [load_data, new_game])

        if True in options:
            #the booleans aren't actually needed since the
            #-save data method will create a new player file
            pygame.mixer.Channel(2).play(pygame.mixer.Sound(os.path.join("sfx", "select.wav")))
            select, player_name = enter_name(screen, background)

        start_update(screen, background, button_group, x, y)

    player_data.set_name(player_name)
    player_data.load_data()
    player_data.set_characters_unlocked()

    return player_data
```

And the start practice mode function:

```
def start_practice(screen, player_data):
    background = pygame.image.load(os.path.join("spritesheets", "backgrounds", "menu_background.png"))
    load_data, dont_load = start_practice_setup(screen)
    button_group = add_group([load_data, dont_load])

    select, load, dont, framerate = get_load_data_values()
    x = y = 0

    while not select:
        ticks = pygame.time.get_ticks()
        framerate.tick(30)

        x,y, options = select_eventcheck(x,y,[load_data, dont_load])
        load,dont = load_check_click(options)

        if load:
            select, player_name = enter_name(screen, background)

        elif dont:
            return player_data

        start_update(screen, background, button_group, x, y)

    player_data.set_name(player_name)
    player_data.load_data()
    player_data.set_characters_unlocked()

    return player_data
```

data.py:

This module contains the save data class and the check data function, since I thought it would be better putting the function there instead of the set up module.
The first sql statement used would be in getting the player id; since this column is set to auto increment, the player is isn't stored as an attribute and therefore has to be retrieved when needed:

```
def get_player_id(self):
    SQL = "SELECT player_id FROM Players WHERE Players.name = '%s'" %(self.get_name())
    for row in self.c.execute(SQL):
        playerID = row[0]
    return playerID
```

For loading data, all of the attempts made by the player are checked, where the attributes are overwritten when the level number is different. The sql also returns the attempts data in order of increasing level number, so that the highest number is the last attempt to be checked

```
def load_data(self):
    #set the player name before this
    try:
        SQL = "SELECT character_name, level_number FROM Attempts, Players WHERE Players.name = '%s' AND Players.player_ID = Attempts.playerID ORDER BY level_number
        #order by level number so that the highest number is last
        for row in self.c.execute(SQL):
            if row[1] != self.get_level_number():
                self.set_character_name(row[0])
                self.set_level_number(row[1])
    except:
        pass
```

And the exception is kept there just so that in case there is something wrong with the table, the program doesn't end but data isn't loaded.

And the save method inserts the player into the player table if they aren't already there and adds their attempts to the attempts table:

```
def save(self):
    name = None
    #checking if the player is already in the player table
    SQL = "SELECT * FROM Players WHERE name = '%s'" %(self.get_name())
    for row in self.c.execute(SQL):
        name = row[1]
        playerID = row[0]


    if name == None:
        SQL ="INSERT INTO Players VALUES (NULL, '%s')" %(self.get_name())

        self.c.execute(SQL)
        self.conn.commit()

    #adding the attempt to the attempt table
    SQL = "INSERT INTO Attempts(attempt_id,playerID,character_name,level_number) VALUES(NULL,%d,'%s',%d)"

    self.c.execute(SQL)
    self.conn.commit()
```

main.py:

And here is the main code:

```
import pygame
from set_up import *
from main_functions import *
from data import *

screen = setup_screen()
player_data = check_data()

while True:
    arcade,practice,controls,exit_select = main_menu(screen)
    if arcade:
        player_data = start_arcade(screen, player_data)
        play_char_select_music()
        player = character_select(screen,player_data, True)
        win = level(screen,player,1)
        end_level_screen(screen,player,win,1)
        if win:
            win = level(screen,player,2)
            end_level_screen(screen,player,win,2)
            if win:
                win = level(screen,player,3)
                end_level_screen(screen,player,win,3)
                if win:
                    print("game complete")
                    player_data.set_level_number(3)
                    player_data.save()
                else:
                    player_data.set_level_number(2)
                    player_data.save()
            else:
                player_data.set_level_number(1)
                player_data.save()
        else:
            player_data.save()


    if practice:
        player_data = start_practice(screen, player_data)
        play_char_select_music()
        player = character_select(screen,player_data, True)
        practice_mode(screen,player)

    if controls:
        controls_screen(screen)

    if exit_select:
        exit_game()
```

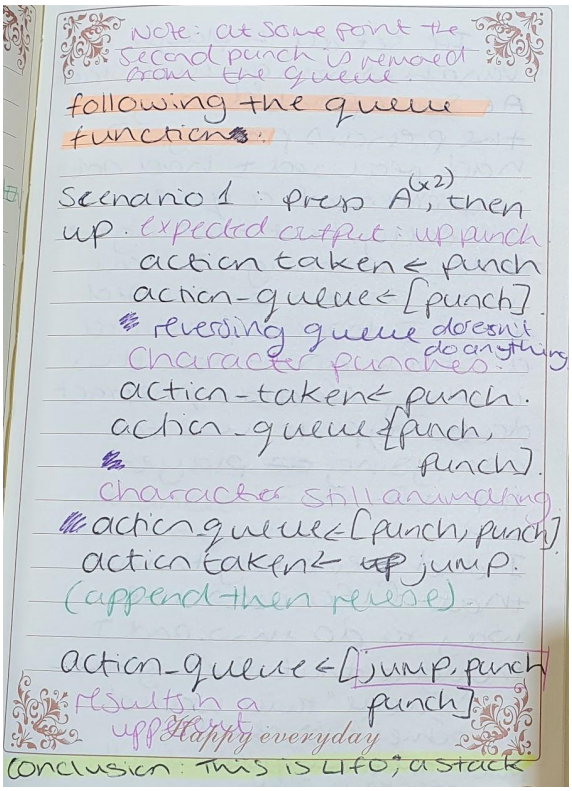
Prototype 6

Any fixes that were made

Most of the fixes made worked as planned and so they don't need explaining here; those were: moving the methods of the character subclasses, correcting Cell's enemy response algorithm and making sure the special move only happens once after being activated. But there are some aspects of the game which were only partially fixed or left semi-working.

Checking the "queue":

When I tried what I had planned out, everything didn't work so I thought I should try to follow through the set queue function more carefully and this is what I had found:



At some point during the previous prototype I had realised that I had not implemented a queue correctly, but I realised now that I have implemented a stack and that this is the preferable abstract data type to use for analysing the player's key inputs during the game.

I had previously thought that using a data structure with a first in, first out format was best for this scenario and that reversing the "queue" twice would achieve this. So I have renamed all the action_queue related variables and words to action_stack, and the functions related also.

Trying to fix two of the special moves:

Through testing, I found that the player attack box is set as the blast hitbox, as intended, but - only when the character is reversed - the collide rect function works with the blast rect. I don't know why this happened and how to fix it. But, these two special moves work fully when reversed and when reversed, at a longer distance; I think that this is okay for the game to work.

The arcade mode main function:

To make sure the correct level number was being saved, I needed to add in a few more conditions in the if statements and the return statement should be outside of the if statement so that player_data is returned to the main code. So this is the code:

```
def arcade_mode(screen, player, player_data, level_number):
    #fist called with level_number = 1
    win = level(screen, player, level_number)
    end_level_screen(screen,player,win,level_number)
    if win and level_number != 3:
        player_data = arcade_mode(screen,player,player_data,(level_number+1))
    elif not win or level_number == 3:
        if level_number != 3 or level_number == 3 and not win:
            level_number -= 1
        player_data.set_level_number(level_number)
        player_data.save()

    return player_data
```

And so the main code now looks like this:

```
import pygame
from set_up import *
from main_functions import *
from data import *

screen = setup_screen()
player_data = check_data()

while True:
    arcade,practice,controls,exit_select = main_menu(screen)
    if arcade:
        player_data = start_arcade(screen, player_data)
        play_char_select_music()
        player = character_select(screen,player_data, True)
        player_data.set_character_name(player.get_name())
        player_data = arcade_mode(screen, player, player_data, 1)

    if practice:
        player_data = start_practice(screen, player_data)
        play_char_select_music()
        player = character_select(screen,player_data, True)
        practice_mode(screen,player)

    if controls:
        controls_screen(screen)

    if exit_select:
        exit_game()
```

Fixing the set name function:

The code that I had planned did work but I found out that looping through the letters list wasn't the thing causing the letters to be repeated, it was the player name not being cleared - which I have fixed previously. So the code now looks like this:

```
def set_name(letters, player_name):
    for letter in letters:
        player_name += letter

    return player_name
```

Fixing the scrolling selection:

I used a pygame function called pygame.mouse.get_pressed() which works like the keyboard.get_pressed() function as it returns a tuple or list of booleans for the state of each button on the mouse - whether it has been pressed or not. so , since the tuple for the mouse booleans would be small, I set the tuple as a variable and selected the right boolean by using the correct index

```
def select_eventcheck(x,y,sprites):
    item_list = []

    for event in pygame.event.get():
        all_mouse_buttons = pygame.mouse.get_pressed()
        if event.type == QUIT:
            exit_game()

        if event.type == MOUSEMOTION:
            x,y = pygame.mouse.get_pos()

        if all_mouse_buttons[0] or all_mouse_buttons[2]:
            for sprite in sprites:
                item = sprite.check_click(x,y)
                item_list.append(item)

    return x,y, item_list
```

The character select screen:

I changed it so that the return player statement was placed in the if player != none if statement so that the program didn't switch back to the first character select screen before returning the player. But while testing this I found that when you press a mouse key relatively quickly, the check click functions error and I'm not sure what is causing that:

using:

```
if len(player_selected) != 0:
    try:
        op1,op2,op3,op4 = player_selected
    except:
        print(player_selected)
    #option 4 being to switch screens
```

I found that the player_selected returns 7 "None"s in a list, so I changed this to:

```
if len(player_selected) == 4:
    op1,op2,op3,op4 = player_selected
    #option 4 being to switch screens
```

And this now works.

Evaluation

Prototype 1

- ~~Display and animate characters on screen~~
- ~~Let the player be able to hit an enemy~~
- ~~Enemy should react~~
- ~~Game should stop when the enemy is defeated~~
- ~~Load a background~~

All objectives for this prototype have been met

My improvements:

- ➔ The hitboxes aren't completely accurate; this could be ignored as the player won't be able to see the hitbox when playing the game but it will make the game strategy less complicated. There should be a hitbox around the attacker's fist/leg instead of the whole body which needs to be an attribute of that character.
- ➔ The moves with combination keys do not work because someone can't press two keys at the exact same time, so I will have to research how to add a time delay between key presses.
- ➔ The background looks weird; i'll fix that in the next prototype

Client feedback:
[prototype was too small to receive feedback]

Prototype 2

- ~~Fix the combination key recognition~~
- ~~Character selection~~
- ~~Health bars~~
- ~~Fix the background~~

All objectives for this prototype have been met

My improvements:

- ➔ The combination key recognition doesn't work properly; the keys for the combination move have to be pressed while the character is animating for this to work. I need to check the current_move and check_move checks at the beginning of the player_reaction

function. Also I should organise the code in the player_reaction function into separate functions to make it easier to maintain.

➔ The character select screen looks weird and I need to fix that.

Client Feedback:

➔ Client 1: Curtis

He said that he likes the game so far, it looks cool and he thinks that it can be an interesting game. The improvements he asked for was with the player movement: moves like “crouch” and “block” should be able to be held for a longer amount of time, the player should be able to move gradually instead of in increments and the player should be able to jump then move in a direction in the air. He also pointed out that when the enemy is attacked, they should move back a few pixels to make it look realistic.

I said that I will try to include this in the next prototype.

➔ Client 2: Kanny

“As a tester of the gaming prototype, I, Kanwal Rashid, have the following statement:
Recently, after downloading Pygame and setting up my computer, I tested Prototype 2 of the game module.
The characters (sprites), although rather pixel, are very well made in their overall form.
The overall movement of the characters, though too slow for my liking, were rather smooth and swift. It would be most convenient if there were a function in which the gamer could long-press the controls for continuous movement.
While on the topic of controls, it is most pleasant to see (for a rookie gamer like myself) that the controls are very easy to remember, such as; “A” for “Punch”, “X” for “Kick”, etc. I thoroughly appreciate the simplicity within this.
One main issue that I see is the overall detail of the game. It is rather bland and pixel. I want to see more intricate and complex details. (which I anxiously wait for in the upcoming prototypes).
The game programmer had once told me that the purpose of the game is to level up as the gamer plays on. It is the detail that I appreciate and agree on because it keeps the consistency and thrill of the game vivid and lively. I have yet to expect this feature and see it for myself.
In conclusion, Prototype 2 was indeed a successful trial for me and I thoroughly appreciate the hard work the programmer has put into it. I await the upcoming prototypes and the constant development of the game.

Best regards,
Kanwal Rashid”

In response to Kanny’s feedback, I will program an easy-mode enemy AI for the first level and add more detail to the health bars (design and add an image), the top-display of the game screen and a main menu screen.

Prototype 3

- ~~fix the character select screen~~
- ~~create an easy level enemy response~~
- ~~include a main menu screen~~
- amend the player movement to Curtis' requirements
    - I did the hold a key when moving, blocking or crouching but I didn't get the jump → move to work
- ~~create the health bar / timer (top screen) display to Kanny's requirements~~

My improvements:

- ➔ I think the queue check timing can't be changed, so no jump → move or default → combination move
- ➔ The level1 function needs to be separated into smaller procedures which can be reused in later levels
- ➔ Include a pre-level screen procedure; where the game displays "stage1, ready, fight!" before the level
- ➔ The text images/sprites might be a bit off center, I should be more careful in where I place them on screen

Client feedback:

- ➔ Client 1: Curtis
  "The AI is hard to beat without abusing the fact that it just runs at you and spams kick meaning if you just spam kick it before it reaches you, it gets stuck in a cycle of getting hit without being able to reach you. Well done though it was actually a very competent game."

In response to Curtis, I will make a random low-chance (but not too low) of the level1 enemy AI to stay in default and/or block.

- ➔ Client 2: Kanny
  "As a tester of the gaming prototype, I, Kanwal Rashid, have the following statement.
  Recently (while on a Skype call with the programmer) I tested Prototype 3 of the game module. Having already downloaded Pygame for the previous prototype, the set-up was rather easy and that to me is a big relief.
  The controls on Prototype 3 remain pretty much the same, except for the addition of a "Block" function, which is a "Z". It is still evident in its simplicity and as a rookie gamer, I thoroughly appreciate this detail.
  One very exciting improvement that appears in Prototype 3 that keeps the game lively and vivid is the enemy response. In contrast to the previous prototype, the enemy

responds according to how the fighter is controlled. For the first round of the test, I lost to the enemy; which shows the excellent coding that goes into the enemy response. (Also my lack of basic gaming skills that is very apparent as a game tester).

As for the controls, the long-press function works on most actions, however, there are still some functions in which you have to repetitively press the controls; which isn't much of a bother to me, but rather to my poor keyboard. It would be appreciated (mostly by my keyboard) if the game's long-press functions work on everything.

The game's opening cover is splendid and bright; much less pixel than the previous one. There are also options/modes for the game, which the programmer has told me she would work on later.

The programmer has told me her many ideas to improve the game to maintain its excitement. I will anxiously wait for the next projects and continue to enjoy the upcoming game modules, which I am certain will be a successful triumph.

The hard work the programmer has put to her game is thoroughly appreciated and enjoyed by me and I know for certain she will continue to show her best.

Best regards,
Kanwal Rashid"

I'm not sure if adding a long press for each attack will be worth it since that will make the program inefficient since you will have to check if each key is still being held and if it is moving or if the move needs to be repeated. I will discuss this with Kanny before designing the next prototype. But, I will add the practice mode into the next prototype with a character "dummy" and keys pressed displayed on screen.

Prototype 4

- ~~Make the level 2 enemy response, which will be harder than level 1.~~
- ~~Include a practice mode, where the player can fight against a responseless character and the keys they are pressing are displayed onscreen.~~
- ~~Include a "special move" for one playable character and the level 2 enemy.~~
- ~~Pre-level screen and "player advances to the next level" screen.~~
- Organise the level functions into smaller functions and procedures.
    - I think I did as much as I could with separating the level function, but it might still be a bit too big
- ~~Change each of the character's attack power, health and defense values so each is unique, and with that re-design the check attack method so that the character's defense is included.~~

My improvements:

- ➔ After sending the prototype, I just realised that holding the block move doesn't work but moving right, left and crouch do; this is probably because of how the hold frame is the last frame of animation. I should look at this in the next prototype.
- ➔ When Frieza (the level2 enemy) switches side, his blast direction also changes direction and I don't know if that should happen; I could change it is the clients want it to be changed so that the ki blast follows the same path and doesn't change direction.
- ➔ I should have a get check move function in the check queue function so that code isn't repeated with the two check moves.
- ➔ I could also make it so that in the end-level enemy tree, the enemy's icon could change to grey if the player has defeated them, which can easily be done in the set_enemy_rect method.

Client feedback:

- ➔ Client 1: Curtis

  "Ok so listing off the good things first, the modes that have been implemented work as intended, the sprites and animations look good, tournament formatting works for the level transition and there's a decent variety of moves available along with combo's which are fairly consistent.

  The 2 main issues that I have are that due to the movement being set distances, often you will overshoot your opponent and be unable to hit them and as they don't want to get hit they won't approach you leaving you in a situation where you have to take damage to approach the enemy.

  Second is Frieza... he just ran to the right of the map far away from me and spammed projectiles towards me which are unblock-able and deal significant damage, making the second stage virtually impossible to beat.

  In terms of bugs I was able to go off the right side of the screen and come back onto the other but not the other way around, not sure if intended or not? Another thing that I found while in the practice range is that if you perform an action mid jump with just the right timing you can attack mid air and jump again meaning you can jump indefinitely and reach the top of the screen."

I did include a percentage chance for Frieza to do close range attacks but that percentage was probably too low, so in the next prototype I'll make it so that the chances for Frieza to do a close or far range attack are switched. I can also test for a smaller distance moved when the right/left key is pressed now that the hold button down functions/methods work. With the bugs, I didn't know about the "infinite jumping" so I'll look into that and I was meant to go back to make sure the wrap round the screen didn't happen but I forgot about that. So in short Curtis' requirements are going to be fixed by:

-decreasing the pixels moved when right/left is pressed

-changing the chances on Frieza's choice of attack

-looking and fixing the jump bug and the wrap-around screen; the player should stop at the -screen boundary.

- ➔ Client 2: Kanny
  "As a tester of the gaming prototype, I, Kanwal Rashid, have the following statement.
  Recently, I tested Prototype 4 of the game module. Having downloaded Pygame already, the set-up was a rather easy process.
  In contrast to the other prototypes, there are various details that have been added to add a touch of colour to the game itself.
  Once again, as a rookie gamer, the simplicity of the controls will always be a feature which I appreciate; for it makes the whole gaming process much less daunting and I can enjoy every aspect of the game without ever having to worry about pressing the wrong key.
  However, I have noticed a slight lag to the game. The actions seem to come around slower and the enemy response seems to be rather pixel. Although, I am not entirely sure if this has to do with my computer or the coding of the game (I hope it is my computer).
  During a chat with the programmer, I have been told that the process is running rather smoothly and newer prototypes will be coming very soon. I am nothing but impressed with the quality of the work so far."

Kanny's feedback sounds mostly positive which is good. With the enemy response, it did act quite awkwardly and so I hope that when I add in the fixes for Frieza stated previously, this solves the issue. The actions do have a slight delay due to the action objects being added to the queue and then analysed through various functions, and previously I thought this wasn't noticeable but I'll check if I can decrease the time taken between the key being pressed and the action taken in the last prototype that I do by looking at the action queue as a whole.

Prototype 5

- ~~Saving and loading player data so that enemy characters can be unlocked to play as once that specific level has been completed~~
- ~~The unlockable character select screen~~
- ~~Add background music and sound effects to the game~~
- ~~Level 3 enemy response~~
- ~~Special moves for Vegeta and Trunks~~
- ~~Controls screen~~
- ~~Client improvements:~~
    - ~~Decreasing the pixels moved when right/left is pressed~~
    - ~~Changing the chances on Frieza's choice of attack~~
    - ~~Looking and fixing the jump bug and the wrap-around screen; the player should stop at the screen boundary.~~

All objectives met

My improvements:

- ➔ Both Frieza and Cell are still quite awkward, so I might want to adjust their set queue method.
- ➔ The main code looks like it might be able to be simplified with a recursive function/procedure for the arcade mode.
- ➔ Some methods in the character subclasses are repeated, so i need to put these methods in the parent character class.
- ➔ The character's action queue doesn't work as intended in some instances, and also the queue is reversed, so I should look at the queue again and fix this.
- ➔ I forgot to set the character name in the save data (I noticed this after sending the prototype to the testers) so I will add this in.

Client feedback:

- ➔ Client 1: Curtis:
  "Overall the game has come together very nicely and everything functions very competently, the soundtrack is appropriate; I haven't run into any problems with it overlapping at all. There's a good collection of characters available each with unique moves that fit them well, pulling off the special moves however proved to be quite difficult and I'm not sure whether the timings are too tight or it's just slightly miss-communicated as I was able to pull them off for each character but incredibly inconsistently. The different backdrops are also very nice to break up the different stages and give them their own personality. A few bugs that I found include:

Scrolling being able to select characters and if you do it at a moderate speed it will just crash the instance.
Cell's special move sprite is very high and just looks a bit odd (not sure if intentional).
Both Cell and Frieza tend to get stuck in a corner spamming their specials which isn't so bad for Frieza as his projectile typically goes over the player anyways but Cell's makes it basically impossible to approach him given the large horizontal hitbox of the move.
Overall a great improvement over the last iterations and the makings of a great programmer.
p.s. I managed to somehow get vegeta stuck spamming gallic gun without touching any keys and it took care of Cell no problemo”

To activate a special move, you have a small time window to press the 3 keys needed and I don’t think I can change this, but Cell’s kamehameha is a bit too high and I should fix this in the next prototype. With the two enemies being stuck on their special moves, in a corner of the screen: Frieza is already a hard character so I’ll keep him the same, but for Cell I might add a x value limit which changes the chance value and therefore changes what he is most likely to do. I will also check the mouse scrolling error as this can be easily solved by looking at the specific pygame functions in the documentation. Also I don’t know how to stop the character to not repeat the special move after you press the 3 keys then repeat one of those 3 keys, but I will try clearing the queue in the special move function and check if that works.

- ➔ Client 2: Kanny:
  “As a tester of the gaming prototype, I, Kanwal Rashid, have the following statement.
  Recently, I tested Prototype 5 of the game module. Having downloaded Pygame already, the set-up was a rather easy process.
  In contrast to the other prototypes, there are various details that have been added to add a touch of colour to the game itself.
  Once again, as a rookie gamer, the simplicity of the controls will always be a feature which I appreciate; for it makes the whole gaming process much less daunting and I can enjoy every aspect of the game without ever having to worry about pressing the wrong key.
  The visible lag from the last prototype has been fixed and renovated for the better. The movements are much smoother than before and the characters do not become pixels when they move.
  There are various combination moves for each character; each unique to their own. My personal favourite would be the sword from Trunks’s combination. There are no problems whilst I was trying out the combinations.
  The game is coming together into a wonderful masterpiece and with each prototype, I anticipate even more development. Every game I play is such a joy.”

Prototype 6

- ~~Fix previous errors in code where possible~~
- ~~Make sure that the enemies are easy enough to beat and aren't awkward~~
- Any final client requirements:
    - ~~Fix the errors Curtis found~~

All objectives met so far, and I am happy with how the game works.

Client feedback:

➔ Client 1: Curtis

He said that he is happy with the game as a whole, since he really liked the previous prototype and the fixes just needed to be implemented, which he think have worked to how he would like them to.

➔ Client 2: Kanny

"As a tester of the gaming prototype, I, Kanwal Rashid, have the following statement.
Recently, I tested Prototype 6 of the game module. Having downloaded Pygame already, the set-up was a rather easy process.
After all the prototypes tested, the final form of the game has arrived and indeed, it was not a disappointment.
The entire concept of the game has retained very well from the very first prototype to the last; a fighting simulator with Dragon Ball Z characters. The levels and modes fit very well with the game and give me nostalgia; as I reminisce about my good times as a child on my DS with the Dragon Ball Z fighting game.
The controls are very simple and allow me to fully immerse myself in the game without ever having to worry about pressing the wrong button. A feature that I love is the various combination keys. Every combination for a different character unlocks a new power move; each unique in their own way. This keeps the game very exciting and lively.
The enemy response is great and I sometimes (AKA every single time) find it difficult to defeat them on the second or third level. However, this just displays my incapabilities to become a gamer.
The scenes in the background display amazing colours and are not pixel shaped. They add a nice touch to the game and are much appreciated.
The movements in the entire game do not lag and are very smooth; just like a professional game.
Overall, every aspect of the game has pleased me and every single time I tested a prototype, the creator always has hidden surprises for me to find. I have thoroughly enjoyed this experience and expect to be a tester for your future works, too."

Final Evaluation

Project objectives:

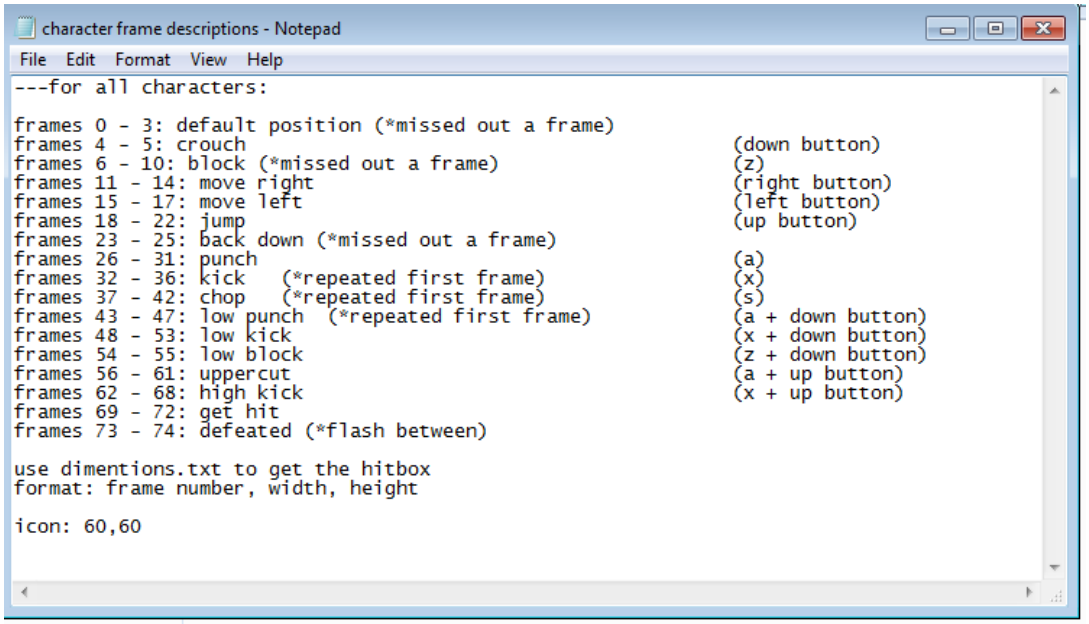
- ~~Allow the player to select a character, where each character has different move sets, strengths and weaknesses~~
- ~~Allow the player to fight against enemies of different levels of difficulty, either by selecting the difficulty or playing in an arcade-mode type style~~
    - Done by arcade mode, but selecting the difficulty can be implemented
- ~~Allow the player to play different modes like practice mode or local multiplayer mode (where the game mode is player v player using the same keyboard)~~
    - Practice mode was included, but local multiplayer wasn't
- ~~The game should be close to the style and aesthetic of a 2d-fighter game~~

I think I have done well at creating playable characters which offer a variety of advantages and disadvantages to how they operate, whether it would be their health, attack and defence values or the style of special move that they can activate. The arcade mode has been made as planned and it seems like the testers thought it was good, and the practice mode worked well also. I also think that the game gave the atmosphere of a 2d-fighting game by the music and images (backgrounds, text and characters) chosen. As proven by one of my testers who doesn't play games as much, the game is pretty easy to understand which will help with how I wanted the game to be something which can reduce stress.

There are some features that I had planned at the beginning that I haven't included in the game, like local multiplayer or selecting the difficulty of the level, but these were left out because of how I didn't have a lot of time to do everything and I can see a way of how this can be implemented in the future. I think the code can be easily maintained and added to as the functions, procedures and classes are separated into different modules, mostly smaller procedures and functions are used and I have commented the more complicated aspects of the code.

When thinking about adding extra downloadable content to the game post production, I think it would be pretty easy to add a new character, special move to a character or a new level. For the characters, I have made a text file describing the frames needed for each animation

cycle for the new character spritesheet to fit into the character class:



This is similar to the text file shown in prototype one, but now this isn't unique to each character, and each character has their own text file describing their special moves and the spritesheet and frame dimensions. Then a character subclass has to be made, following the format of the previous characters which will be easy to figure out. And I have also made my own character testing file to test and add the hitboxes and the character values, like the default y value. A special move can be added by adding another key:value pair to the special moves dictionary attribute and adding the special move method to the subclass. But a hard part of adding this would be figuring out how the special moves updates, since this is unique to each special move and character. A new enemy can be added by making a subclass inheriting from the enemy class and the character subclass of that character and a set queue method would need to be added, and adding them to the enemy function and procedures when checking which enemy is needed at each level and end level screen.