Lucky Dodgers

Jeevan Badial

AQA A-level Computer Science

NON-EXAM ASSESSMENT

Contents

Analysis

The project is a cross-platform online multiplayer dodgeball game. The end user of the product is anyone who enjoys playing online video games who owns a Windows or Linux computer or an Android phone. The game will have a retro graphic style so will attract older gamers due to nostalgia.

Investigation

Current Systems

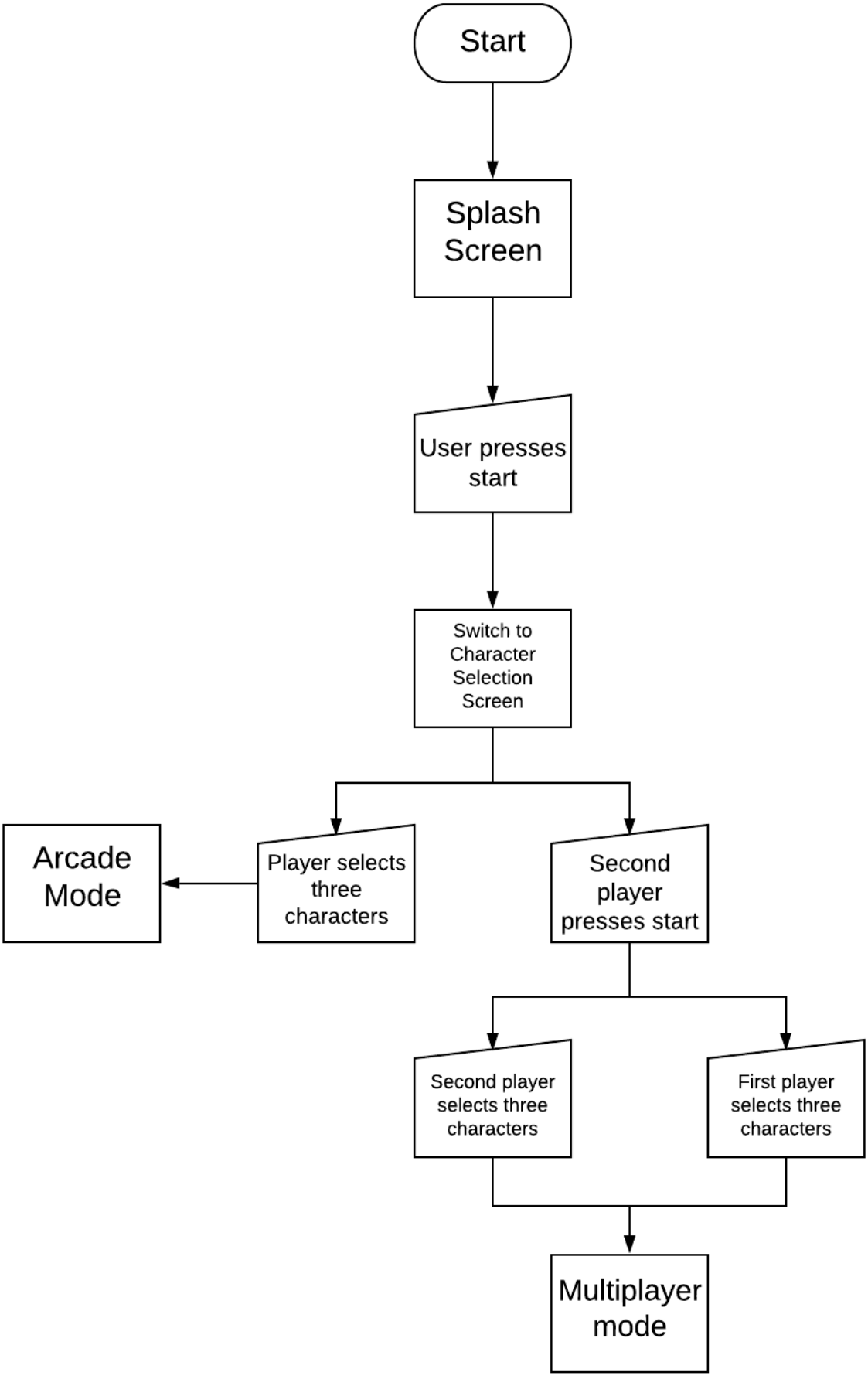
Super Dodge Ball (Neo Geo)

Super Dodge Ball came out in 1996 on the Neo Geo console. This is the game that Lucky Dodgers is inspired by.

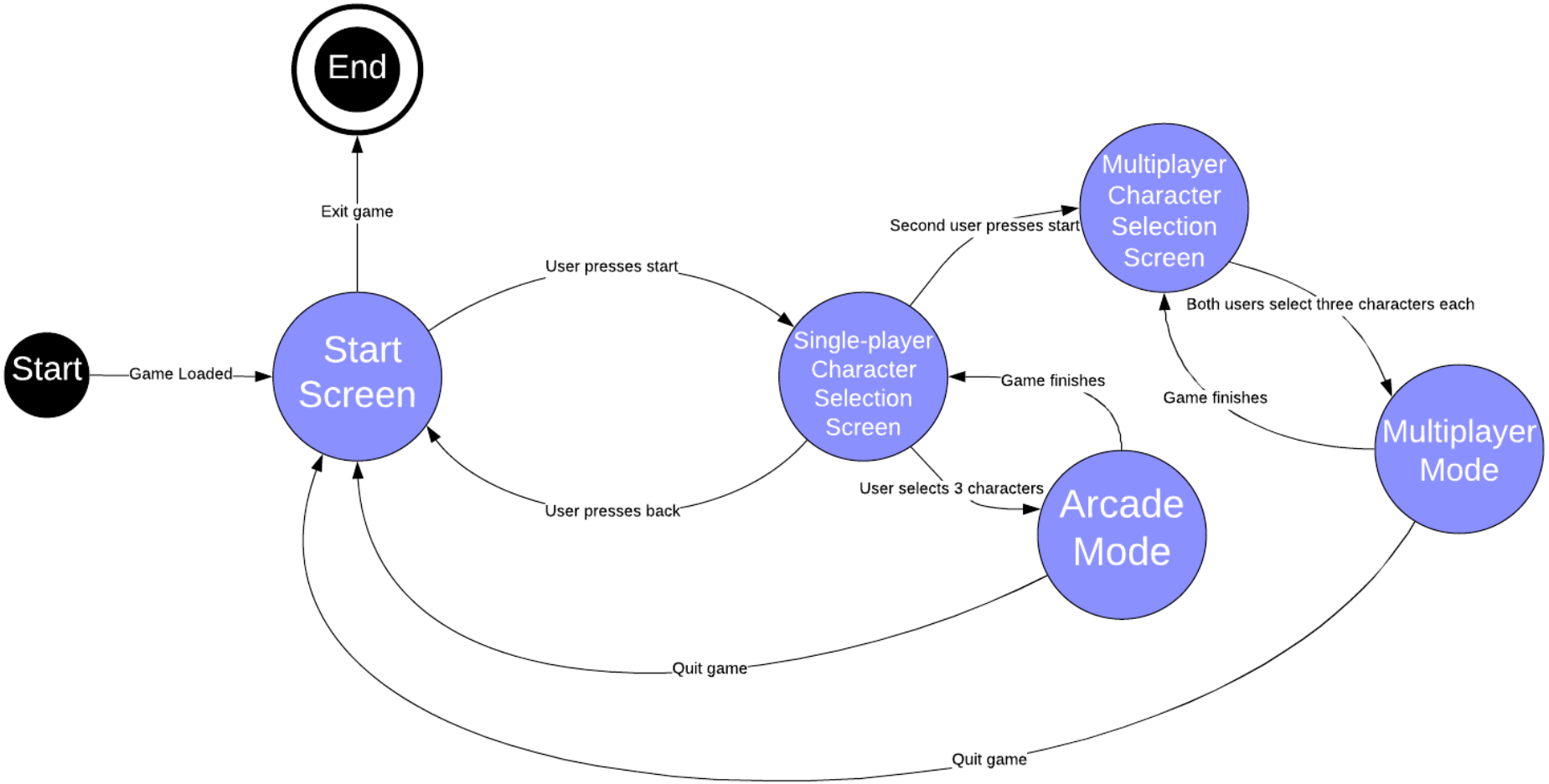
User Interface

- The game goes straight into the character selection screen from the splash screen. This was typical for older games but isn't very common nowadays with game having more features.
- On the character selection screen, the game is in arcade mode by default and changes to local multiplayer if a second player presses start on their controller.
- The players then pick three characters for their teams and the game starts.
- This is a very simple system but would not work for a more complex game.
- Visually, the interface is very garish and difficult to understand.

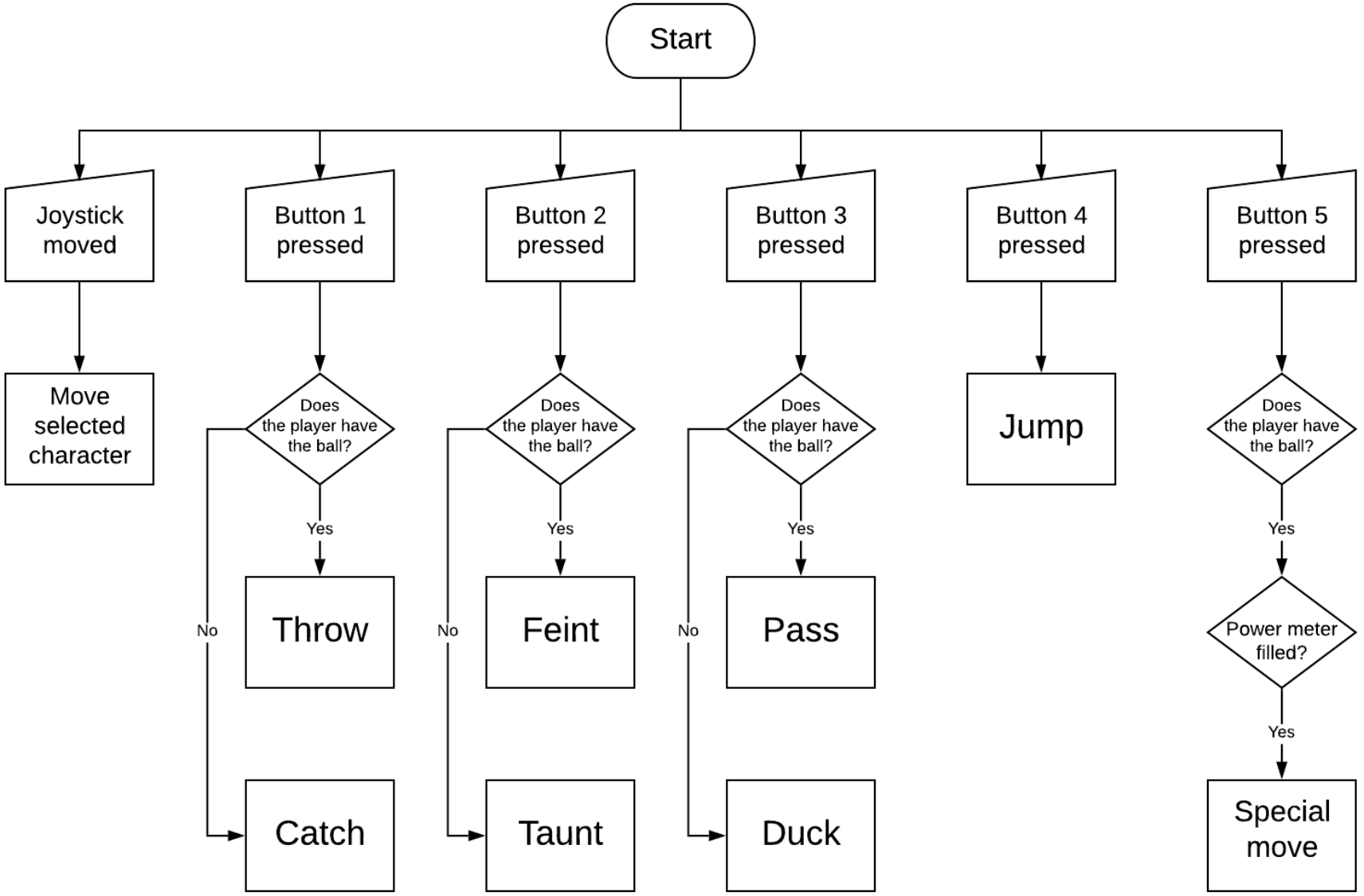
Interface Flowchart



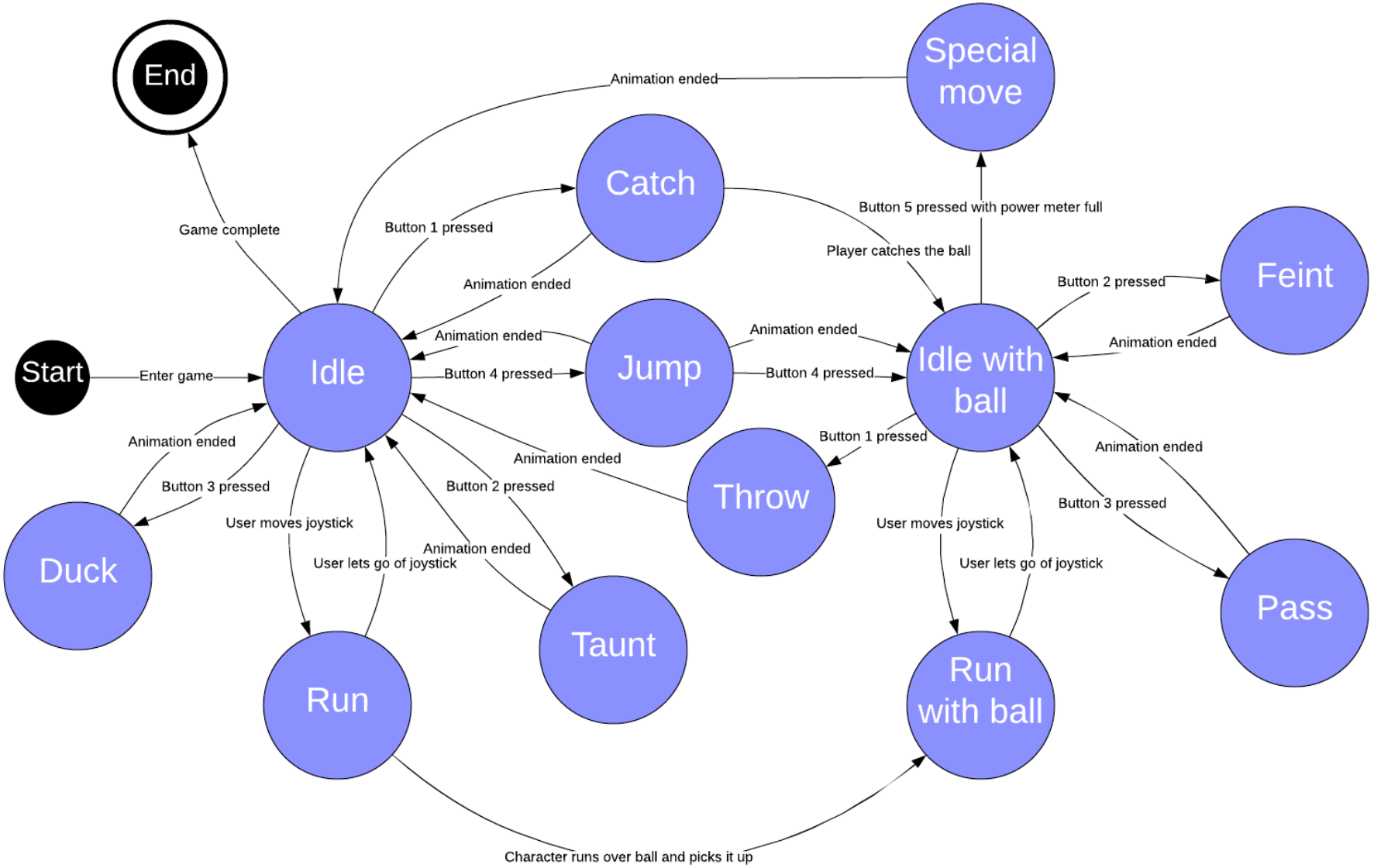
Interface State Diagram



Gameplay Flowchart



Gameplay State Diagram



Features

- The gameplay consists of three characters on each team. One character is selected at a time.
    - This is a good amount of characters because any more would be overcrowded and any fewer would be too few.
    - There are 7 different characters to choose from.
- When the controlled character has the ball, they can:
    - Throw the ball at the enemy.
    - Jump
    - Feint (pretend to throw the ball).
    - Pass to a teammate.
    - Special move
- When the enemy team has the ball, the controlled character can:
    - Catch
    - Jump
    - Duck
    - Taunt
- There is a single-player mode where the player plays against the computer.
    - This will definitely be in Lucky Dodgers so that users can play the game offline as well.
- There is a local multiplayer mode where two players on the same console can play against each other.
    - This isn't viable for phones so this would be a low priority feature to add unless the responses in the survey are heavily in favour of having local multiplayer.
- It has 2D graphics but 3 dimensions of movement.
    - I will replicate this style for Lucky Dodgers since it is easier to make 2D graphics and it gives a retro feel to the game.
- Players have a special meter which can be used to do special shots.

Flaws

- It being on the Neo Geo makes it hard to play now. Very few people have a Neo Geo and emulation is too much of a hassle for most people.
- There is no cooldown on the catch ability. This means that players can just stand still and repeatedly press the catch button and never get hit by the ball.
- When a character runs out of health, rather than being out, they move to the edge of the opponent court and can still play. Although this is a cool concept, these players are actually in an advantageous position so it's sometimes better to actually let one of your characters lose their health.
- The AI teammates do nothing except hide behind the controlled character.
    - This makes the game feel less interesting and may as well be a 1v1 game rather than 3v3.
- There is only one type of throw, so the throws are quite predictable.
    - This coupled with the catch being too easy makes the game more repetitive and boring.
- Throws are always targeted at the opponent's selected character. This gives the player less freedom and choice and therefore makes the game less tactical.
- Several actions are not very useful:
    - Jump with the ball doesn't really do anything to help.
    - Jump and duck for dodging are inferior to catch because catch is too strong.
    - Taunt, although amusing, is very long and not cancellable so you are practically guaranteed to get hit.
    - The feint is too slow to make a difference.
- In the game, the camera moves around quite a lot, making the game harder to follow. Additionally, the frame rate is quite low, so the camera movement is not smooth. Furthermore, the camera is quite zoomed in, so the screen looks quite crowded sometimes.
    - I will have a static camera instead which is fairly zoomed out.
- The special shots make the game more about filling your meter quickly than playing normally.
- Maximum of two players.
    - This can be annoying when there are more than two people who want to play together.

Dodgeball Arena

Dodgeball Arena is a dodgeball game on Android and iOS. It has over 100,000 downloads. It is free but has adverts and in-app purchases.

User Interface

- The main menu has 9 buttons:
    - Play – this has 3 options:
        - Custom
        - Tournament
        - Runner
    - Stats – this shows:
        - Level
        - Available stars (in-game currency)
        - Available matches
        - Played matches
        - Connected hits
        - Wins
    - Exit – quits the game.
    - Info – shows information about the developers.
    - Share – allow the user to send a link for the game to a friend.
    - Settings – has the options:
        - Language
        - Move type
        - Music
        - Sound
        - Tips
    - Watch a video advert – This is not cancellable so if you accidentally click on the button, you have to watch an advert for 30 seconds.
    - Leaderboards
    - Store – Allows the user to spend money to remove ads and buy in-game currency.
- The user interface is quite good. It is fairly clear what most of the buttons do and the style is quite nice with the buttons being dodgeballs.
- The exit button is pointless though because phones have a home button anyway.
- Sometimes, the buttons are unresponsive which is not good for an interface.

Features

- 3D gameplay.
    - In my opinion this is not a great idea for a mobile game since it is controlled by on-screen controls with only one joystick.
    - Additionally, it means that the graphics look worse because they are compared to other 3D games which have big budgets and very good graphics.
- On-screen controls:
    - A joystick for moving the character around.
    - Touch the screen to throw where you touched.
    - Sometimes a catch button appears. But it just auto-catches the ball while active rather than requiring timing.
- Many balls.
- Power-ups.
- 2 game modes:
    - Normal dodgeball
    - Runner – you have to reach as far as you can without getting hit.

Runner mode

Flaws

- No multiplayer means that you can't play the game with friends which is a major draw of the sports game genre.
- The game forced me to take the tutorial before every single match.
- The language sometimes switched to Spanish.
- The stats screen shows completely wrong stats.
- You only get a few matches for free. The game said it was 20 but that number decreased to 7 after 2 matches and ran out after 3 matches. When you run out of matches, you have to spend in-game currency (which you get by paying real money) or by watching a 30 seconds long, unskippable video advert.
- Aiming the ball is too hard and the balls are tiny, so it is basically impossible to get hit if you just run around.
- The game is riddled with unskippable 30 seconds long popup adverts.
- Full of microtransactions and in-app purchases.
- There are often major glitches e.g. the bots started running into the border so the balls couldn't be thrown at them. Also, balls sometimes disappear randomly.
- The controls are very confusing.
- You can only throw the ball and run around. The catch button appears rarely and then disappears.
- The user interface is not very smooth and looks quite dated.
- The AI is not very smart and often throws the ball badly.
- Having lots of balls makes the game more confusing.
- The power-ups give a frustrating advantage to whoever has them.

Client

My client for this project is a friend who enjoys playing video games. I've also played Super Dodge Ball with him, so he has some experience in dodgeball games.

Interview Transcript

Me: "Would you play a new game similar to Super Dodge Ball?"

Client: "Yeah, it was really fun."

Me: "How come you wouldn't just play Super Dodge Ball then?"

Client: "It's just too inconvenient to play when it's on a different machine. It also had some gameplay issues and is kinda outdated."

Me: "Would you prefer it to be on your phone or your computer?"

Client: "I play on both a decent amount so I would play it on either really."

Me: "Would character creation be the kind of feature you would want in the game?"

Client: "Definitely. It's always more fun to play with a character I've made myself rather than just a generic character."

Me: "For the character creation, would you prefer a system with sliders which change specific features like nose size, or would you want to be able to draw the character's face yourself."

Client: "I've never played a game which lets you draw your own characters but that sounds interesting because you can make more detailed characters."

Me: "Would you like there to be online multiplayer?"

Client: "That would make the game a lot more fun. I don't really like playing games solo."

Me: "Should there be a friends system so you can easily match up with your friends?"

Client: "Yeah, it would be annoying if there wasn't. It also would work well with the character creation because I can make funny characters to show my friends."

Me: "If there was local multiplayer like split-screen on Bluetooth, would you use it?"

Client: "Probably not. Especially if there is already online multiplayer."

Me: "Would you like a large single-player game mode other than just a quick match?"

Client: "I don't think I would play it much."

Me: "Are there any other features in particular that you would like?"

Client: "Nah."

Survey

I also created a short survey and sent it to friends and family. There were 30 responses. Here are the results:

1. What device do you play games on?

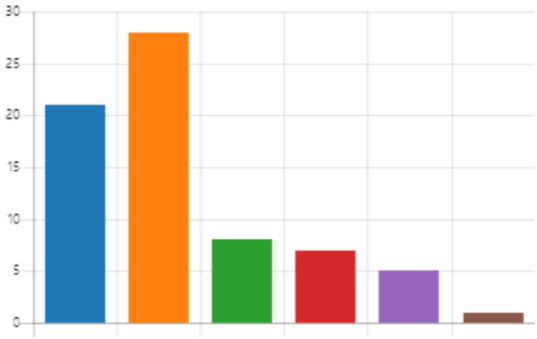
More Details

Mobile Phone	27
PC	20

2. Which of the following features would you want in a dodgeball game?

More Details

Character Creation	21
Online Multiplayer	28
Local Multiplayer (split-screen,...	8
A Campaign	7
Different difficulty settings for...	5
Other	1



3. If you picked character creation as a feature you would like, would you prefer being able to draw the faces of the characters yourself on a canvas in-game or having sliders changing individual features e.g. eyes size.

More Details

Drawing faces	16
Sliders	5

4. If you picked online multiplayer as a feature you would like, would you want a system that allows you to add players as friends and specifically match up with them?

More Details

Yes	24
No	4

5. If you would play on mobile, would you want to be able to change the positions, sizes and colours of the buttons and joystick on the screen.

More Details

Yes	22
No	5

Summary

Of the 30 responses to the survey, 17 said they played games on both their mobile phones and their computers. Additionally, the client said the same. However, 10 people said they exclusively used mobile. This data suggests that the game should be cross-platform in order to be accessible for the largest amount of people.

The majority of people responding wanted online multiplayer (28/30) and character creation (21/30) while few people wanted any other features. The client gave similar feedback. Due to this, I will prioritise these online multiplayer and character creation.

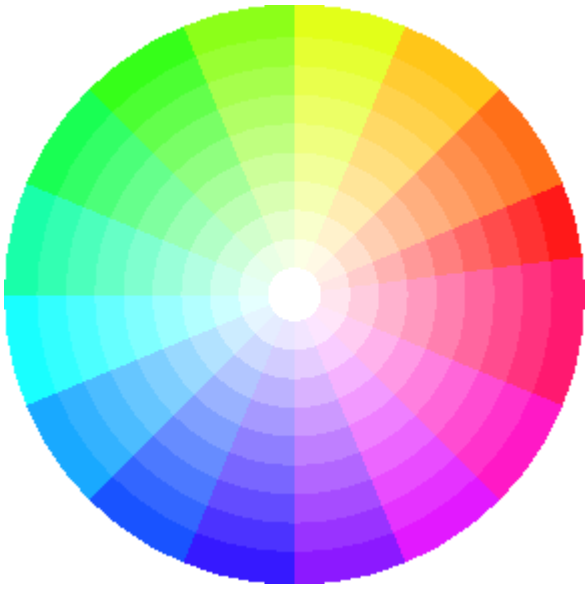
Both the client's feedback and the survey results suggest that I should make the character creation work such that users can draw their own characters and have a system in the online multiplayer to add and play with friends.

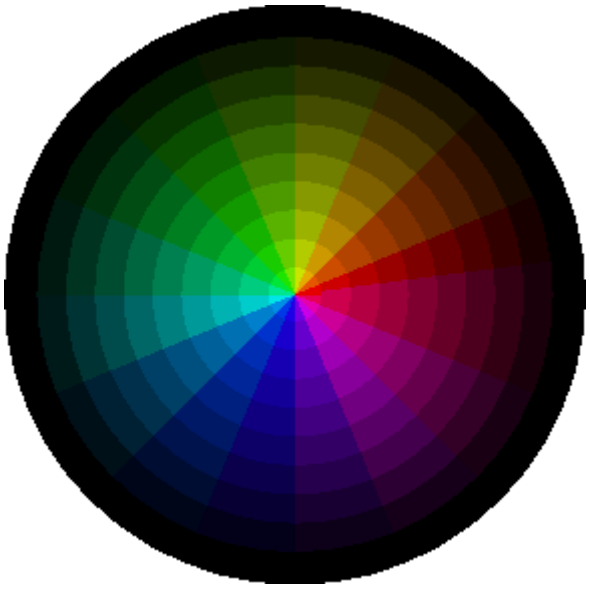
The survey also had the majority of mobile users wanting a way to change their controls. I will try to implement this as well.

Features

Character Creation

- The user gets a 30x30 canvas to draw a new head. They will get the following tools:
    - Colour wheel with almost 300 different colours. The user can drag the colour wheel inwards to get darker shades. It should look like this:





    - A colour palette of all the previously used colours in the image for easier reusing of colours.
    - Pencil with 3 different sizes (1x1, 2x2 and 3x3).
    - Eraser with 3 different sizes.
    - Fill.
    - Undo and redo up to 99 actions.
- The user can preview the new character with their new head.
    - Within the preview the user can change the colour of the character's skin, top, bottoms and shoes.
- A gallery of saved characters.
    - The user can load a character to reuse and can delete characters they don't want anymore.
- Users can play with their created characters in-game.

Gameplay

- A joystick and three buttons
    - The joystick controls the movement of the selected character.
    - When the user has the ball, the buttons do the following:
        - Light throw – a throw with a quick start-up animation which deals less damage. The ball moves slower but goes higher, so it is harder to jump over.
        - Heavy throw – a throw with a slow start-up animation which deals more damage. The ball moves quickly but stays close to the ground so it can be easily jumped over.
        - Pass – allows the user to pass to teammates.
        - Additionally, pressing one of the throw buttons while in the middle of a throw starts a new throw, acting as a feint e.g. if the play is in the middle of a heavy throw but then presses light throw, the player does a light throw instead.
            - Having two different throws each with positives and negative plus having a feint adds a mix-up to defending so you can't just catch everything. This fixes one of the major issues of Super Dodge Ball.
        - The throws will have a targeting system where the throw will be automatically aimed at the opponent who the player is aiming towards with their joystick.
            - This is a good middle point between Super Dodge Ball and Dodgeball Arena since the targeting still gives the player the freedom to aim at any character they want to but prevents the throws from being too hard to aim.
    - When the user doesn't have the ball, the buttons do the following:
        - Jump – a quick upwards dodge which goes over low throws. If the player was running when they pressed jump, the character does a running jump.
        - Catch – when timed correctly, catches the ball thrown at the character, preventing any damage done and allowing the player to throw the ball back at the enemy.
        - Switch characters – allows the user to switch to controlling a different character.
    - These will be customisable in the settings.
        - The user can change the colours, sizes and opacities of each button and the joystick.
- 3 players on each team.

    - The controlled character will have an arrow above them.
    - A health-bar will be visible above each character's head.
- One ball which bounces off of the boundaries to keep the ball in play at all times, reducing wasted time.
    - The ball will explode after a while if not thrown to prevent time-wasting.
- 4 different pitches to play in with different surfaces so the ball bounces more or less on each surface. The pitches will be:
    - Mud – low friction.
    - Grass – medium friction.
    - Snow – high friction.
    - Night – low friction.
- I will implement variable frame rate to ensure that game speed stays constant.
    - This will also make the game more playable on weaker devices and make the game look better on strong devices.

Multiplayer

- Host lobbies for people to join.
    - While in the lobby screen, users can pick their team.
    - When someone wants to join, a popup appears on the host's screen asking to join.
    - When all the players have chosen a team, the host can start the game.
- Join people's lobbies.
    - The join screen will show the currently available lobbies to join with the host's username and the number of people in the lobby.
- Add friends by entering their username.
    - Users can see what your friends are doing (offline, online, in a lobby or in a game).
    - Users can request to join a friend's lobby directly from the friends screen and friends' lobbies appear first in the join screen.
    - Users can see their friend's custom characters when playing with them.
- Up to 6 players can play in a lobby.
    - Each player would then be controlling one character. If there are fewer players, characters are allocated between them.
    - Collision detection will be done client-side to prevent unexpected hits.
    - Due to the variable frame rate, users will be playing at different frame rates. There will have to be syncing when updating characters to prevent users on low frame rates from being slowed down by users on higher frame rates.

Limitations

- There may be some lag in the online multiplayer due to there not being several dedicated servers around the world like in major online games. This will be more noticeable in countries far away from the UK. This is not too big of a problem because it is inescapable and happens in all online games to some extent. The biggest factor will be how low the user's ping is. If the user's ping is quite high, they would probably be used to having lag in online games so it would not be much of a surprise for them.
- The game may not run well on older phones. It should be playable on the vast majority of phones, but a really old phone may not be able to handle the game. However, a phone that couldn't handle the game would struggle in most basic apps as well. The game should be able to run on basically any laptop or PC unless it was about a decade old.
- The game may not be compatible with older versions of android. I wouldn't be able to test this because the oldest phone I have access to for testing is on Android 8 (current version is Android 10) so I wouldn't know if the game is compatible on Android 7 and earlier. However, very few people still have Android 7 or lower, so this is not a major issue.
- Saving the characters may be quite slow because there are a lot of images to change. This will be more apparent for making characters where the tops, bottoms, shoes and skin are all changed colour because these have to be changed pixel-by-pixel whereas the head can be replaced in one block. The saving will be slower on weaker devices.
- Drawing on a canvas on a small screen may be quite difficult for users. This is somewhat counteracted by the undo and redo tools as well has the erase. Additionally, the user will be able to drag their finger to draw many pixels in one go.
- The graphics will be quite simplistic. This is because I am not very good at drawing things. This isn't much of an issue though because the game has a retro style so the sprites would be simple anyway. Additionally, the users will often be creating their own characters, so it doesn't really matter how good the in-built characters are.

Objectives

1. The game will be playable on Windows, Linux and Android.
2. The user interface should be intuitive and responsive, with smooth animations. Changing between screens should take no longer than a second.
3. Users can create custom characters by drawing new faces and changing the colours of the character's skin and clothes. The canvas to draw on should be responsive and easy to use.
4. Users will be able to delete and load custom characters they have made.
5. Users can preview and then save their characters. Saving should take under 10 seconds.
6. The image hashing should be fast with very few collisions.
7. The game will allow users to choose from their custom characters or the built-in characters to play with.
8. The game should work with a variable frame rate.
9. The game should be well optimised. This will be achieved if the game runs at over 30 fps at all times and be fairly consistently be at around 60 fps on my phone (3 years old Google Pixel XL).
10. The gameplay should be fair and enjoyable. The success of this objective will be based on client feedback.
11. The game should be without any major bugs or glitches.
12. The AI should successfully simulate a real player and should not be too easy or difficult to beat. This objective's success will also be determined by client feedback.
13. Users can connect to the server and create a username.
14. Users should be able to add friends and see what their friends are doing.
15. Users will be able to join their friends' lobbies directly from looking at their friends' statuses.
16. Users can see a list of all available lobbies along with the host's username and the number of players in the lobby and can request to join them.
17. Users can host lobbies.
18. Users can accept or reject people from joining their lobbies.
19. Users can pick their teams while in the lobby.
20. Users will be able to see their friends' characters when playing with or against them.
21. Users can play online matches with up to 5 other players.
22. Online gameplay should run about as well as offline, reaching similar frame rates.
23. The server should be able to handle disconnects and respond correctly.
24. Users should be able to quickly reconnect if a disconnect occurs for a short period of time.

Choices of Software

- I have researched different frameworks and game engines to create the game. I decided to use Kivy[1] for many reasons:
    - It is GPU accelerated and compiles to C, so it is fast while also allowing for the flexibility of being programmed in python.
    - Kivy is open source so I can read the source code to understand how to most efficiently use the functions and classes.
    - I have experience with Kivy and understand how it works.
    - Despite being in python, Kivy still provides access to android functions such as notifications.
    - The documentation is thorough and easy to understand.
    - Kivy is quite small and not bloated while still having a large amount of functionality.
    - It works directly in python on Windows, so it doesn't have to be compiled to test.
    - It can be compiled to Android so the game can be a mobile app.
- I will be using the python library Twisted[2] to implement the network because:
    - It is easy to use.
    - It is very effective.
    - It has clear and detailed documentation.
    - Kivy has built-in integration with Twisted. This is required because both Kivy and Twisted are event-based. This means that they both need to have event loops running which wouldn't work without built-in integration.
- I will be using OpenCV[3] and NumPy[4] for image editing for the character creation. This is because:
    - OpenCV is useful for converting images to NumPy arrays and vice versa.
    - OpenCV also has functions for resizing and other useful tools.
    - NumPy arrays are very good at representing images and are well optimised with many useful functions to modify the arrays. This makes it easier to change the images than if the images were stored as python lists.

---

[1] *Kivy: Cross-platform Python Framework for NUI*. (2019). Retrieved from https://kivy.org

[2] *Twisted*. (2019). Retrieved from https://twistedmatrix.com

[3] *OpenCV*. (2019). Retrieved from https://opencv.org/

[4] NumPy. (2019). Retrieved from https://numpy.org/

- I will use PyCharm[5] as the IDE. This is because it has:
    - Good code completion.
    - Type hinting.
    - Built-in debugger and run-time system.
    - Code coverage.
    - Package management.
    - Refactoring.
- To compile the python code to an APK (Android package), I will use Buildozer[6]. Buildozer is basically an improved version of python-for-android because it automatically organises and handles prerequisites in an easier way.

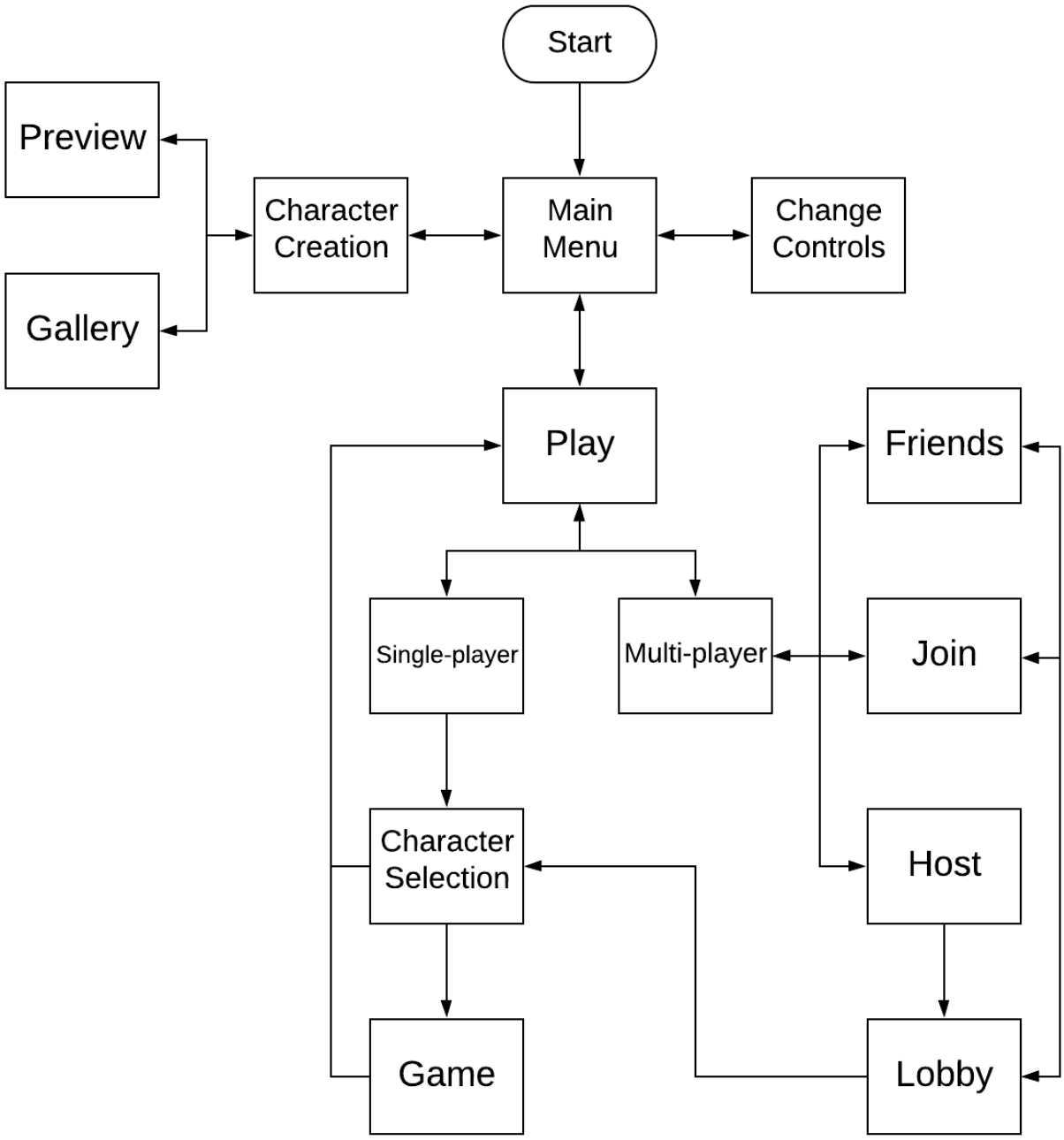
[5] PyCharm: the Python IDE for Professional Developers by JetBrains. (2019). Retrieved from https://www.jetbrains.com/pycharm/

[6] Buildozer. (2019) Retrieved from https://buildozer.readthedocs.io/

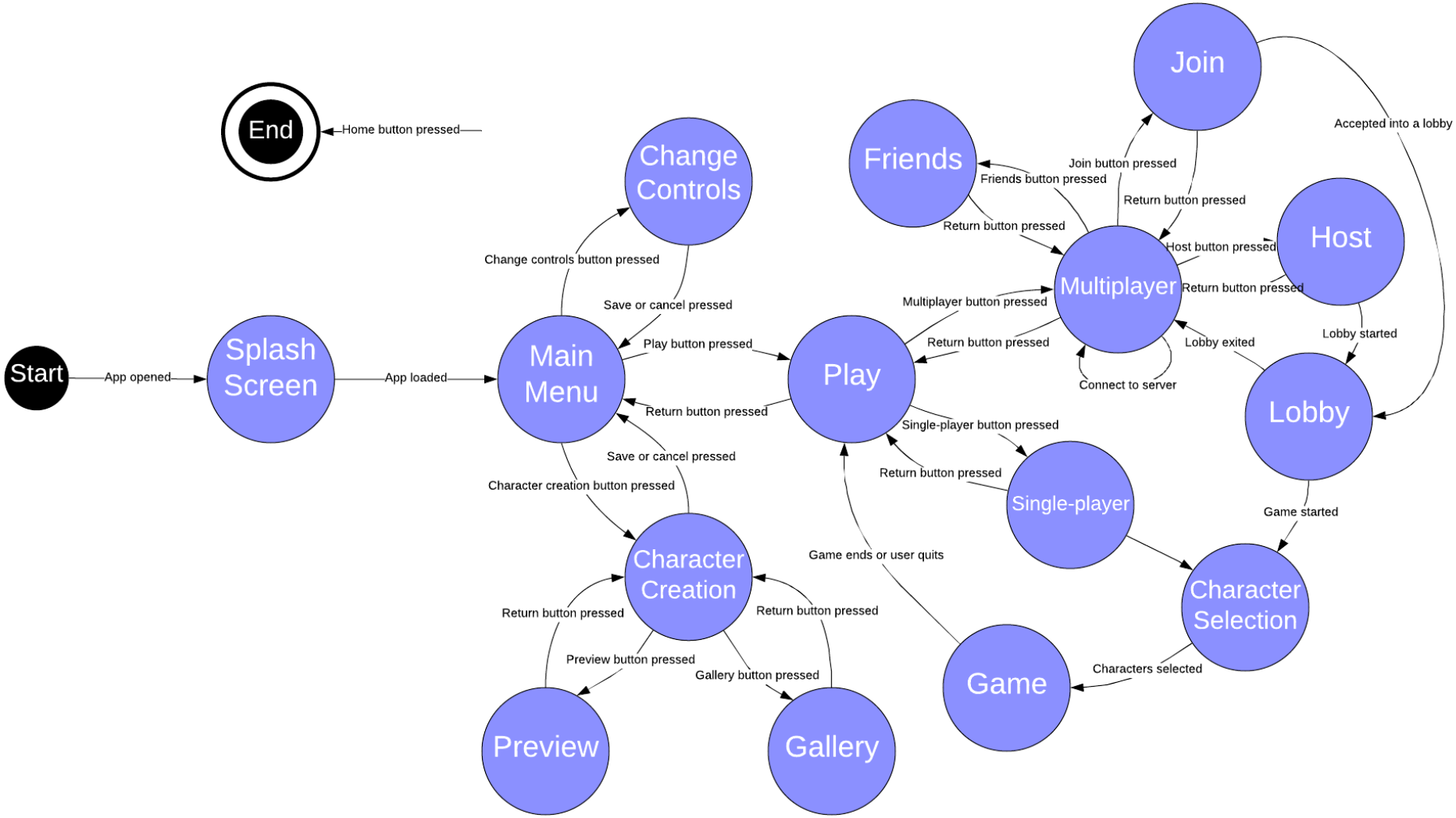
Documented Design

User Interface

User Interface Flowchart



User Interface State Diagram



User Interface features

- The transition animation between the screens will be a slide. This is a smooth and simple animation.
- The direction of the slide will differ depending on the screens e.g.
    - Left for Main Menu → Character creation
    - Right for Character Creation → Main Menu
    - Up for Main Menu → Play
- The background of the user interface will just be some lines:

    - The background will be slightly tinted with a random colour whenever the user loads the game.
    - A simple background like this is good because it isn't distracting and doesn't clutter the user interface.
- The buttons will be either symbols or text, depending on how obvious the function of the button is e.g.
    - Settings will be a cog.
    - Preview will be a button with the word preview.

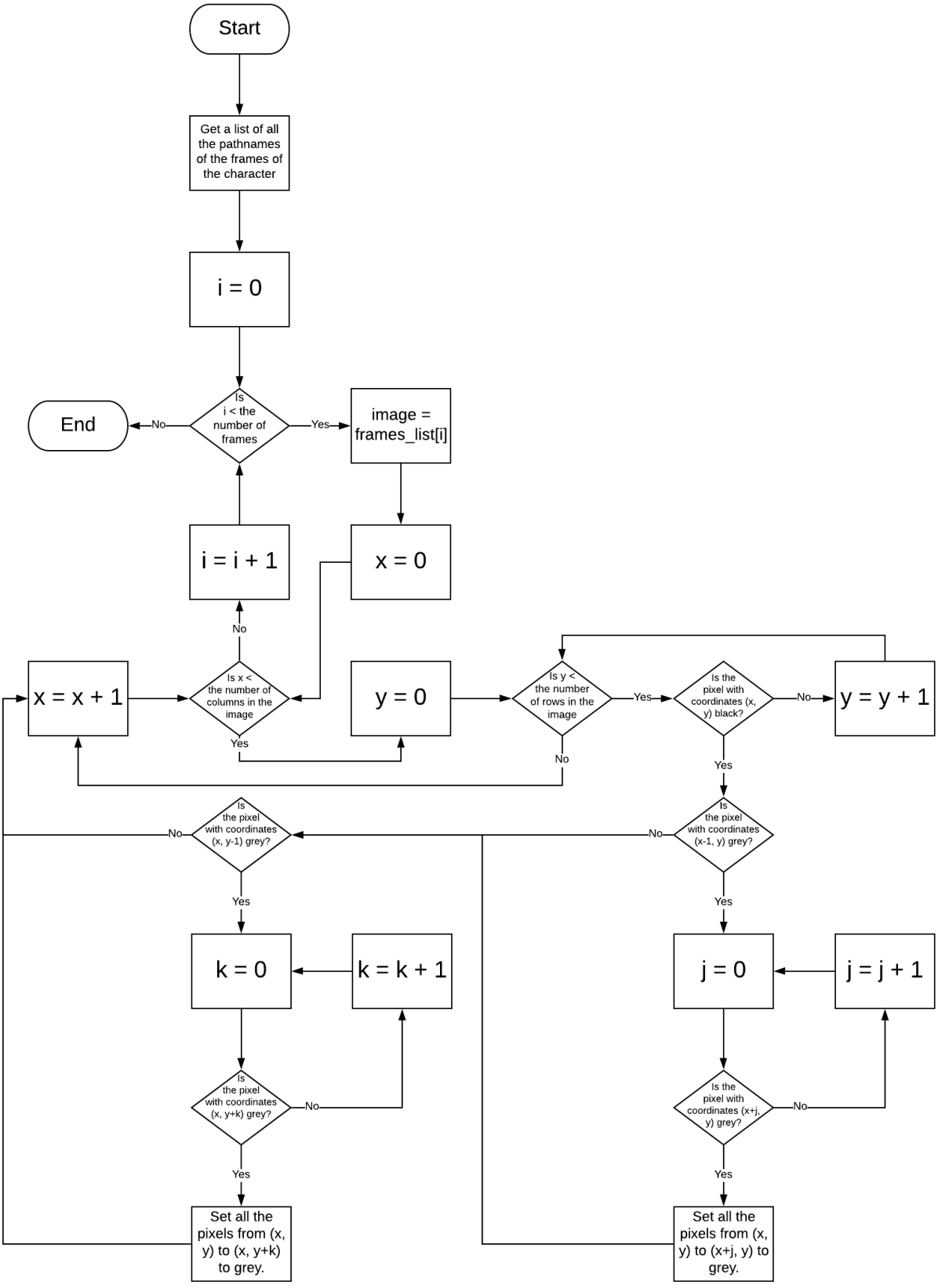
Making the graphics

Characters

The first thing I had to do was to make the characters that can be used as a template for character creation as well as be used if the user hasn't made any custom characters. To make the first character, I used GIMP (an open-source image editor) to draw them. I then had to redraw and move around different parts of the body to create frames for different actions e.g. for running I had to rotate the arms and legs and move them outwards. Here is the original image and an example frame made from adjusting the limbs:


When the character is holding the ball, I wanted the ball to move with the character and look like it was being held so it was between his hand and his body. This wouldn't be possible if the ball was displayed as a separate image. My solution was to have new images for when the character has the ball, and I would make the ball sprite in the game go invisible while the character held the ball. Here is are example frames with the character holding the ball:


I decided to have three base characters because one or two characters isn't enough. However, I didn't want to draw new characters because it took a long time to make the first character. Instead, I decided to slightly change the first character. To make the new characters look somewhat different, I had to get rid of the pattern on the first character's shirt. Unfortunately, I hadn't expected to do this when making the character so I hadn't made a copy without the patterns. Additionally, removing the pattern individually from about 40 images would be extremely tedious. To solve this, I wrote a quick function to do it for me. The function goes through every single image from the first character. For each image, it goes through each pixel. It then checks if the pixel is black. After this, it then checks if the pixel to the left is the top colour (dark grey). If the pixel to the left is the top colour, the function then iterates through the pixels to the right of the initial pixel until it finds another pixel that is the top colour. The function then sets the colour of all the pixels between the first and the last one iterated through and sets them as the top colour. The function then does the same but instead of left and right, it does above and below pixels. The reason why this function gets rid of the pattern is because it removes any black pixels or clusters of black pixels are surrounded by the top colour in one dimension (either horizontally or vertically). Here is a flowchart of the process used to remove the pattern:



Flowchart for removing the pattern on the top of the character

Here is the code for removing the pattern:

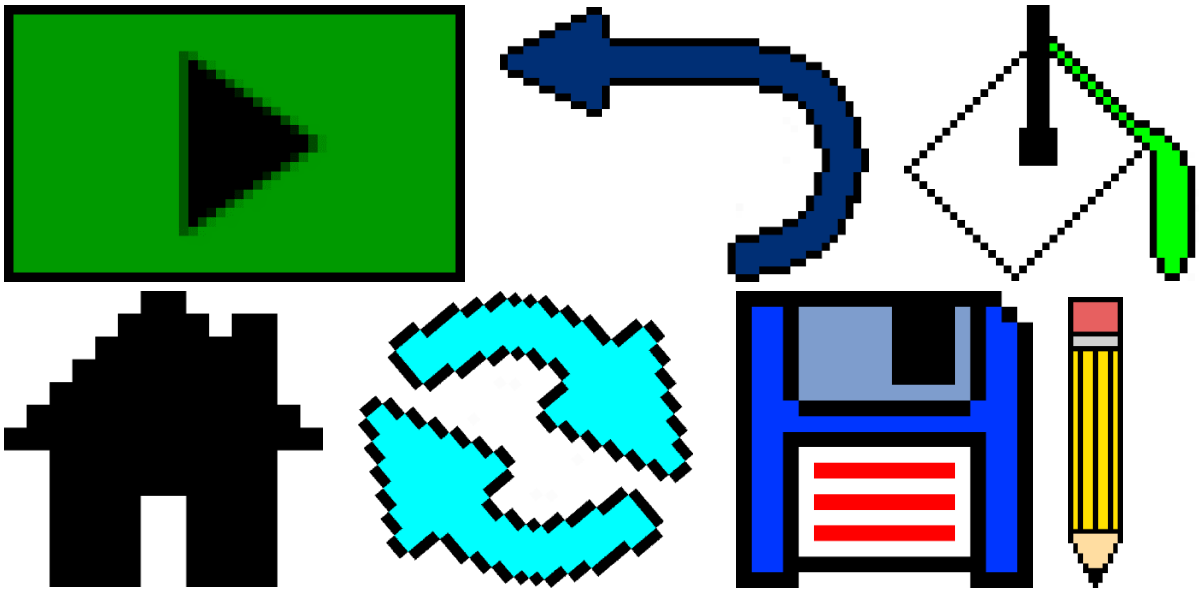
```
# Function for removing the pattern on the old character's top from all of its frames.
def remove_pattern():
    character_dict = make_dict("old_character")  # This function makes a dictionary containing all the images.
    final_images = {}  # This dictionary will be used to contain all of the images.
    for state in character_dict:
        final_images[state] = []  # Create a list for each state. This list will contain all the images from that state.
        for pathname in character_dict[state]:
            im_arr = make_image_array(pathname)  # Make an image array from the image at pathname.
            for x in range(0, im_arr.shape[1]):  # Iterate through the x coordinates of the image.
                for y in range(0, im_arr.shape[0]):  # Iterate through the y coordinates of the image.
                    if (im_arr[y, x] == [0, 0, 0, 255]).all():  # If the pixel at (x, y) is black.
                        if (im_arr[y, x-1] == [39, 39, 39, 255]).all():  # If the previous pixel in x is grey.
                            for i in range(im_arr.shape[1]):
                                # If the row of consecutive black pixels has ended.
                                if not (im_arr[y, x + i] == [0, 0, 0, 255]).all():
                                    if (im_arr[y, x + i] == [39, 39, 39, 255]).all():  # If the pixel is grey.
                                        for j in range(i):
                                            # Set the row of consecutive black pixels to grey.
                                            im_arr[y, x + j] = [39, 39, 39, 255]
                                    break
                        # Repeat the process in y.
                        if (im_arr[y-1, x] == [39, 39, 39, 255]).all():
                            for i in range(im_arr.shape[0]):
                                if not (im_arr[y + i, x] == [0, 0, 0, 255]).all():
                                    if (im_arr[y + i, x] == [39, 39, 39, 255]).all():
                                        for j in range(i):
                                            im_arr[y + j, x] = [39, 39, 39, 255]
                                    break
            final_images[state].append(im_arr)  # Add the image array to the list of images for the current state.
    # This function saves the new character in the correct format.
    make_new_character_directories("new_character", final_images, {}, "old_character")
```

The characters' body shapes were too similar, so I decided to stretch and squash them so one was slightly taller and thinner while the other was shorter and fatter.

The final thing to do was to add new heads and change the colours of their clothes. This is exactly the functionality of the character creation within the game so I decided to wait until I finished that and then used it to make the two new character. Here are the final results:

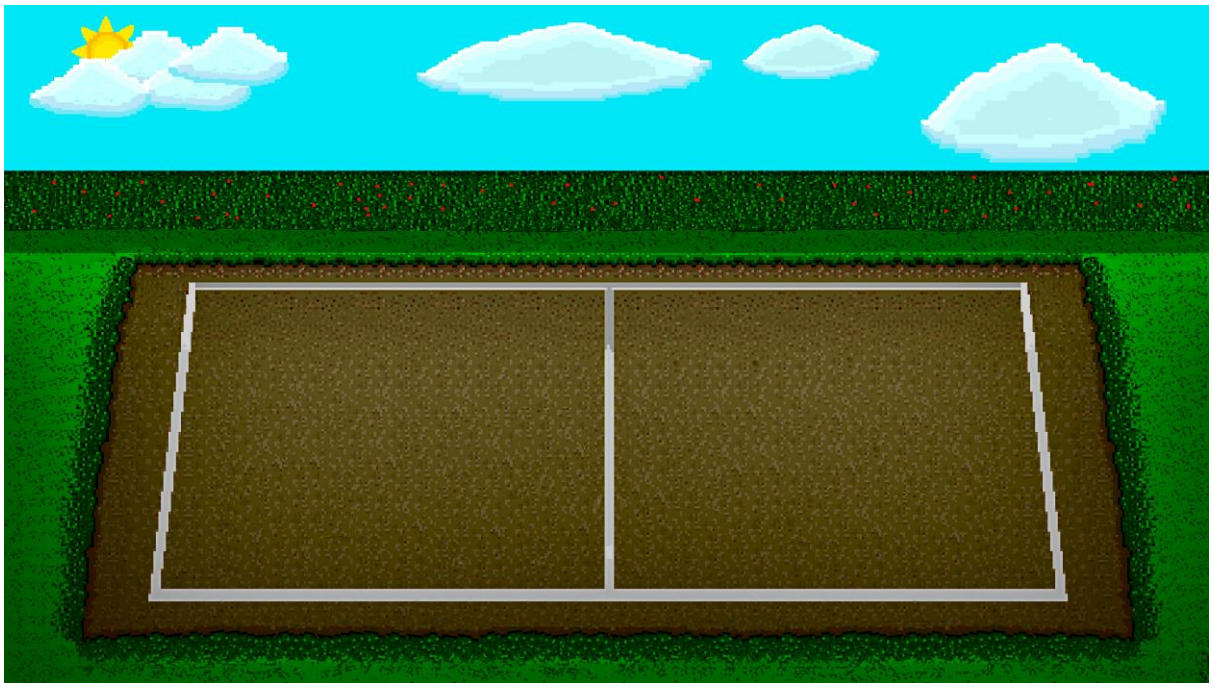
Buttons

I also had to draw the buttons within the game. I used GIMP again for this. My goal was to make them simple and minimalistic in a pixel art style. Here are some example buttons:

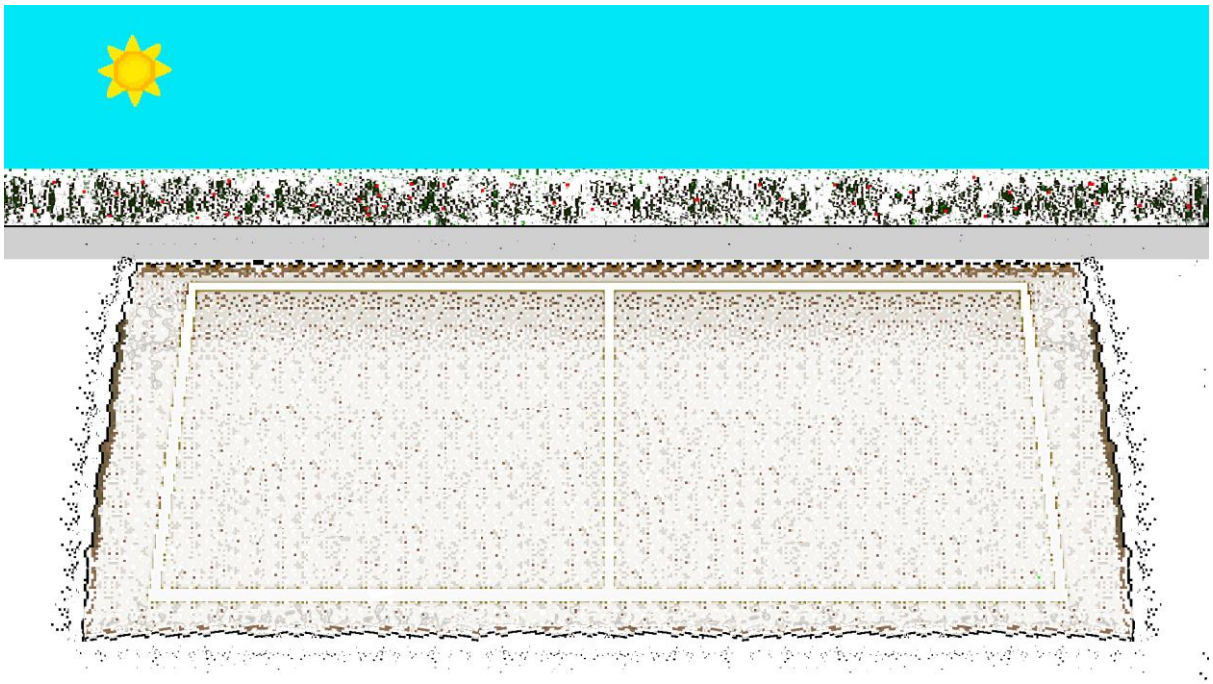


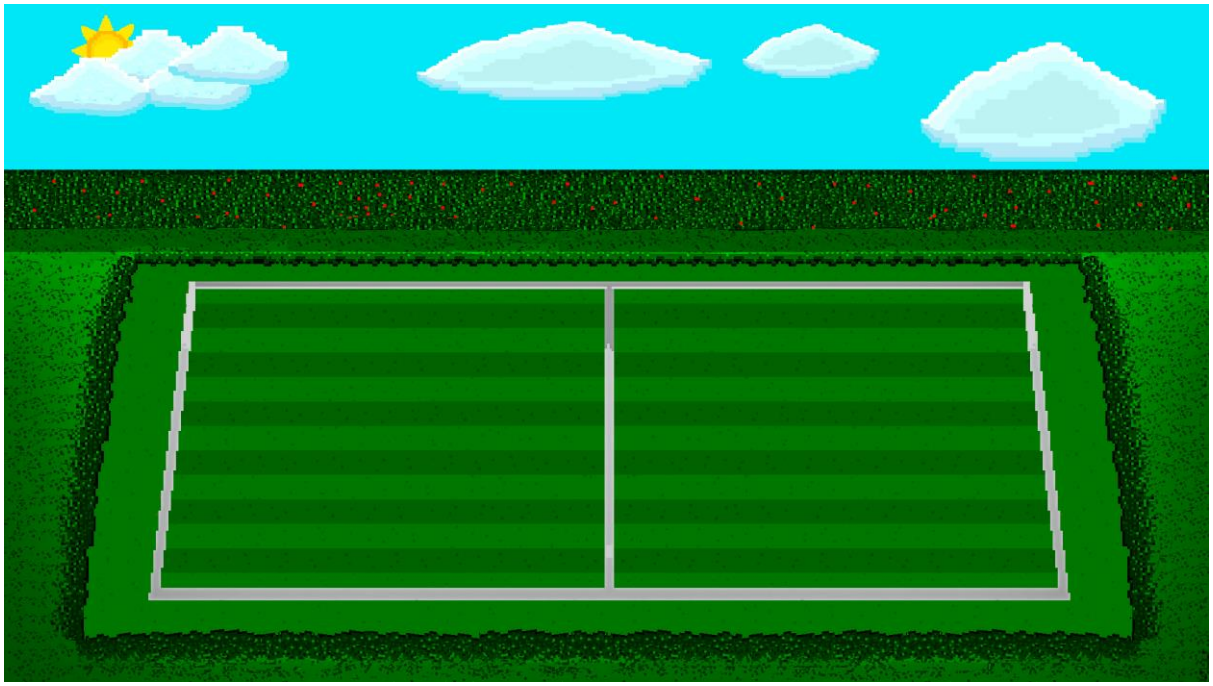
Dodgeball Courts

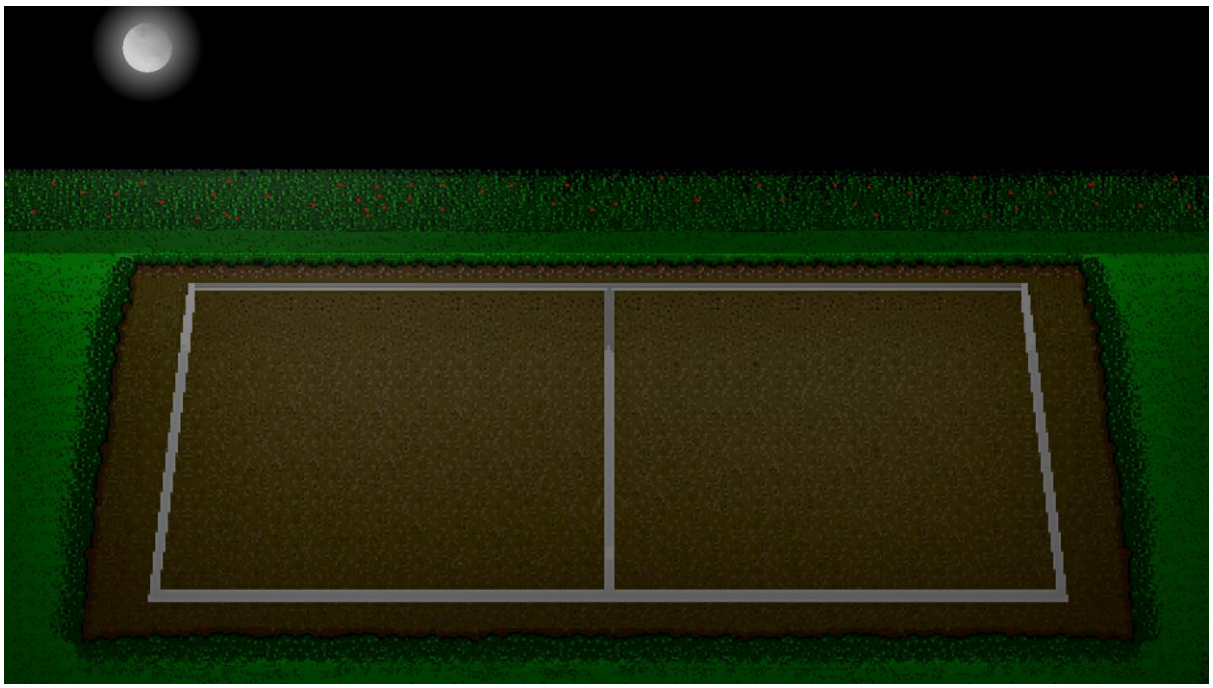
To make the dodgeball courts, I once again used GIMP. I drew the first one:



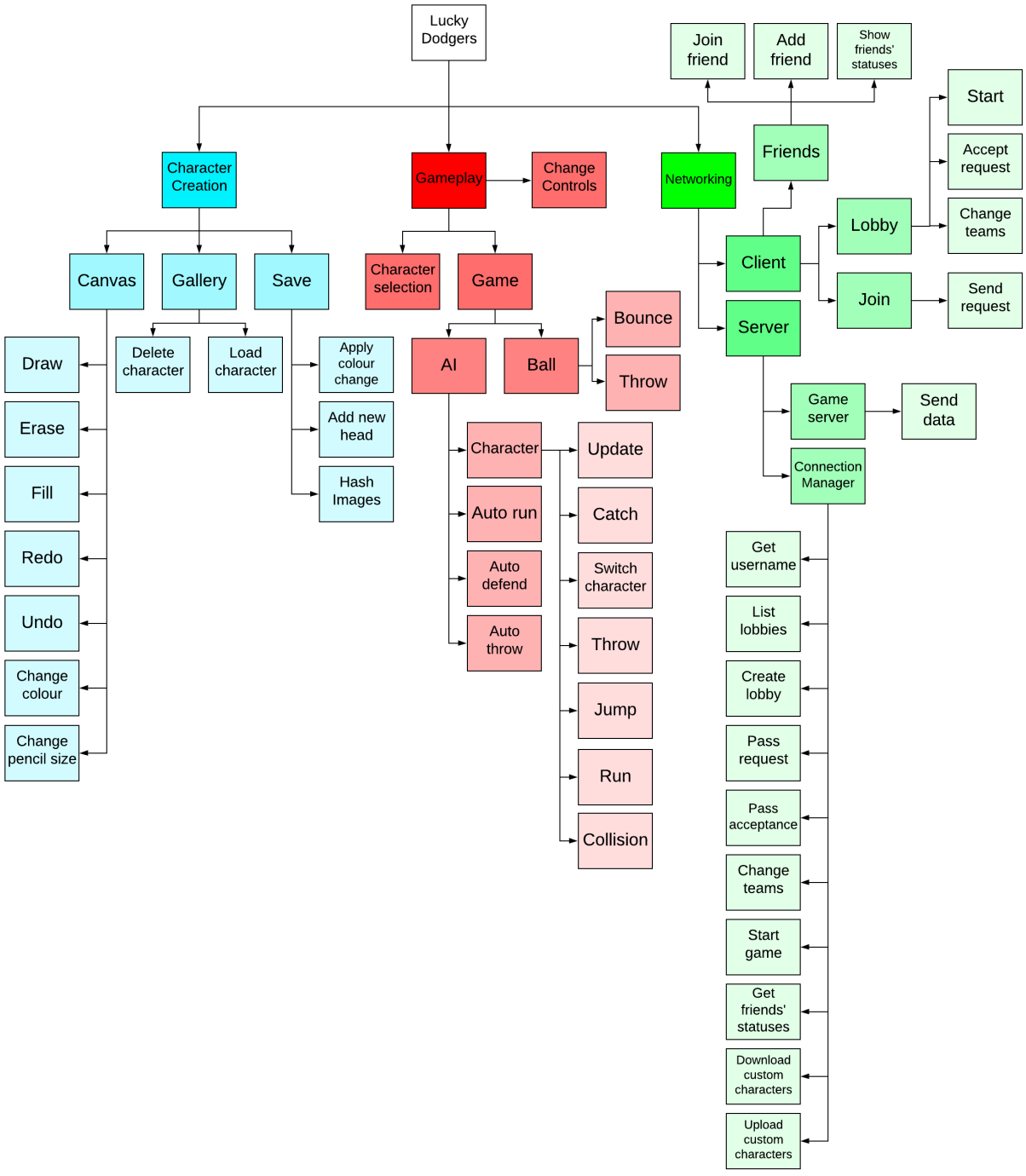
I then used different tools, filters and redrawing to make three different courts from this one:







Top down diagram



There are three main elements of the game:

- Character Creation
- Gameplay
- Networking

Character Creation

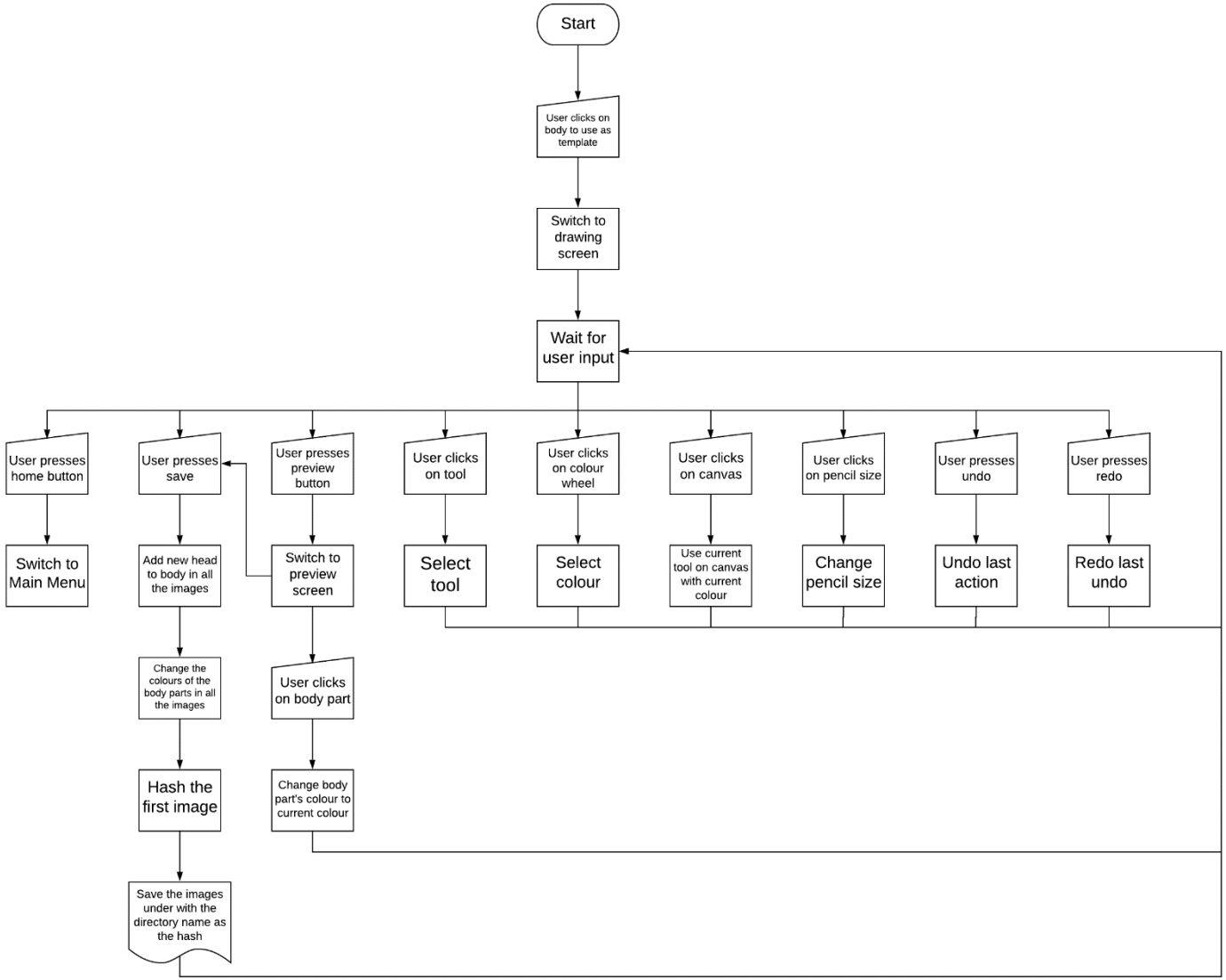
Here is an IPSO chart for character creation

Input	Process	Storage	Output
Choose tool Click on canvas Pick colours Undo/Redo	Draw Erase Fill Change colours Apply undo/redo Show preview Apply changes Hash image	New character directory containing every frame saved as a png	Success message

The character creation process should go as follows:

1. The user is given a plain canvas with a grid with some simple tools – pencil, rubber and fill. The user can also change the pencil/rubber size. There will be a colour picker and undo and redo buttons.
2. The user draws a head for the character on the canvas (body is pre-made, only the colours can be changed because animating a user drawn image would be almost impossible unless they drew all of the frames. Also, the user would otherwise be able to draw a blank character).
3. The user can preview the character at any time to see what they will look like with the new head and can change the colours of the character's clothes and skin colour (for the hands and neck).
4. Once satisfied with their new character, the user can then save it.
5. The user can now use the new character in-game and go back and edit the character whenever he/she wants.

Character Creation Flowchart



Canvas

The canvas is made up of initially transparent rectangles for each square in the 30x30 grid. The colour of each rectangle object is saved in an ordered dictionary (it is ordered to make it convenient to turn into an image array later) as the value and the key is a tuple with the x and y position of the square.

e.g. {(30, 40): [255, 0, 0, 255], (30, 60): [0, 255, 0, 255]} has two squares: one red square at position (30, 40) and one green square at (30, 60). The reason why the colours have four elements is because they are in RGBA format where a is for alpha (opacity).

Change Colour

When the user clicks on the colour wheel, the colour they clicked on is set as the current colour. The user can also drag the colour wheel to get darker colours.

A new button is added to the palette with this colour. When the user presses this button, the colour of the button is set as the current colour.

Change pencil size

There are three pencil sizes: 1x1, 2x2 and 3x3. When the user clicks on one of them, the pencil size is set as the one clicked.

Draw

When the user uses the pencil tool and clicks on a square, the coordinates of the touch are passed as a parameter, and the current colour is assigned as the new value in the canvas dictionary. This process is repeated with the adjacent squares if the pencil size is 2x2 or 3x3.

Initially, I intended the drawing to be that a new rectangle is created each time a user clicks on the canvas, but this is more memory-intensive and creates complexities when trying to erase.

Erase

If the rubber tool is selected, the same process as draw happens except the colour is set to transparent.

Fill

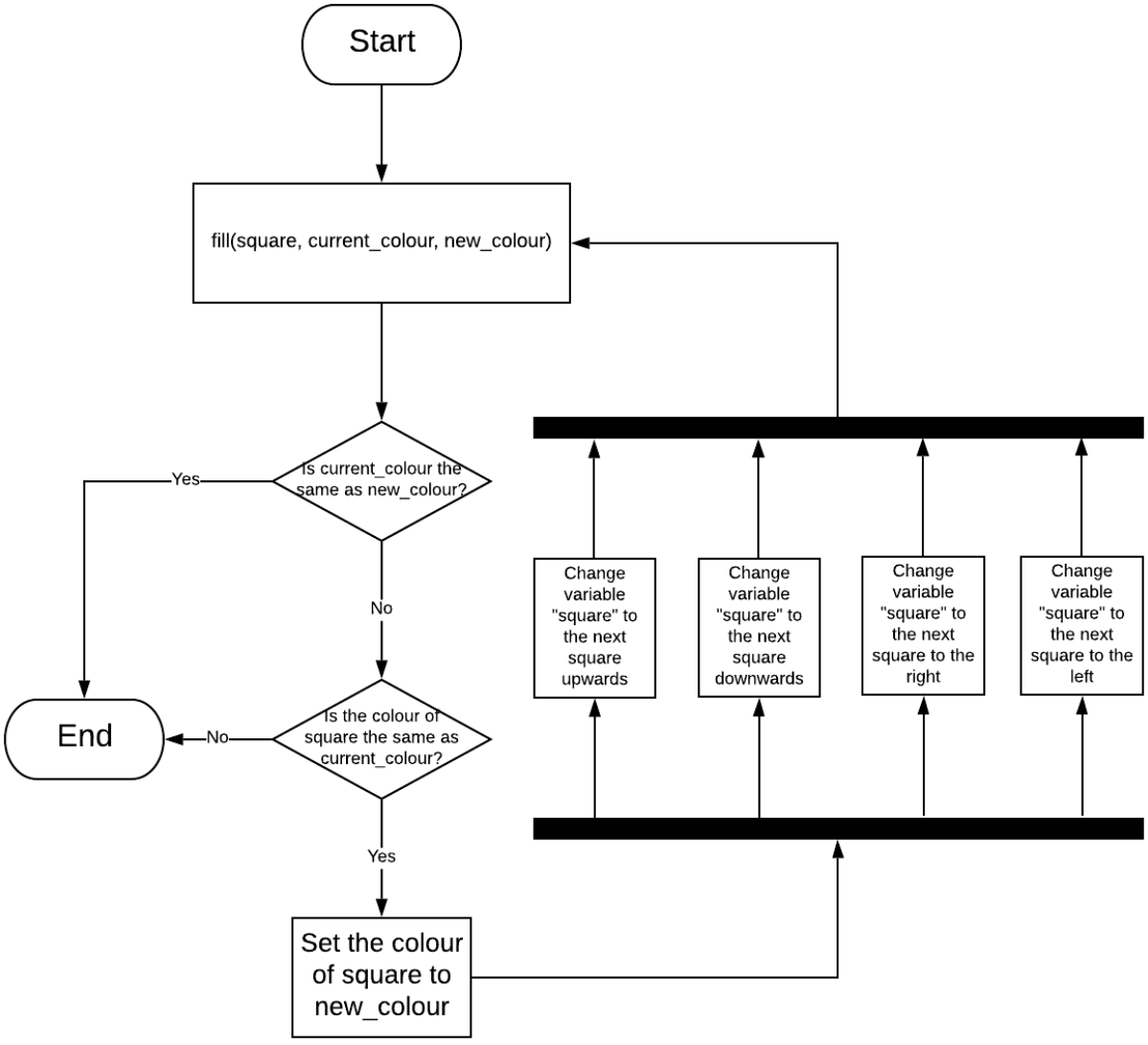
The fill tool uses the flood fill algorithm. This is a recursive function. The function takes in three parameters:

- square - the square being filled.
- current_colour - the original colour of the initial square.
- new_colour - the new proposed colour.

The function has two stages:

1. Check if the current square is valid to be filled. For this, new_colour must not be the same as current_colour (this would mean that the square is already the right colour) and the colour of square must be the same as current_colour (otherwise the square is not part of the same section as the first square).
2. Fill the square and call the function again but with the squares to the left, right, above and below the current square.

Here is a flowchart of the function:



Undo and Redo

The undo and redo buttons are linked to two different stacks.

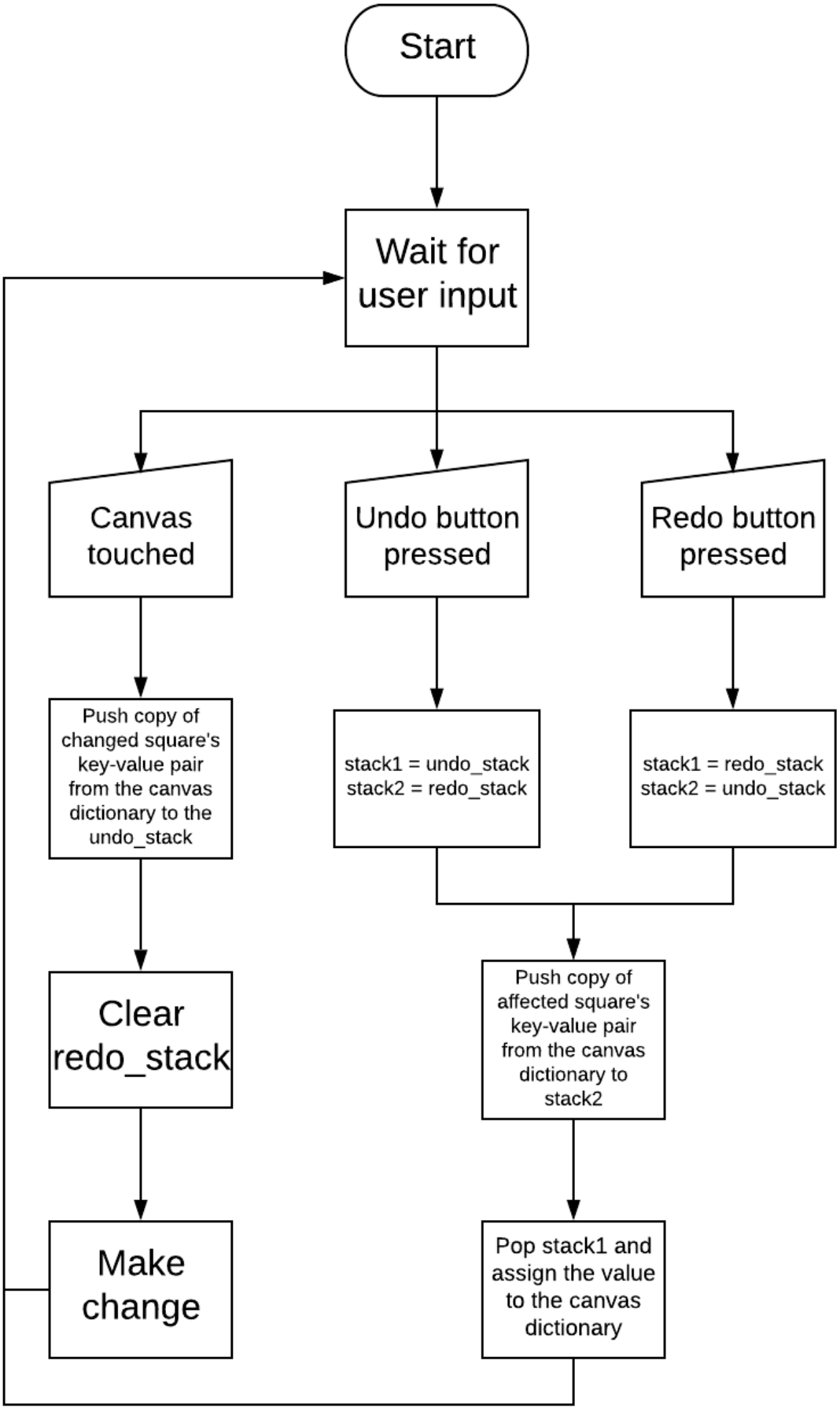
The stacks will have a limit of 99 changes to save memory since people very rarely would need to undo or redo 99 changes, and at that point it would be quicker to restart the drawing.

Before a change is made to the drawing, the key-value pair of the square about to be changed is copied and pushed into the undo stack from the canvas dictionary. The redo stack is cleared when a change is made.

When the undo button is pressed, the key-value pair of the square about to be reversed is copied and pushed into the redo stack from the canvas dictionary. Then the undo stack is popped, and the key-value pair is put back into canvas dictionary, reversing the change.

When the redo button is pressed, the key-value pair of the square about to be re-changed is copied and pushed into the undo stack from the canvas dictionary. Then the redo stack is popped, and the key-value pair is put back into canvas dictionary, redoing the change.

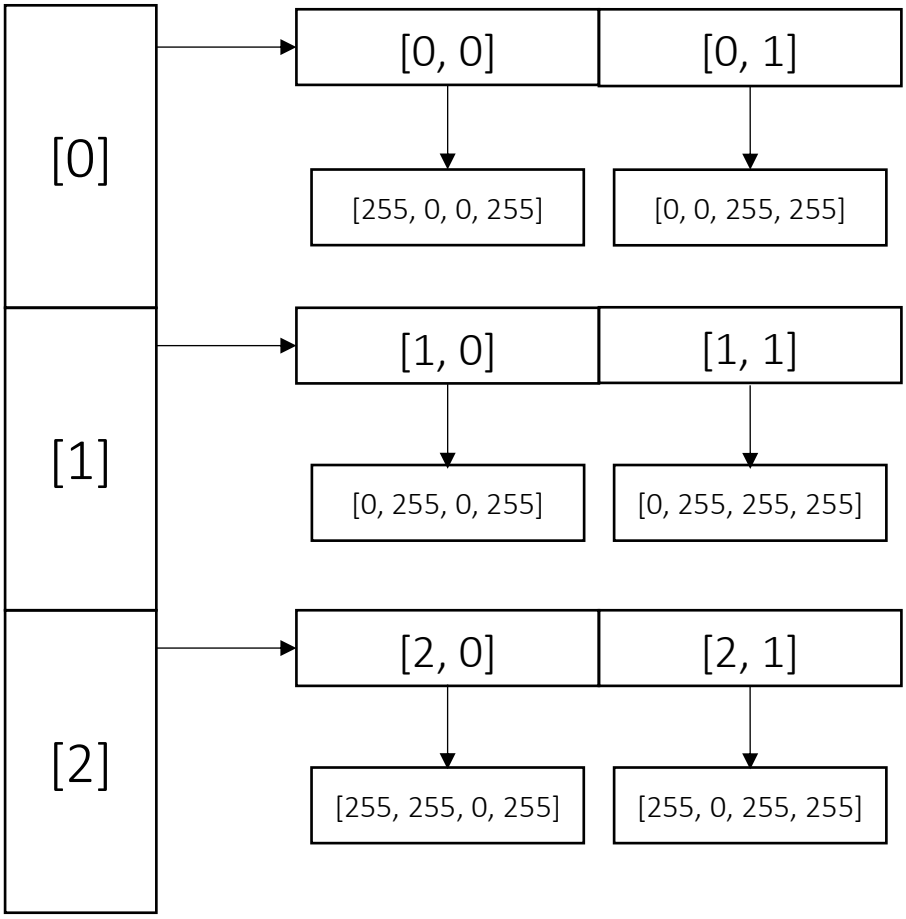
The flowchart for this process is on the next page:



Undo and redo flowchart

Save

To change the head of the character to the new drawing by the user, image arrays must be used. Image arrays are data structures that represent an image. They are formatted as 3-dimensional arrays. The outer layer of subarrays represents each row of the image. The next layer represents each pixel within the row. Each pixel is represented by four values, the RGBA values of the pixel. To access a pixel from the image array, you can use indexing. For example, if you want to print the value of a pixel with x-position as 30 and y -position as 50, the code is print(image_array[50, 30]). Note that the y-coordinate comes first. This is because the first layer of subarrays represents each row. Here is an example of an image array's structure with the image array representing the block of six pixels:



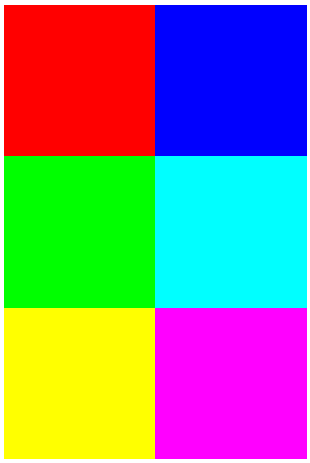


Image the array represents

Visual representation of an image array

To convert the original character images into image arrays, I use the function imread() from the library opencv. However, I have to add a flag (cv2.IMREAD_UNCHANGED) as a parameter to ensure that the alpha channel is kept. I then have to flip the order of the first three elements of the arrays for each pixel because opencv uses BGRA rather than RGBA e.g. a yellow pixel is initially [0, 255, 255, 255] but I want it as [255, 255, 0, 255].

Add new head

Converting the user's drawing into an image array is more complex than just using imread(). First, I have to make an empty image array with 30 rows and 30 columns (due to the grid being 30x30). Next, I have to iterate through the colours dictionary. The colours dictionary is ordered so all that needs to be done is to replace each element in the image array with the corresponding value in the dictionary.

Now that we have everything as an image array, we can replace the head. However, we need to know where the head is to replace it. To do this, I created an info text file for each character which contains the dimensions of the head relative to the top-left-most pixel of the head as a list e.g. if the head is 100x100 pixels with 20 pixels to the left of the top-left-most pixel, the dimensions will be saved as [0, 100, -20, 80]. The dimensions are in the form [start_y, end_y, start_x, end_x] where start_y and x are the y and x positions relative to the top-left-most pixel where the head starts and end_y and x are the same but for where the head ends. This system works for all the character images because the size and rotation of the head stays constant in all animations but the position of the head changes.

To find the top-left-most pixel, I have to iterate through each row of the image array until I find a pixel that is not transparent. I then use the head dimensions from the info text file to create a start and end point of the head e.g. if the top-left-most pixel is on row 1 and column 320 and the head dimensions are [0, 100, -20, 80], the start row is 1 and the end row is 101 and the start column is 300 and the end column is 400. A list of [start row, end row, start column, end column] is returned.

Next, what needs to be done is turning the replacement head image array into the right size. I use the opencv function resize() to do this with the dimensions in this case being 100x100. I have to turn off interpolation for this otherwise the image gets smoothed. To do this, I add the flag cv2.INTER_NEAREST.

Now all that needs to be done is to place the new head in the original image. We know the position that the new head needs to go so combining the images is simple. Using the list containing the start and end rows and columns from earlier we can use array slicing to insert the new head. The process will be:

```
original_image[start row:end row, start column:end column] = new_head
```

The canvas dictionary containing the drawing of the new head will be saved as a file as well so that the character can be loaded and edited.

Apply Colour Change

The next process is to allow the users to change the colours of the character's clothes and skin. To do this, the program has to know the positions of all the pixels that are in different sections of the character. The way to do this is by having an info text file for each character which gives the colours of each section. This allows the program to find all the pixels of that colour and then change it to the new user selected colour. However, this is extremely inefficient and would take the program a very long time since it would have to search every single pixel in every image and check its colour.

The solution to this is creating a large data structure in advance and then serialising it so it can be reused. The data structure is a dictionary with keys as pathnames of the individual frames and values as a dictionary. This dictionary has keys as each section (i.e. shoes, top, bottoms, skin) and values as another dictionary. This third dictionary has keys as the column containing the colour and the values as a list of the rows which are the certain colour. However, this can be made more efficient. Instead of putting all the rows with the certain colour, putting only the first and last row in a sequence of rows of the certain colour would massively reduce the number of elements in the data structure. For example, instead of [120, 121, 122, 123, 124, 220, 221, 222, 223], the list would be [120, 124, 220, 223].

An example key-value pair from the dictionary would be:

{"Images/char1/idle/1.png": {"shoes": {20: [50, 90, 100, 150], …}, …}, …}

As you can see, this dictionary is massive and would take very long to create if it was made very time the user saved a character. Generating the dictionary in advance and saving it for reuse is massively more efficient.

To serialise the data structure, I can use the library pickle. This will save the data structure as a file which I can then load into a variable at any time. When the user picks a new colour for the section, the program turns all the pixels given in the data structure that are in that section into the new colour.

A dictionary called section_colours will contain the new chosen colours by the user. When the user picks a new colour for a section, the value with the key of that section will be set as the chosen colour e.g.

{"shoes":[0, 255, 0, 255],"skin":[255, 0, 0, 255],"top":[0, 0, 255, 255],"bottoms:[0, 0, 0, 255]}

would mean that the user chose green shoes, red skin, blue top and black bottoms.

Hash Images

The user-created characters will be saved on the local device and the online file system so the characters can be used offline as well as being easily shared online without having to upload the images every time the character is used. When a user adds a friend, the user downloads the friend's characters from the file system if they don't already have the character saved. The characters will need a hash code to determine if the player has the most updated version of the character.

Usually, image hashes are used to find similarities between images and try to ignore artefacts and other results of compression. However, the image hash I need for the program should work like a normal hash. It should result in a different hash even if just one pixel is different.

Initially, I planned to make the hash just work on the full image. However, a more efficient method would be to hash just the new head. This wouldn't consider the base body and the colours of the clothes and skin though. To solve this, I can add the name of the character whose body is being used and I can hash the section_colours dictionary and add that to the end of the overall hash.

To hash the head, I first crop it to make the hashing quicker and reduces the similarities between images. To do this, I have to find add all of the values within the inner layer of the image array (the colour array). If this value is zero, that pixel is empty. Next, I flatten all the columns in the same way. I then get the indices of all the full columns without value of zero (a non-empty column). I repeat this process for rows. I now get the minimum and maximum indices of the non-empty rows and columns i.e. I find the first and last non-empty rows and columns and store them in a list. I then return the image array with only the rows and columns between the values in the list.

Next, I use NumPy's tobytes() method to convert the image array to bytes. I then use python's inbuilt int.from_bytes() function to convert the bytes to an integer. This allows me to manipulate it. To hash something, the original value has to be diffused. This means that two similar values result in very different hashes. There are many methods for diffusion. I will use bit shifts, XORs, multiplication and modulo as these are very quick processes and are good at diffusing. I will do the first three processes multiple times to improve the diffusion.

First, I will bit shift the integer to the left by a constant. I will then XOR the original image with the bit shifted image. Finally, I will multiply the integer by a constant. These processes will be repeated three times.

Next, the most important part of the hash is the modulo. This is essential for reducing the size of the integer to the table size. At this point, the integer is massive. The hash will be used for directory names, so the character limit is 248 on Windows and 255 on Linux. However, we don't need the hash to be that long to prevent collisions and the larger the number we

modulo by the slower the hash will be. A more reasonable size would be about 50 characters. The final hash won't be an integer (this would make the table size 10^50 if we wanted 50 characters). Instead the hash will be converted to base 64. This means that the table size can be 64^50 which is approximately 2x10^90. When using modulo for hashing we want to modulo by a coprime of the initial number. This is to deal with any patterns that may occur in the number which would result in collisions otherwise. Fortunately, it isn't hard to find a prime close to 64^50 because 2^n - 1 is always prime (Mersenne primes). Therefore, I will modulo the integer by 64^50 – 1.

Finally, we add the section_colours and the name of the original character to the end of the hash. The new character will be saved under a directory with its name being the hash. This makes checking for duplicates and checking if the user already has the image very easy.

A flowchart of the hash is on the next page:

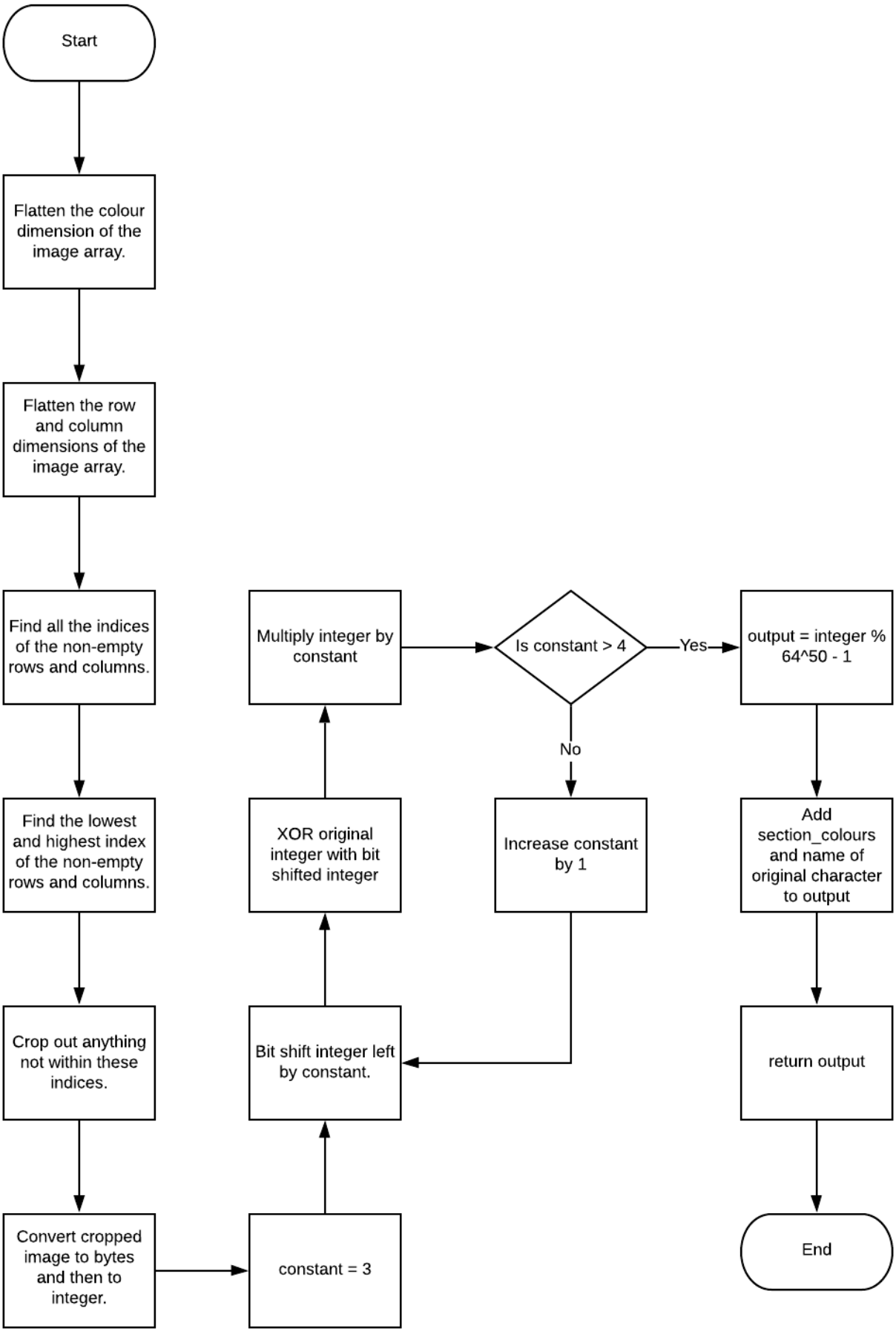
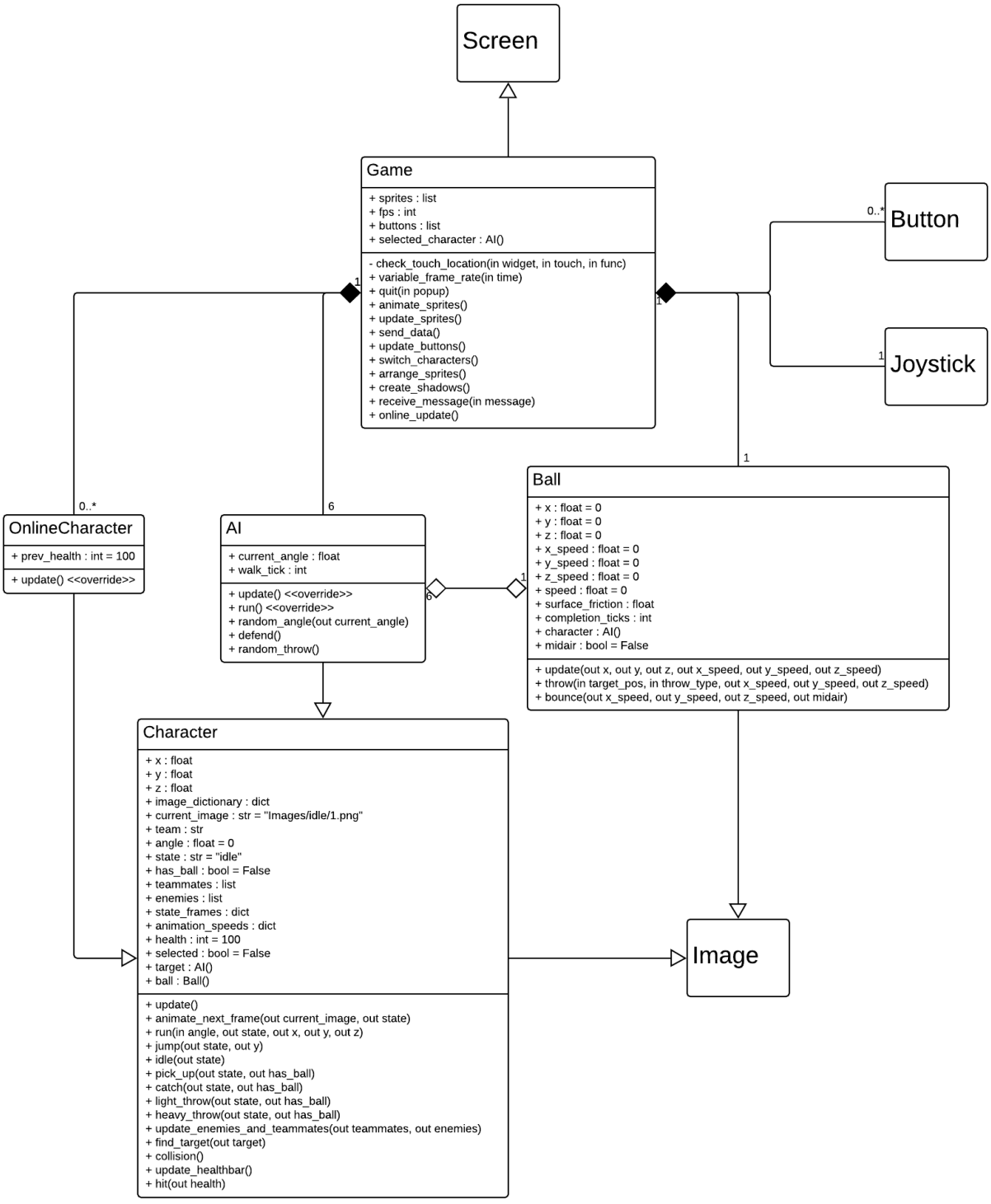


Image hash flowchart

Gameplay

The classes for the gameplay should interact as following:



Here is an IPSO chart for gameplay

Input	Process	Storage	Output
Joystick movement Button presses	Update Animate character Run Jump Pick up ball Catch Throw Collision	None	Next frame of the game

Variable Frame Rate

To get maximum performance on all device, a variable frame rate has to be implemented. In Kivy, actions can be put on a schedule where they repeat every x seconds. I set the game schedule to repeat every 1/fps seconds. I then have to change the fps to the maximum amount that doesn't result in slowdown (caused if the device can't process frames in time). Kivy's schedule passes a time parameter every tick. This parameter has the value of the amount of time since the previous tick. If the time is greater than 1/fps, slowdown has occurred.

To get the right fps, I have to maximise fps until slowdown occurs. What I do is increase the fps by 1 every tick where the time is the same as 1/fps (within a certain threshold). I decrease the fps by 1 every tick where the time is greater than 1/fps. The reason why I constantly change the fps every tick rather than find a right value and keep it at that is because the speed of the processing changes often. A background application could start using slightly more or less resources at any time so a frame rate that adapts quickly is essential. Additionally, the more often the fps adapts, the more accurate it is.

Character Selection

The user will be able to select characters from a screen before starting a game. The screen will show all the characters the user has. The user selects three characters and then confirms. When a character is pressed, a coloured box appears around the character. If the user wants to change their choices, they can click on a cancel button which clears their character choices. The AI opponent's characters are randomly chosen.

Characters

Animations

The characters will be able to do several different actions (run, jump, catch, heavy throw, light throw). Each of these will need to be animated. Therefore, each frame of each action will have to have an image. These will be saved in the following way:

./Images/character_name/action/frame.png

For example: ./Images/dave /jump/2.png

For the program to quickly access the files, a dictionary is to be used with key as the name of the action and value being a list of the pathnames of each frame for the action.

The program has to animate the next frame at each update. There will be a dictionary with the key being the action and the value being the current frame. Each update, the current frame is incremented by one until it reaches its maximum, it then resets to 1. However, some actions need to be animated slower than other (i.e. two clock ticks per frame). To solve this, I will use an animation_speed dictionary with each action's speed as the increment rather than one.

Movement

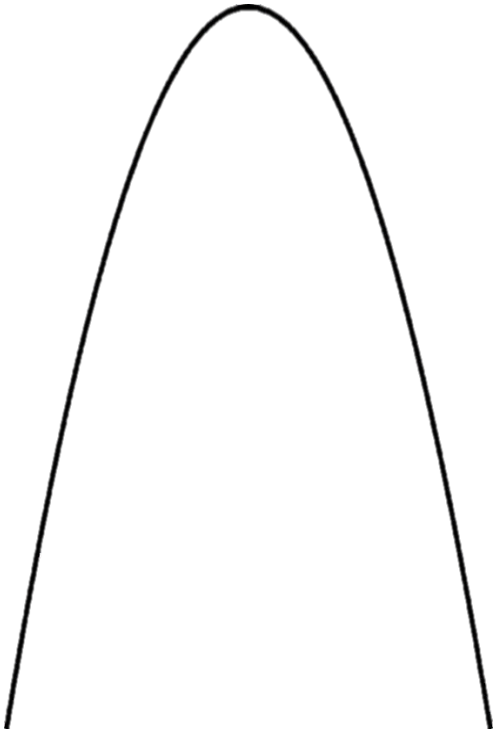
The positions and movement of the characters has to be done in a 3D space so there needs to be x, y and z positions. However, since x and y are built-in to Kivy, the x, y and z axes are slightly different from the traditional way. x and z are the same as normal, but y is equal to z + height above ground. The characters will be moved by a virtual joystick on the screen. When the user moves the joystick pad, the angle is sent to the currently controlled character. The character then moves in the x direction by a constant * cos(angle) and the y and z directions by a constant * sin(angle). This will ensure the character's speed remains the same in all directions.

However, there needs to be boundaries for the movement since the character would otherwise run off of the screen. These boundaries will initially be the court. However, when the ball is outside of the court, the boundaries would have to expand so the players can get to the ball. When the ball returns to within the court, the boundaries change back to the court. To apply these boundaries to the characters, the movement of the character in the x direction is prevented if either cos(angle) < 0 and the x position < the left boundary or if cos(angle) > 0 and the x position > the right boundary. The same applies to z movement except sin is used over cos and the vertical boundaries are used.

Actions

For most of the actions, all that needs to be done is to set the state of the character to that action. The ones that need additional code are jump, switch character, pass, heavy throw and light throw.

For the jump, the character's y position needs to change in a parabola like this:



To do this, I can increase the character's y position by a constant multiplied by (the total of number of frames / 2 minus the current frame). E.g. if there are 8 frames in the jump action and the current frame is 2, I increase the y by constant * (8/2 - 2) which is constant * 2. If the current frame is 7, I increase the y by constant * - 3 (I decrease the y).

For switching characters, I have to change the current character to the next character in the list of controllable characters.

Shadows

The characters and the ball also need shadows. To do this, I can create an ellipse with the size being based on the sprite's size and height above ground (when a character or ball is in mid-air, their shadow is larger).

Ball

I decided to make the ball explode after being held for a long time to prevent time-wasting. When the ball explodes, it damages whoever is holding it and bounces towards the other team.

Throw

For the throws, I first need to make a targeting system (I decided the throws should auto-lock on to the enemy the controlled character is aiming at rather than being completely controlled by the user since that would make it too hard to hit the opponent). This targeting system has to go through the list of targets (enemies for heavy and light throw and teammates for pass). It then finds the one whose angle with the controlled character is closest to the angle the user is aiming at with the joystick. This is now the target.

Now, the ball class's throw function is called with the parameters of the throw type (heavy, light or pass) and the position of the target. The throw function first calculates the distance from the ball to the target using Pythagoras' theorem. Next, the total speed value is decided based on the type of throw. The time the throw will take is then calculated by dividing distance by total speed. Now, the individual speeds in each direction is calculated. x and z speeds are calculated by dividing x and z distances respectively from the ball to the target by completion time. The y speed is just the z speed plus a constant to give a slight initial upwards trajectory. At each frame, the speeds are added to the ball's position. Additionally, at each frame, the ball's y speed is decreased to simulate gravity.

Bounce

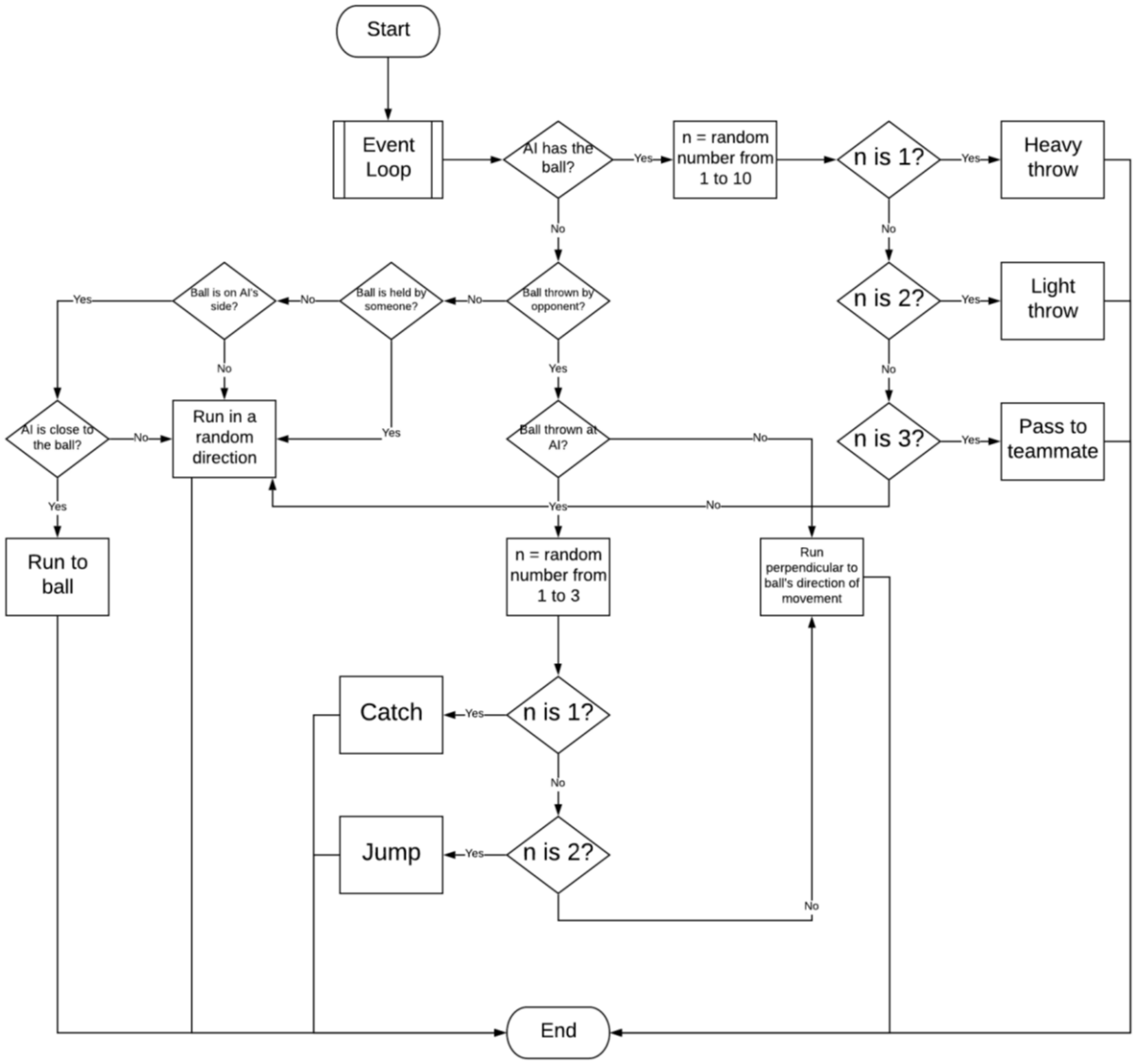
When the ball hits the ground, a player, or one of the edges of the screens, it bounces. When the ball hits the ground, the x and z speeds are multiplied by the surface friction constant which is determined by the stage the game is played on e.g. a snow stage has a friction constant of 0.3 but a grass stage has a friction constant of 0.7. The y speed becomes the z speed plus the vertical speed multiplied by the negative friction constant. If the ball bounces off a boundary or a player, the same applies but x or z speed are reversed rather than y speed.

When the ball collides with a character, the program checks if the character is in the catch action. If the character is, the character now has the ball. If not, the character is hit by the ball and loses health.

When the ball is vertically slow enough, it rolls rather than bounces. Its speeds are multiplied by surface friction every tick now, so the ball slows down quicker.

AI

The next step is to create an AI class. To allow the player to switch characters, the AI class would have to be able to switch between acting like a player-controlled character and acting like an AI. For this, the AI class can inherit the character class and have a few extra methods and override some old methods. When the class is being controlled, it doesn't call the additional methods and the overridden methods only run the parent class method. The AI's functionality should be like this:



AI functionality flowchart

Run

The first method to be overridden is the run method. In the character class, the method receives the parameter "angle" which is given from the joystick and the method makes the character run towards this angle within the boundaries. For the AI, there is no input from a joystick. Instead the AI must base its angle on the situation it is in. If the ball is in the AI's side and is not mid-air plus the AI is closer to it than its teammates (otherwise all the AIs would run to the ball), the AI should run towards it. If the ball has been thrown at the AI by an enemy, the AI should run perpendicular to the angle from the AI to the ball. If the ball is being held by anyone or the AI is not close enough to run to the ball in the first situation, the AI should move around, changing its direction randomly at random but slightly large intervals (with small intervals, the AI's movement would look like it was vibrating rather than smoothly moving).

For the first two situations the program must first find the angle from the AI to the ball. This can be done by finding the arctan of the z distance from the AI to the ball over the x distance between the AI and the ball. For the first situation, nothing has to be done to this angle. For the second situation, the angle has to be changed by half pi. If the ball is moving towards below the character, the angle is increased by half pi, otherwise it is decreased by half pi. For the third situation, there is a walk_tick variable which is increased each time the character moves in the same angle. If the walk_tick reaches a random number between 5 and 15, a new angle is created and the walk_tick is reset to 0. Otherwise, the previous angle is used. Now the parent class run method is called with the parameter being the angle the method has created.

Defend

When the ball is thrown at the AI, the AI should sometimes react with a defensive action (jump or catch). The program finds the angle from the ball to the AI and finds the angle of the ball travel. If these are close enough to each other, there is a 1/3 chance the AI jumps, a 1/3 chance the AI catches and a 1/3 chance the AI just moves. To do the actions, the parent class' methods for the actions are called. When the AI has the ball, it needs to throw the ball at some point. There is a 5% chance at each frame for the AI to throw if it has the ball. The AI can do a light throw, heavy throw or pass to a teammate.

Throw

If the AI has the ball. There is a chance that the AI throws the ball. The AI has an equal chance to heavy throw, light throw or pass and the AI could throw the ball at any time.

Controls

The default controls will be a black and red joystick in the bottom-left and three translucent, grey buttons in the bottom-right. I decided to allow users to change their controls. The things they will be able to change are the positions, sizes, colours and opacities of each button and the joystick individually.

When the user clicks on an object, that object becomes selected. The program will create a box around the object to indicate that it is selected.

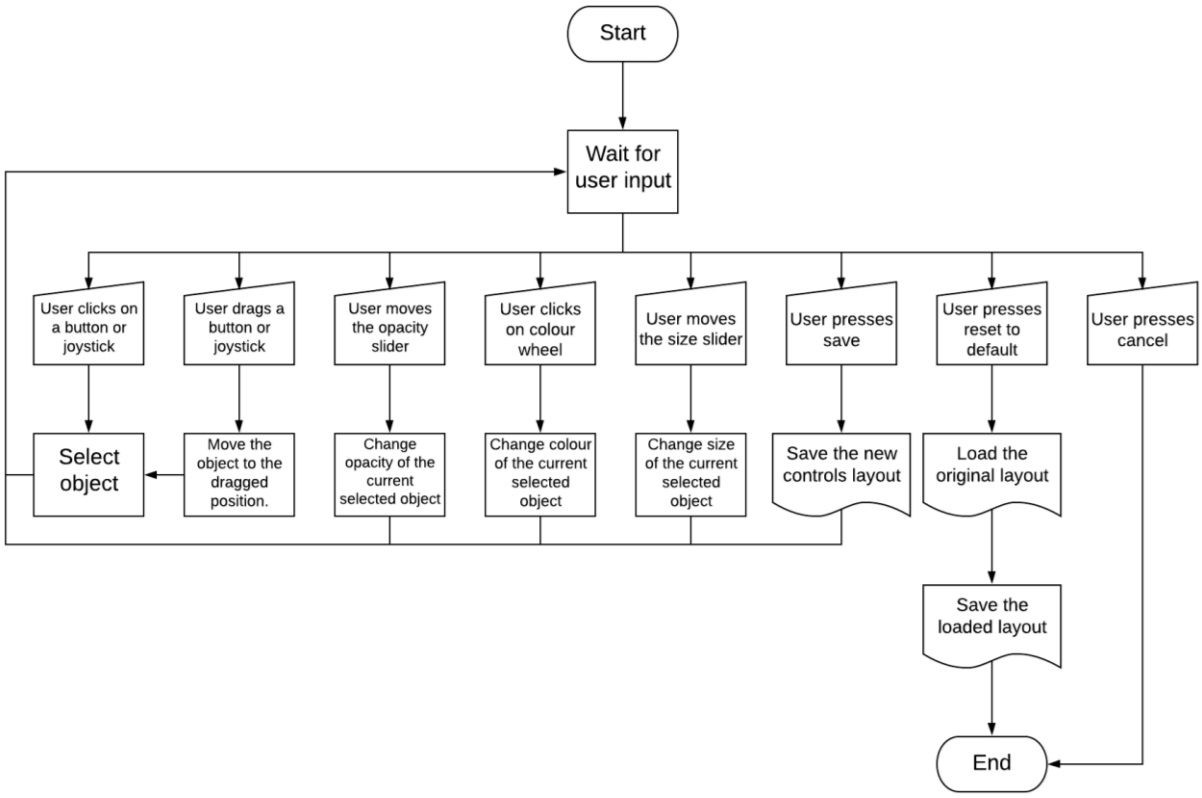
I will have another colour wheel like in the character creation to allow users to pick colours. When the colour wheel is pressed, the colour chosen will be applied to the selected object.

I will have a size slider and an opacity slider to change these values for the selected object. I will also allow the users to drag on widgets to move them, although I will have boundaries to prevent buttons from going off the screen.

The controls layout will be saved in a text file. This will contain the positions, sizes and colours of each widget.

The default layout will be saved in a separate file allowing the user to reset to default. This will overwrite the controls layout file with the default file.

Here is a flowchart detailing how the controls changing works:



Networking

The client-server network will have a star topology. For the network, I had other choices such as a peer-to-peer network using Google Play Games Services or my own peer-to-peer network. I decided not to use Google Play Games Services because it would only work on android and therefore wouldn't allow cross-platform multiplayer. I decided not to make my own peer-to-peer network either because I would still need a server for matchmaking, or I would require users to enter each other's IP addresses. Furthermore, if the host of the game disconnected, the whole lobby would be disconnected which is especially problematic on mobile where connection is not always consistent. Additionally, some issues with syncing could occur in a peer-to-peer network.

The connection between the clients and the server will be TCP rather than UDP. I have chosen TCP because it provides error-checking and therefore guarantees delivery of all the data sent and it is received in the correct order. This is important because otherwise changes may be made in the wrong order. Additionally, the character files being sent may end up corrupted and wouldn't work if any data is lost or in the wrong order.

The server will work as both a connection manager and a game server. As a connection manager, the server works as a middleman between clients, passing on messages and storing data. As a game server, the server will take in game information from clients and distribute it to the other clients.

The server will be waiting for messages from clients and will then respond appropriately depending on the message. The clients will send messages based on the user's actions within the application e.g. when the user clicks on host, the client sends a message to the server and the server does an action based on this.

The server will have three dictionaries:

- clients – a dictionary with the keys being the usernames of the clients and the values being instances of the connection object between the server and the client.
- pending_lobbies – a dictionary with the keys being the lobby id and the values being dictionaries. The inner dictionaries will have keys as team names plus a key for host and a key for all_players. The values will be lists of all the usernames of the clients within that group e.g. team_1 has a list of all the users in team 1, all_players has a list of all the users in the pending lobby.
- lobbies – a dictionary with the keys being lobby id and the values being dictionaries. These dictionaries have keys as users in the lobby and values as a list of the characters the user has chosen.

When a user first enters multiplayer, they have to create a username. The server checks if the username has been taken. If it has not been taken, the server creates a new info file for the user in the online file system and the user is sent a success message that the username was valid. The username is saved locally in an info file so that the device knows the user's username.

Since mobile connections are often unreliable, there will be a system for reconnecting if a connection is lost. The system will stop attempting to reconnect after a certain amount of time.

Join

When a user presses the join button, a message is sent to the server. The server then sends back a list of all the pending lobbies along with the host's username and the number of players in the lobby. The user's device then switches to the join screen where these lobbies are displayed in a table. Lobbies hosted by friends will be automatically put at the top of the table. Above the table will be a refresh button. When this is pressed, a message is sent to the server and the server resends an updated list of all the pending lobbies.

Next to each lobby row will be a join button. When this is pressed, a message is sent to the server. The server then sends a message to the host of the game containing the username of the player who wants to join. On the host's device, a popup appears asking if the host wants the user to join. If the host presses the yes button, a message is sent to the server. The server then notifies the user wanting to join and the user's device switches to the lobby screen.

Friends

Users will be able to add friends from the friends tab. When the user enters the username of a friend, a check is made that it is not already a friend and that it is not the user's own username, then a message is sent to the server. The server checks if the user exists and sends a message back to the user. If the friend exists, that player is now on the user's friends list. This list is stored in the same info file as the user's username.

Whenever the user enters the friends tab, a message is sent to the server containing a list of all the user's friends if the user has any friends. The server checks if these users are connected to the server right now and if so, checks what they are currently doing. The server then sends this information back to the user. The user's device creates a table of all the friends and their statuses which is displayed in the friends tab.

If a friend is in a lobby, a join button is placed next to his status and allows the user to request to join the lobby directly from the friends tab.

Lobby

Users can host lobbies by pressing the host button. This sends a message to the server. The server then adds a new key-value pair to the pending_lobbies dictionary. The user's device then switches to the lobby screen. This screen shows all the users in the lobby within each team. There will be buttons to join a team and a start button to start the game. The start button will be greyed out until there is at least one player on each team.

When the user presses start, a message is sent to the server. The server then notifies all the players in the lobby that the game has started, and they switch to character selection. Character selection will have a ten second timer now since it is online. If the timer runs out, characters are randomly chosen. Once characters have been chosen and confirmed, the chosen characters are sent to the server. The choices are added to the server's lobbies dictionary. When all of the users in the lobby have chosen their character and informed the server, the server starts the game and sends a message to the clients.

When a user joins a lobby, they also switch to the lobby screen. However, their screen doesn't have the start button since they are not the host.

For the lobby, at first, I decided to have the changing teams process working as follows:

1. User presses change teams button.
2. User changes teams if there is a spare slot, moves to centre if not.
3. Message is then sent to server.
4. Server passes the message on to all clients.
5. Clients change the player's team on their devices' lobbies.

There was a large oversight to this method though. What if there is one space on the opposite team and two people on the same team press switch teams at the same time? This would result in both players' devices saying that they are in that team since the teams changing applies on their devices first. This is very problematic.

To solve this, I decided to have teams changing done server-side so there would be no syncing errors. So now, the process will be:

1. User presses change teams button.
2. Message is sent to server.
3. Server checks if there is a spare slot and changes player's team, moves player to centre if no spare slot.
4. Server sends the new teams layout to each client.
5. Clients update the lobby on their device.

The teams layout is stored in the pending_lobbies dictionary.

Downloading and Uploading Characters

There were two different options for saving characters:

- When players are in the lobby, all the players download all the images from each other. Would cause a massive slowdown for everyone.
- When a user adds a friend, they download all of that friend's images. This means that user will only be able to see their friends' characters. This is ok though because those are the only ones that people really care about.

The second option made a lot more sense, so I decided to go with that one.

However, if the friend creates a new image, the user won't receive it. The solution to this is that whenever a user connects to the server, the user sends the names of all of their custom characters (the names are hashed so they should be unique to that character). The server checks if these characters are stored in the online file system. The server sends a message containing the names of all the images not already saved in the online file system. The user then uploads those characters only. The server stores the names of all the characters from that user in the info file created when the user first created a username.

In the same way, the user still needs to download any new characters their friends have created. To do that, the user has to send a list of friends and characters already downloaded from friends to the server whenever the user connects to the server. The server knows which characters each of the friends have uploaded because of the info file. The server then uploads all the characters that the user didn't already have.

Now whenever a user plays with a friend and the friend picks one of their custom characters, the user can see that custom character.

Test Plan

Test No.	Objective No.	Description	Expected Outcome
1	1	Run the game on Windows, Linux and Android.	Should work on all of the platforms with no issues.
2	2	Let the client use the user interface and get feedback.	The client should find the UI intuitive.
3	2	Test the time it takes to transition between screens.	Should be under a second.
4	3	Test drawing new faces for multiple characters.	The new faces should be applied to the characters.
5	3	Let the client use the canvas and get feedback.	The client should find the canvas responsive.
6	4	Test loading a custom character.	Character should be able to be loaded and reused.
7	4	Test deleting a custom character.	Once deleted, character should be completely gone from the game.
8	5	Try previewing a new character.	Old character with new character's head should appear.
9	5	Test saving a new character.	Save should take less than 10 seconds.
10	6	Write a program to test the hash that tests the speed and number of collisions.	One hash should take half a second, there should be very few collisions.

11	7	Play a game with custom characters and pre-made characters.	The custom characters should work with no inconsistencies.
12	8	Observe the frame rate and game speed throughout a game.	The frame rate should be changing but the game speed should be constant.
13	9	Run the game on my phone (Google Pixel XL) and observe the frame rate.	Should run at between 30 and 60 fps.
14	10	Let the client play the game and get feedback.	The client should enjoy the game.
15	11	Complete all the other tests.	No major bugs should occur during any of the tests.
16	12	Let the client play the game and get feedback.	The client should find the AI fair to play against.
17	13	Connect to the server and create a username.	If the username is not taken, the server should accept it.
18	14	Add a friend.	The friends should now be in the friend's list.
19	14	Check the friend's list.	It should show all of the user's friends and their statuses.
20	15	Join a friend from the friend's list.	A request should pop up on the host of the game's screen.

21	16	See all the available lobbies.	A table of lobbies should be generated.
22	17	Host a lobby.	The user should be in a lobby on their own until someone joins.
23	18	Use a second instance of the game to request to join a hosted lobby.	The host should be able to accept or deny the request.
24	19	Change teams within the lobby screen.	The teams should be updated on the screen of everyone in the lobby.
25	20	Play a game with a friend who made a custom character.	The custom character should appear on both player's screens.
26	21	Start a match with 6 instances of the game.	The game should run normally with all six players being able to play.
27	22	Compare frame rates in online and offline.	They should be about the same.
28	23	Disconnect a device while connected to the server.	The server should notify any players interacting with the disconnected player.
29	24	Disconnect and reconnect quickly.	The player should be able to resume playing.

Technical Solution

Techniques used

Technique	Page numbers
Candidate written classes (OOP)	p70, 87, 93, 100, 106, 116
Stacks	p75
Use of hashing techniques	p83
Use of recursion	p74, 94
Time scheduling	p86
Complex client server model	p105-122
3D arrays	p78-82
Complex data structures	p80
Implementing complex mathematical processes	p101

Character Creation

Canvas

Here is the class I created for the canvas. It inherits the Widget class from kivy, allowing it to be treated like a widget. This means that it can be placed on the screen and have some attributes such as size and position.

The width of the canvas is decided based on the size of the screen. Note that the actual size of the canvas set as self.size is different to the local variable width. This is because I had to first round the width to the nearest rect_width. This may seem pointless because rect_width is just width/30 so width should always be rounded to the nearest rect_width (i.e. be a multiple of rect_width). However, rect_width is the integer result of width/30 because it needs to be an integer for later calculations. This means that the width has to be updated to be a multiple of this rect_width. This is where the function specific_rounding comes in (this will be used later as well).

The canvas is made up of initially transparent rectangles for each square in the 30x30 grid. Each rectangle has the height and width as self.rect_width which is equal to the canvas size divided by 30. The colour of each rectangle object is saved in an ordered dictionary called self.rects_colours (it is ordered to make it convenient to turn into an image array later). The colour is saved as the value and the key is a tuple with the x and y position of the square.

The grid is generated by creating black lines. The rectangles are then placed in front of the grid.

```
# Function used to round to the nearest something that's not a power of 10.
def specific_rounding(value, rounding_number, subtractor=0):
    # The rounding is done to the nearest rounding_number so if rounding_number = 4, rounding is done to the nearest 4.
    # This is because round(x / 4) * 4 rounds x the nearest 4. This is used in the program to round the touch position
    # to the nearest rectangle to be painted on. The subtractor is needed because the rounding needs to be done relative
    # to the position from the start of the widget but the value given is relative to the position of the screen.
    return round((value - subtractor) / rounding_number) * rounding_number + subtractor
```

```
# This is the canvas that the player draws on. Inherits the widget class from kivy.
class PaintWidget(Widget):
    def __init__(self):
        super().__init__()
        self.register_event_type('on_draw')
        width = min(Window.width*0.45, Window.height*0.9)
        self.pos = ((Window.width-width)/2, (Window.height-width)/2)  # Set the widget's position based on its size.
        self.rect_width = int(width / 30)
        self.rounding_number = float(self.rect_width)  # The number things will be rounded to.
        self.size = (specific_rounding(width, self.rounding_number),
                     specific_rounding(width, self.rounding_number))
        self.tool = "pencil"  # Current tool in use is pencil.
        self.rects_colors = OrderedDict()  # An ordered dictionary containing the colours of each dot.
        self.undo_list = deque(maxlen=99)  # A list with a limit of 99 elements.
        self.redo_list = deque(maxlen=99)
        self.pencil_state = "pencil"  # Determines whether pencil button is in pencil or rubber mode.
        self.color = (1, 1, 1, 1)  # Initial colour is black.
        self.pencil_size = 1  # Number of rects drawn by a single tap of the pencil.
        with self.canvas:
            Rectangle(pos=self.pos, size=self.size)  # Create a background.
            Color(0, 0, 0)  # Set line color as black.
            # Make a grid.
            for i in range(int(self.y + self.rect_width), int(self.y + self.height), self.rect_width):
                Line(points=[self.x, i, self.x + self.width, i])
            for i in range(int(self.x + self.rect_width), int(self.x + self.width), self.rect_width):
                Line(points=[i, self.y, i, self.y + self.height])
            # Create all the dots and make them transparent.
            for pos_y in range(int(self.y), int(self.y + self.height), self.rect_width):
```

```
            for pos_x in range(int(self.x), int(self.x + self.width), self.rect_width):
                # Ensure that the grid being made is 30x30.
                if pos_y == int(self.y) + self.rect_width * 30 or pos_x == int(self.x) + self.rect_width * 30:
                    continue
                self.rects_colors[(pos_x, pos_y)] = Color(0, 0, 0, 0)
                Rectangle(pos=(pos_x, pos_y), size=(self.rect_width, self.rect_width))
        self.prev_touch = None
```

Within the PaintWidget class, the method for handling a user's touch is on_touch_down. This method first makes sure the canvas is correctly sized.

Next, it uses specific_rounding to round the position of the touch to the nearest rect_width. In doing so, it finds the coordinates of the closest rectangle to the touch. Note the subtractor parameter in the specific_rounding call. This is used because the rectangles only start at the start of the widget, but the initial coordinates used are relative to the entire screen.

The method then checks if the touch was within the canvas. It then checks if the intended action would be redundant (either the tool is pencil or fill and rectangle is already the chosen colour, or the tool is rubber and the rectangle is already rubbed out). If none of these apply, the method continues.

Now, the redo stack is cleared (because redo can't be done after a new action has been done). The undo and redo numbers are then updated (these numbers are displayed next to the undo and redo buttons and just show how many undos and redos can be done). The method then calls the appropriate function for the tool selected with the correct parameters. Both pencil and rubber actually call the same method just with a different parameter because the two actions are so similar.

```
    def on_touch_down(self, touch):  # This method is called when the screen is touched.
        self.size = (round(Window.width * 0.4, -1), round(Window.height * 0.8, -1))  # Resets the size of the canvas.
        # Get the rounded pos
        pos = (specific_rounding(touch.x-self.rect_width/2.0, self.rounding_number, subtractor=self.x),
               specific_rounding(touch.y-self.rect_width/2.0, self.rounding_number, subtractor=self.y))
        if pos in self.rects_colors.keys():  # Ensure that the click was in the grid.
```

```
            if self.rects_colors[pos].rgba != self.color or self.tool == "rubber":
                if self.tool == "rubber" and self.rects_colors[pos].a == 0:
                    return
                self.redo_list.clear()
                self.parent.parent.update_undo_nums()
                if self.tool == "pencil":
                    self.pencil(pos)
                elif self.tool == "rubber":
                    self.pencil(pos, erase=True)
                else:
                    self.undo_list.append(self.copy_rects_colors())
                    self.fill(pos, self.rects_colors[pos].rgba)
```

Additionally, a method is required for dealing with drags. The method that does this is on_touch_move. One issue with kivy that I noticed when creating this method is that its drag detection struggles with quick movements. What happens is that there is a large gap between two touches. This results in large blank spaces on the canvas between two squares drawn in. To solve this, I have to simulate touches between the previous touch and the current touch.

The first check I make is if this is the first method call of this drag. If it is, then I don't have to simulate any in-between touches since there haven't been any yet. I then make sure this touch is of the same drag as the prev_touch attribute. Next, I use Pythagoras theorem to calculate the distance between the current touch and the previous touch. I then have to make sure that the distance is greater that rect_width / 2 because otherwise the touches would be close enough anyway so there would be no need for simulated touches. I then have to find the angle between the two touches using arctan. Next, I get the number of simulated touches needed by dividing the distance by rect_width / 2. I then use trigonometry to work out how much I have to increment the x and y positions of each touch for each simulated touch. Now, I just need to create new touch objects with the attributes x and y set to the intended x and y of the simulated touch. Using this new_touch, I can call on_touch_down with new_touch as the parameter. After this, I call on_touch_down with the real touch and then save a copy of this touch as self.prev_touch.

```
    def on_touch_move(self, touch):  # This method is called when the player holds down and moves their finger.
        # Kivy's drag detection struggles with quick movements. What happens is that there is a large gap between two
        # touches. This results in large blank spaces on the canvas between two squares drawn in. I have to simulate
        # touches between the previous touch and the current touch.
        if self.prev_touch:
            if self.prev_touch.opos == touch.opos:  # If the previous touch was part of the same drag as this touch.
                # Find the distance between the previous touch and the current touch using Pythagoras.
                dist = math.sqrt((touch.x-self.prev_touch.x)**2 + (touch.y-self.prev_touch.y)**2)
                # The simulated touches will be self.rect_width/2 apart.
                # If the distance is less than this, we don't need simulated touches.
                if dist > self.rect_width / 2:
                    # Find the angle between the previous touch and the current touch.
                    angle = math.atan2((touch.y-self.prev_touch.y), (touch.x-self.prev_touch.x))
                    count = int(dist // (self.rect_width/2))  # The number of simulated touches needed.
                    x_increment = self.rect_width/2 * math.cos(angle)
                    y_increment = self.rect_width/2 * math.sin(angle)
                    for i in range(count):
                        # self.on_touch_down takes a touch object as a parameter and uses the x and y properties of the
                        # touch object. We need to create a fake object with these properties to call the function.
                        new_touch = type('new_touch', (object,), {'x': self.prev_touch.x + x_increment * i,
                                                                   'y': self.prev_touch.y + y_increment * i})()
                        self.on_touch_down(new_touch)
            self.on_touch_down(touch)
            self.prev_touch = copy(touch)  # Have to copy it so it doesn't get overwritten by the next call.
```

For the fill, I created a recursive method. First, the method has to check if the current square is valid to be filled. For this, init_color must not be the same as color (this would mean that the square is already the right colour) and the colour of square must be the same as init_colour (otherwise the square is not part of the same section as the first square). Next, the method has to fill the square and call the function again but with the squares to the left, right, above and below the current square.

```
    def fill(self, pos, init_color, color=None):
        if not color:
            color = self.color
        if pos not in self.rects_colors.keys():  # If the current pos is out of bounds.
            return
        if init_color == color:  # If the rectangle is already the desired colour.
            return
        # If the rectangle is a different colour to the first rectangle that was clicked on.
        if self.rects_colors[pos].rgba != init_color:
            return
        self.draw(pos, color)  # Fill in that rectangle.
        # Repeat the function with the next rectangle to the left, right, up and down.
        self.fill((pos[0]+self.rect_width, pos[1]), init_color, color=color)
        self.fill((pos[0]-self.rect_width, pos[1]), init_color, color=color)
        self.fill((pos[0], pos[1]+self.rect_width), init_color, color=color)
        self.fill((pos[0], pos[1]-self.rect_width), init_color, color=color)
```

For the undo and redo, some of their functionality is in other methods because they get updated when other changes are made. The undo and redo buttons are linked to two different stacks.

The stacks have a limit of 99 changes to save memory since people very rarely would need to undo or redo 99 changes, and at that point it would be quicker to restart the drawing.

When the undo button is pressed, the key-value pair of the square about to be reversed is copied and pushed into the redo stack from the canvas dictionary. Then the undo stack is popped, and the key-value pair is put back into canvas dictionary, reversing the change.

When the redo button is pressed, the key-value pair of the square about to be re-changed is copied and pushed into the undo stack from the canvas dictionary. Then the redo stack is popped, and the key-value pair is put back into canvas dictionary, redoing the change.

This functionality is similar enough to be combined into one method where the attributes are just changed depending on the situation. This is what is done here:

```
def undo(self, button, other_button, start_list, end_list):  # If button is undo, other_button is redo etc.
        if start_list:
            temp_rects_colors = start_list.pop()
            end_list.append({})
            for pos in temp_rects_colors:
                end_list[-1][pos] = self.rects_colors[pos].rgba
                self.rects_colors[pos].rgba = temp_rects_colors[pos]
            other_button.color = (1, 1, 1, 1)
        if not start_list:
            button.color = (.7, .7, 0, 1)
        self.parent.parent.update_undo_nums()
```

Save

Once the user is happy with their created character, a lot of processing has to be done on their creation to turn it into an actual playable character. The first function make_dict, is used to convert the file structure of the character into a dictionary so the files within can be easily accessed. The dictionary contains lists of all the pathnames of the frames of a specific action.

```
from os import walk, makedirs
import cv2
import numpy as np
import pickle
from pathlib import Path
from shutil import copy
import base64


# A function to create a dictionary for the character specified with the keys being the actions of the character and the
# values being lists of the pathnames of each image within that state.
def make_dict(character_name, path="Images"):
    image_dict = {}  # This will be a dictionary containing lists of pathnames
    for subdir, dirs, files in walk(path + "/" + character_name):  # Finds each file in the character name folder
        temp_subdir = subdir.split("/")
        # os.walk uses backslashes but this doesnt work on android so these lines turn everything into forward slashes
        temp_subdir.extend(temp_subdir[1].split("\\"))
        del (temp_subdir[1])
        temp_subdir.remove(path)
        temp_subdir.remove(character_name)
        temp_subdir = "".join(temp_subdir)  # Theses 4 lines are for formatting the path into just the subdir name
        if not temp_subdir:
            continue
        image_dict[temp_subdir] = list()
        for i in range(len(files)):
            # Add the file path to the dictionary
            image_dict[temp_subdir].append("Images/" + character_name + "/" + temp_subdir + "/" + str(i + 1) + ".png")
    return image_dict
```

The function make_new_character_directories is the opposite of make_dict. It takes in a dictionary similar to the one created in make_dict except instead of pathnames, image_arrays of the frames are stored. The function then crceates the new character file structure and save all the images within.

```
# Basically the opposite of the make_dict function because this uses a dictionary to create folders with images saved.
def make_new_character_directories(character_name, image_dict, drawing, old_character_name):
    char_path = "Images/" + character_name + "/"
    makedirs(char_path)
    saveable_drawing = {}
    for pos in drawing:
        # Turn the values into tuples containing rgba rather than kivy Color objects so it can be pickled
        if drawing[pos].rgba != [0, 0, 0, 0]:
            saveable_drawing[pos] = drawing[pos].rgba
    copy("Images/" + old_character_name + "/info.txt", char_path)
    with open(char_path + "rects_colors.pickle", "wb") as output_file:
        pickle.dump(saveable_drawing, output_file)  # Save the drawing so it can be edited later by the player
    for key in image_dict:
        key_path = char_path + key + "/"
        makedirs(key_path)
        for i, image in enumerate(image_dict[key]):
            save_image(image, key_path + str(i + 1) + ".png")
            if key == "idle" and i == 0:  # This is for making a cropped preview image for the character
                cropped_image = crop_image(image)
                save_image(cv2.resize(cropped_image, (cropped_image.shape[1] * 5, cropped_image.shape[0] * 5),
                                      interpolation=cv2.INTER_NEAREST), char_path + "preview.png")
```

To change the head of the character to the new drawing by the user, image arrays must be used. Image arrays are data structures that represent an image. They are formatted as 3-dimensional arrays. The outer layer of subarrays represents each row of the image. The next layer represents each pixel within the row. Each pixel is represented by four values, the RGBA values of the pixel. To access a pixel from the image array, you can use indexing. For example, if you want to print the value of a pixel with x-position as 30 and y -position as 50, the code is print(image_array[50, 30]). Note that the y-coordinate comes first. This is because the first layer of subarrays represents each row.

drawing_to_image converts the drawing dictionary created by the user into a 1-dimensional list at first (this is why an ordered dictionary was needed). The list is then converted to an array which is then reshaped into a 30x30 3-dimensional array with 4 channels (RGBA).

```
def drawing_to_image(drawing):  # Turns the initial drawing in the format of a dict with pos as keys to image
    temp_list = []
    # Make the dictionary into a 1D list and multiply each colour value by 255 because kivy uses a 0-1 range for rgba
    for key in drawing:
        temp_list.append(list(map(lambda x: x * 255, drawing[key].rgba)))
    im_arr = np.array(temp_list)  # Make the 1D list into a 1D image array
    im_arr = np.flipud(im_arr.reshape(30, 30, 4))  # Reshape the image array to become 3D
    return im_arr
```

Cropping is used for creating a preview image as well as for hashing. To crop the image, the image array has to be flattened to find empty pixels. The first and last rows/columns that aren't completely empty are the crop box. Anything outside of this crop box is removed from the image.

```
def crop_image(im_arr, hashing=False):  # Only want to crop width since this is constant except for when hashing
    # Turn the values of the rgba axis into a combination of all of them, so an empty pixel will have a value of 0.
    image_data = im_arr.max(axis=2)
    # Find columns where all pixels has a value > 0, so not an empty column or row
    non_empty_columns = np.where(image_data.max(axis=0) > 0)[0]
    # Create the dimensions containing the first and last non_empty row
    crop_box = (min(non_empty_columns), max(non_empty_columns))
    im_arr = im_arr[:, crop_box[0]:crop_box[1] + 1, :]  # Crop the image array
    if hashing:  # Crop rows as well
        non_empty_rows = np.where(image_data.max(axis=1) > 0)[0]
        crop_box = (min(non_empty_rows), max(non_empty_rows))
        im_arr = im_arr[crop_box[0]:crop_box[1] + 1, :, :]
    return im_arr
```

To convert the original character images into image arrays, I use the function imread() from the library opencv. However, I have to add a flag (cv2.IMREAD_UNCHANGED) as a parameter to ensure that the alpha channel is kept. I then have to flip the order of the first three elements of the arrays for each pixel because opencv uses BGRA rather than RGBA e.g. a yellow pixel is initially [0, 255, 255, 255] but I want it as [255, 255, 0, 255].

```
def make_image_array(pathname):
    # cv2.imread creates an image array from a pathname. The -1 is to make it keep alpha channel.
    # The slicing afterwards is there because cv2 use bgra rather than rgba so this is converting it to rgba.
    # IMPORTANT: The image array is in the format [y][x][colour] not [x][y][colour] like expected.
    # x and y are integers and color is a list of 4 integers up to 255.
    im_arr = cv2.imread(pathname, -1)[:, :, [2, 1, 0, 3]]
    return im_arr
```

We need to know where the head is to replace it. To do this, I created an info text file for each character which contains the dimensions of the head relative to the top-left-most pixel of the head as a list e.g. if the head is 100x100 pixels with 20 pixels to the left of the top-left-most pixel, the dimensions will be saved as [0, 100, -20, 80]. This value is passed has the head_dimensions parameter. The dimensions are in the form [start_y, end_y, start_x, end_x] where start_y and x are the y and x positions relative to the top-left-most pixel where the head starts and end_y and x are the same but for where the head ends. This system works for all the character images because the size and rotation of the head stays constant in all animations but the position of the head changes.

To find the top-left-most pixel, I have to iterate through each row of the image array until I find a pixel that is not transparent. I then use the head dimensions from the info text file to create a start and end point of the head e.g. if the top-left-most pixel is on row 1 and column 320 and the head dimensions are [0, 100, -20, 80], the start row is 1 and the end row is 101 and the start column is 300 and the end column is 400. A list of [start row, end row, start column, end column] is returned.

```
def image_search(im_arr, head_dimensions):
    # This function finds the uppermost pixel and then calculates using head dimensions, where the head is.
    image_y, image_x = im_arr.shape[:2]
    for y in range(-head_dimensions[0], image_y - head_dimensions[1]):
        for x in range(-head_dimensions[2], image_x - head_dimensions[3]):
            test = (im_arr[y, x] == (0, 0, 0, 255))
            if test.all():
                return [y + head_dimensions[0], y + head_dimensions[1], x + head_dimensions[2], x + head_dimensions[3]]
```

The next process is to allow the users to change the colours of the character's clothes and skin. To do this, the program has to know the positions of all the pixels that are in different sections of the character. The way to do this is by having an info text file for each character which gives the colours of each section. This allows the program to find all the pixels of that colour and then change it to the new user selected colour.

However, this is extremely inefficient and would take the program a very long time since it would have to search every single pixel in every image and check its colour.

The solution to this is creating a large data structure in advance and then serialising it so it can be reused. The data structure is a dictionary with keys as pathnames of the individual frames and values as a dictionary. This dictionary has keys as each section (i.e. shoes, top, bottoms, skin) and values as another dictionary. This third dictionary has keys as the column containing the colour and the values as a list of the rows which are the certain colour. However, this can be made more efficient. Instead of putting all the rows with the certain colour, putting only the first and last row in a sequence of rows of the certain colour would massively reduce the number of elements in the data structure. For example, instead of [120, 121, 122, 123, 124, 220, 221, 222, 223], the list would be [120, 124, 220, 223]. This dictionary is created by the find_sections function.

An example key-value pair from the dictionary would be:

{“Images/char1/idle/1.png”: {“shoes”: {20: [50, 90, 100, 150], …}, …}, …}

As you can see, this dictionary is massive and would take very long to create if it was made every time the user saved a character. Generating the dictionary in advance and saving it for reuse is massively more efficient.

To serialise the data structure, I can use the library pickle. This will save the data structure as a file which I can then load into a variable at any time. When the user picks a new colour for the section, the program turns all the pixels given in the data structure that are in that section into the new colour.

```
def find_sections(character_name):  # This is only used for new characters upon creation, never called by the program
    character = make_dict(character_name)
    info = get_info(character_name)
    head_dimensions = info["head_dimensions"]
    section_locations = {}
    for key in character:  # Go through each animation in the character directory
        for file in character[key]:  # Go through each image in the animation
            section_locations[file] = {}
            # Remove the head, np.zeros creates an empty array
            im_arr = combine_images(file, head_dimensions, np.zeros((head_dimensions[1] - head_dimensions[0],
                                                                     head_dimensions[3] - head_dimensions[2], 4)))
```

```
            for section in info:  # Go through each section
                if section.endswith("colour"):  # If this is a section_colour and not other info like head_dimensions
                    section_locations[file][section] = {}  # Create a new dictionary for this section
                    color = info[section]  # Get the color of that section
                    for x in range(0, im_arr.shape[1]):  # Go through each column of pixels in the image
                        section_locations[file][section][x] = list()  # Create a list for that column
                        first_pixel = True  # This is for the first pixel of each column of the section color.
                        for y in range(0, im_arr.shape[0]):  # Go through each pixel in the column
                            test = (im_arr[y, x] == color)
                            if test.all() and first_pixel:  # If this pixel is the right colour and is the first
                                section_locations[file][section][x].append(y)  # Add it to the list for that column
                                first_pixel = False
                            elif not test.all() and not first_pixel:
                                section_locations[file][section][x].append(y)  # Add it to the list for that column
                                first_pixel = True
                        if not section_locations[file][section][x]:
                            del section_locations[file][section][x]
    with open("Images/" + character_name + "/section_locations.pickle", "wb") as output_file:
        pickle.dump(section_locations, output_file)  # Serialise the dictionary and save it
```

A dictionary called section_colours will contain the new chosen colours by the user. When the user picks a new colour for a section, the value with the key of that section will be set as the chosen colour e.g.

{“shoes”:[0, 255, 0, 255],“skin”:[255, 0, 0, 255],“top”:[0, 0, 255, 255],“bottoms:[0, 0, 0, 255]}

would mean that the user chose green shoes, red skin, blue top and black bottoms.

The swap_colors function goes through each section and applies the changes using section_colors and section_locations.

```
def swap_colors(im_arr, section_colors, section_locations, original_pathname):
    section_locations = section_locations[original_pathname]  # Get the section locations for this file
    for section in section_locations:  # Go through each section
        if section_colors[section]:  # If the section colors list contains a value for this section
            for x in section_locations[section]:  # Go through each column
                for i in range(0, len(section_locations[section][x]), 2):  # Get every odd value
                    # Get every value between the first and last in a column
                    for y in range(section_locations[section][x][i], section_locations[section][x][i + 1]):
```

```
                # Set the pixel to the new colour
                im_arr[y, x] = list(map(lambda num: num * 255, section_colors[section]))
    return im_arr
```

The user-created characters will be saved on the local device and the online file system so the characters can be used offline as well as being easily shared online without having to upload the images every time the character is used. When a user adds a friend, the user downloads the friend's characters from the file system if they don't already have the character saved. The characters will need a hash code to determine if the player has the most updated version of the character.

Usually, image hashes are used to find similarities between images and try to ignore artefacts and other results of compression. However, the image hash I need for the program should work like a normal hash. It should result in a different hash even if just one pixel is different.

Initially, I planned to make the hash just work on the full image. However, a more efficient method would be to hash just the new head. This wouldn't consider the base body and the colours of the clothes and skin though. To solve this, I can add the name of the character whose body is being used and I can hash the section_colours dictionary and add that to the end of the overall hash.

To hash the head, I first crop it to make the hashing quicker and reduces the similarities between images.

Next, I use NumPy's tobytes() method to convert the image array to bytes. I then use python's inbuilt int.from_bytes() function to convert the bytes to an integer. This allows me to manipulate it. To hash something, the original value has to be diffused. This means that two similar values result in very different hashes. There are many methods for diffusion. I will use bit shifts, XORs, multiplication and modulo as these are very quick processes and are good at diffusing. I will do the first three processes multiple times to improve the diffusion.

First, I will bit shift the integer to the left by a constant. I will then XOR the original image with the bit shifted image. Finally, I will multiply the integer by a constant. These processes will be repeated three times.

Next, the most important part of the hash is the modulo. This is essential for reducing the size of the integer to the table size. At this point, the integer is massive. The hash will be used for directory names, so the character limit is 248 on Windows and 255 on Linux. However, we don't need the hash to be that long to prevent collisions and the larger the number we modulo by the slower the hash will be. A more reasonable size would be about 50 characters. The final hash won't be an integer (this would make the table size 10^50 if we wanted 50 characters). Instead the

hash will be converted to base 64. This means that the table size can be 64^50 which is approximately 2x10^90. When using modulo for hashing we want to modulo by a coprime of the initial number. This is to deal with any patterns that may occur in the number which would result in collisions otherwise. Fortunately, it isn't hard to find a prime close to 64^50 because 2^n - 1 is always prime (Mersenne primes). Therefore, I will modulo the integer by 64^50 – 1.

Finally, we add the section_colours and the name of the original character to the end of the hash. The new character will be saved under a directory with its name being the hash. This makes checking for duplicates and checking if the user already has the image very easy.

```
def hash_image(image, section_colors, name):
    # Cropping the image both makes the hashing faster and decreases the similarities between images
    try:
        image = crop_image(image, hashing=True)
        hash_output = int.from_bytes(image.tobytes(),
                                     "little")  # Convert the image array to bytes and then convert to int
        for i in range(3, 5):
            # XOR hash with the itself bit shifted to the left by i and then multiply the hash by i
            hash_output ^= hash_output << i
            hash_output *= i
        # At this point hash_output is a huge number (approximately 10^70000). We need to reduce the size of
        # hash_output to the hash table size. The best way to do this is with modulo. The hash will be used for
        # directory names so the character limit is 248 on Windows and 255 on Linux. However, we don't need the hash
        # to be that long to prevent collisions and the larger the number we modulo by the slower the hash will be. A
        # more reasonable size would be about 50 characters. The final hash won't be an integer (this would make the
        # table size 10^50 if we wanted 50 characters). Instead the hash will be converted to base 64. This means
        # that the table size can be 64^50 which is approximately 2x10^90. When using modulo for hashing we want to
        # modulo by a coprime of the initial number. This is to deal with any patterns that may occur in the number
        # which would result in collisions otherwise. Fortunately it isn't hard to find a prime close to 64^50
        # because 2^n - 1 is always prime (Mersenne primes).
        hash_output %= 64 ** 50 - 1
        byte_length = (hash_output.bit_length() + 7) // 8  # This is needed for converting an integer to bytes
        hash_output = hash_output.to_bytes(byte_length, "little")
    except ValueError:  # This means that the image is empty.
        hash_output = b""
    # The previous section of the hash only dealt with the head of the character but now we have to take into account
```

```
    # the colours of the character's skin, clothes and shoes.
    for section in section_colors.values():
        if section:  # If the user hasn't picked a new colour, section will be None
            # Round each value to 2 d.p. and concatenate as a string
            section = "".join(map(lambda x: str(round(x, 2)), section[:-1]))
        else:
            section = "0"
        hash_output += section.encode("utf-8")
    hash_output += name.encode("utf-8")  # name is the name of the character's body which the new character is based on
    # '/' is not allowed in filenames so replace with '_'
    return base64.b64encode(hash_output).decode("utf-8").replace("/", "_")
```

Now, all the processes are combined to create the new character.

```
def main(old_character_name, replacement_drawing, section_colors, section_locations, preview=False):
    new_character_name = None
    old_character = make_dict(old_character_name)
    info = get_info(old_character_name)
    head_dimensions = info["head_dimensions"]
    replacement_image = drawing_to_image(replacement_drawing)
    # Resize the image to the head dimensions
    replacement_im_array = cv2.resize(replacement_image, (head_dimensions[3] - head_dimensions[2],
                                                          head_dimensions[1] - head_dimensions[0]),
                                      # This gets rid of interpolation which would make the image blurry
                                      interpolation=cv2.INTER_NEAREST)
    if preview:
        combined_image = combine_images(old_character["idle"][0], head_dimensions, replacement_im_array)
        color_swapped_image = swap_colors(combined_image, section_colors, section_locations,
                                          original_pathname=old_character["idle"][0])
        cropped_image = crop_image(color_swapped_image)
        # Resize image so it can be seen at full size on the screen
        resized_image = cv2.resize(cropped_image, (cropped_image.shape[1] * 5, cropped_image.shape[0] * 5),
                                   interpolation=cv2.INTER_NEAREST)
        save_image(resized_image, "Images/preview_image.png")
        return
    final_images = {}  # This will be a dictionary of lists containing images
    for key in old_character:
```

```
        final_images[key] = []
        for filename in old_character[key]:
            combined_image = combine_images(filename, head_dimensions, replacement_im_array)
            combined_image = swap_colors(combined_image, section_colors, section_locations, original_pathname=filename)
            final_images[key].append(combined_image)
            if key == "idle" and len(final_images[key]) == 1:
                new_character_name = hash_image(replacement_image, section_colors, old_character_name)
                if Path("Images/" + new_character_name).is_dir():  # If this character already exists
                    return
    make_new_character_directories(new_character_name, final_images, replacement_drawing, old_character_name)
    return
```

Gameplay

Variable Frame Rate

To get maximum performance on all device, a variable frame rate has to be implemented. In Kivy, actions can be put on a schedule where they repeat every x seconds. I set the game schedule to repeat every 1/fps seconds. I then have to change the fps to the maximum amount that doesn't result in slowdown (caused if the device can't process frames in time). Kivy's schedule passes a time parameter every tick. This parameter has the value of the amount of time since the previous tick. If the time is greater than 1/fps, slowdown has occurred.

To get the right fps, I have to maximise fps until slowdown occurs. What I do is increase the fps by 1 every tick where the time is the same as 1/fps (within a certain threshold). I decrease the fps by 1 every tick where the time is greater than 1/fps. The reason why I constantly change the fps every tick rather than find a right value and keep it at that is because the speed of the processing changes often. A background application could start using slightly more or less resources at any time so a frame rate that adapts quickly is essential. Additionally, the more often the fps adapts, the more accurate it is. The fps attribute is then used for any movement to adapt the speed to the framerate.

```
# Function for changing the frame rate of the game.
def variable_frame_rate(self, time=None):
    # If the device was able to complete the previous tick in the correct amount of time, increase the fps.
    if math.isclose(time, 1 / self.fps, rel_tol=0.1):
        self.fps += 1
    elif self.fps > 1:  # Otherwise decrease the fps as long as it is above 1.
        self.fps -= 1
    self.main_schedule.timeout = 1 / self.fps
    self.run_schedule.timeout = 1 / self.fps
```

Characters

Here is the character class constructor:

```
class Character(Image):
    def __init__(self, image_dict, ball, team):
        super().__init__()
        self.image_dict = image_dict
        self.allow_stretch = True
        self.ball = ball
        self.unit_x = Window.width / 200  # Create a distance unit that is scaled to window size
        self.unit_y = Window.height / 100
        self.size_hint = (None, None)
        self.size = (self.unit_x*30, self.unit_y*30)
        # Boundaries of the pitch
        self.init_bounds = [[Window.width / 12, Window.width / 2.6], [Window.height / 7.8, Window.height / 1.8]]
        if team == 2:
            # Change the horizontal boundaries to the other side and reverse them.
            self.init_bounds[0] = list(map(lambda x: Window.width - self.width - x, self.init_bounds[0]))[::-1]
        # deepcopy is used because otherwise a change to self.bounds will change self.init_bounds.
        self.bounds = deepcopy(self.init_bounds)
        # Start at a random position.
        self.y = randint(int(self.bounds[1][0]), int(self.bounds[1][1]))
        self.x = randint(int(self.bounds[0][0]), int(self.bounds[0][1]))
        self.z = self.y
        self.team = team
        self.cache_images()
        self.angle = 0
        self.texture = self.image_dict["idle"][0].texture
```

```
        self.state = "idle"
        self.prev_state = "idle"
        self.state_is_continuous = True  # If true, the state is repeating e.g. idle, otherwise it is a one-time action.
        self.has_ball = False
        self.had_ball = False  # This is used to determine which player sends the ball information in multiplayer.
        self.running_jump = False
        self.teammates = []
        self.enemies = []
        self.state_frames = {}  # This is a dictionary of all the current frames of each state.
        self.animation_speeds = {}
        for state in self.image_dict:
            self.state_frames[state] = 0
            self.animation_speeds[state] = 1
        self.animation_speeds["idle"] = 0.5
        self.animation_speeds["idle_ball"] = 0.5
        self.animation_speeds["pick_up"] = 0.5
        self.throw = None
        self.health = 100
        self.dead = False
        self.selected = False
        with self.canvas:
            Color(0, 0, 0, 1)
            self.base_bar = Rectangle()  # Create a black bar beneath the healthbar to look like an outline.
            Color(0, 1, 0, 1)
            self.healthbar = Rectangle()
            self.selected_cursor_colour = Color(1, 0, 0, 0)
            self.selected_cursor = Triangle(points=[self.center_x, self.top + self.unit_y * 2,
                                                    self.center_x - 2 * self.unit_x, self.top + 5 * self.unit_y,
                                                    self.center_x + 2 * self.unit_x, self.top + 5 * self.unit_y])
```

Animations

The characters will be able to do several different actions (run, jump, catch, heavy throw, light throw). Each of these will need to be animated. Therefore, each frame of each action will have to have an image. These will be saved in the following way:

./Images/character_name/action/frame.png

For example: ./Images/dave /jump/2.png

For the program to quickly access the files, a dictionary is to be used with key as the name of the action and value being a list of the pathnames of each frame for the action.

The program has to animate the next frame at each update. There will be a dictionary with the key being the action and the value being the current frame. Each update, the current frame is incremented by one until it reaches its maximum, it then resets to 1. However, some actions need to be animated slower than other (i.e. two clock ticks per frame). To solve this, I will use an animation_speed dictionary with each action's speed as the increment rather than one.

```
def animate_next_frame(self):
    self.prev_state = self.state
    try:
        self.texture = self.image_dict[self.state][int(self.state_frames[self.state])].texture
        # Increment the frame by the animation speed.
        self.state_frames[self.state] += self.animation_speeds[self.state]
    except IndexError:  # The animation is complete.
        self.state_frames[self.state] = 0
        if not self.state_is_continuous:
            self.idle()
            self.state_is_continuous = True
    if self.state_is_continuous:
        self.idle()  # Automatically go back to idle.
```

Movement

The positions and movement of the characters has to be done in a 3D space so there needs to be x, y and z positions. However, since x and y are built-in to Kivy, the x, y and z axes are slightly different from the traditional way. x and z are the same as normal, but y is equal to z + height above ground. The characters will be moved by a virtual joystick on the screen. When the user moves the joystick pad, the angle is sent to the currently controlled character. The character then moves in the x direction by a constant * cos(angle) and the y and z directions by a constant * sin(angle). This will ensure the character's speed remains the same in all directions.

However, there needs to be boundaries for the movement since the character would otherwise run off of the screen. These boundaries will initially be the court. However, when the ball is outside of the court, the boundaries would have to expand so the players can get to the ball. When the ball returns to within the court, the boundaries change back to the court. To apply these boundaries to the characters, the movement of the character in the x direction is prevented if either cos(angle) < 0 and the x position < the left boundary or if cos(angle) > 0 and the x position > the right boundary. The same applies to z movement except sin is used over cos and the vertical boundaries are used.

```
# Called when the joystick is moved or when the AI is controlling the character.
def run(self, angle):
    if not self.running_jump:  # If the character is in a running jump they can't change direction.
        self.angle = angle
        if not self.state_is_continuous:  # Don't interrupt a one-time action e.g. catch.
            return
        self.state_is_continuous = True
        if self.has_ball:
            self.state = "run_ball"
        else:
            self.state = "run"
    # Ensure the character stays within the bounds.
    # If direction is right or x is greater than the left-most boundary.
    if math.cos(self.angle) > 0 or self.x > self.bounds[0][0]:
        # If direction is left or x is less than the right-most boundary.
```

```
        if math.cos(self.angle) < 0 or self.x < self.bounds[0][1]:
            # Use trig to get the amount x should be increased by.
            self.x += self.unit_x * math.cos(self.angle) * 30 / self.game.fps
    # If direction is up or x is greater than the bottom-most boundary.
    if math.sin(self.angle) > 0 or self.z > self.bounds[1][0]:
        # If direction is down or x is less than the top-most boundary.
        if math.sin(self.angle) < 0 or self.z < self.bounds[1][1]:
            # y and z are increase by the same amount because the player is staying on the ground.
            self.y += self.unit_y * math.sin(self.angle) * 30 / self.game.fps
            self.z += self.unit_y * math.sin(self.angle) * 30 / self.game.fps
```

Shadows

The characters and the ball also need shadows. To do this, I can create an ellipse with the size being based on the sprite's size and height above ground (when a character or ball is in mid-air, their shadow is larger).

```
def create_shadows(self):
    for sprite in self.sprites:
        if not hasattr(sprite, "shadow"):
            sprite.canvas.before.add(Color(0, 0, 0, .3))
            sprite.shadow = Ellipse()
            sprite.canvas.before.add(sprite.shadow)
        # The shadow's size is dependent on the sprites size and height above ground.
        sprite.shadow.size = (sprite.width/2+(sprite.y-sprite.z)/2, sprite.height/3+(sprite.y-sprite.z)/2)
        sprite.shadow.pos = (sprite.center_x-sprite.width/5-(sprite.y-sprite.z)/4,
                             sprite.z-sprite.height/6-(sprite.y-sprite.z)/4)
        try:
            if sprite.character:  # If the sprite is the ball.
                sprite.canvas.before.clear()
                del sprite.shadow
```

```
        except AttributeError:
            if sprite.health <= 0:
                sprite.canvas.before.clear()
                del sprite.shadow
```

Ball

Here is the ball class constructor:

```
# Class for the ball. Inherits the Image class from kivy.
class Ball(Image):
    def __init__(self, surface):
        super().__init__()
        self.size_hint = (None, None)
        self.pos = (Window.width / 2, Window.height / 2)  # Start in the middle
        self.z = Window.height / 3
        self.x_speed = randint(int(self.width * -0.5), int(self.width * .5))  # Have a random x_speed
        self.y_speed = 0
        self.z_speed = 0
        self.speed = 0
        # Get the friction constant from the surface
        self.surface_friction = float(surface[surface.index("[") + 1:surface.index("]")])
        self.completion_ticks = 10
        self.throwing_team = None  # This is the team that the ball was thrown by
        self.character = None  # The character holding the ball
        self.midair = True
        self.has_bounced = False
        self.rolling = False
        self.passed = False
        self.throw_type = None
        self.source = "Images/ball.png"
        self.size = (Window.width / 35, Window.height / 20)
        self.timer = time.time()
```

I decided to make the ball explode after being held for a long time to prevent time-wasting. When the ball explodes, it damages whoever is holding it and bounces towards the other team.

```
# If a team is time-wasting with the ball, the ball will explode, damaging them.
def hot_potato(self):
    if self.midair and not self.passed:  # Reset timer whenever the ball is thrown.
        self.timer = time.time()
    if time.time() - self.timer > 7:  # After 7 seconds, explode.
        self.explode()
        self.timer = time.time()  # Reset timer.

def explode(self):
    if self.character:
        self.character.do_throw(explosion=True)
    else:
        self.midair = True
        self.rolling = False
        self.throw(Window.width - self.center_x, randint(0, int(Window.height/3)), "light_throw")
        self.y_speed += self.height * 10 / self.game.fps  # Give a slight boost to the ball's vertical speed.
    if self.center_x < Window.width / 2:  # Set the throwing team as the team on the other side.
        self.throwing_team = 2
    else:
        self.throwing_team = 1
    # Animate the explosion
    self.explosion_animation()

# Recursive function for animating the explosion.
def explosion_animation(self, counter=1, explosion=None):
    if not explosion:
        with self.canvas:
            explosion = Rectangle(source="Explosion/1.png", width=self.width*2, height=self.height*2, pos=self.pos)
    if isfile("Explosion/" + str(counter) + ".png"):
        explosion.source = "Explosion/" + str(counter) + ".png"
        # Animate the next frame in 0.025 seconds.
        Clock.schedule_once(lambda clock: self.explosion_animation(counter=counter+1, explosion=explosion), .025)
```

```
    else:  # Once all the images have been animated, remove the explosion.
        self.canvas.remove(explosion)
```

Throw

For the throws, I first need to make a targeting system (I decided the throws should auto-lock on to the enemy the controlled character is aiming at rather than being completely controlled by the user since that would make it too hard to hit the opponent). This targeting system has to go through the list of targets (enemies for heavy and light throw and teammates for pass). It then finds the one whose angle with the controlled character is closest to the angle the user is aiming at with the joystick. This is now the target.

```
# Function finds the character (either enemy or teammate depending on the type of throw) whose angle from the
# thrower is closest to the current/most recent angle made on the joystick.
def find_target(self, throw_to_enemies=True):
    angle = math.pi * 2  # Start with just above the maximum angle.
    chosen_target = None
    if throw_to_enemies:
        target_list = self.enemies
    else:
        target_list = self.teammates
    for target in target_list:
        if target == self:
            continue
        dif_x = target.center_x - self.center_x
        dif_z = target.z - self.z
        target_angle = math.atan(dif_z / dif_x)
        # Have to sin and then asin self.angle to get the lowest version of the angle e.g. pi rad becomes 0 rad.
        angle_dif = abs(target_angle-math.asin(math.sin(self.angle)))
        if angle_dif < angle:
            angle = angle_dif
            chosen_target = target
```

```
    if chosen_target:
        return [chosen_target.center_x, chosen_target.z]
```

Now, the ball class's throw function is called with the parameters of the throw type (heavy, light or pass) and the position of the target. The throw function first calculates the distance from the ball to the target using Pythagoras' theorem. Next, the total speed value is decided based on the type of throw. The time the throw will take is then calculated by dividing distance by total speed. Now, the individual speeds in each direction is calculated. x and z speeds are calculated by dividing x and z distances respectively from the ball to the target by completion time. The y speed is just the z speed plus a constant to give a slight initial upwards trajectory. At each frame, the speeds are added to the ball's position. Additionally, at each frame, the ball's y speed is decreased to simulate gravity.

```
# Called every tick of the game
def update(self):
    self.x += self.x_speed
    self.z += self.z_speed
    self.y += self.y_speed
    self.speed = math.sqrt(
        (self.x_speed / self.width) ** 2 + (self.z_speed / self.height) ** 2) * 30 / self.game.fps  # Pythagoras
    self.bounce()
    if self.character:
        self.rolling = False
        self.passed = False
    if self.rolling:
        # x_speed and z_speed are multiplied by surface_friction every tick. With a high fps, this would make
        # them decrease more rapidly since there are more ticks. Therefore, surface friction must be
        # exponentiated by a constant over the fps.
        self.x_speed *= self.surface_friction ** (30 / self.game.fps)
        self.z_speed *= self.surface_friction ** (30 / self.game.fps)
    if self.midair:
        # Since y_speed is subtracted from rather than multiplied by, the fps only has to multiply the constant.
```

```
            self.y_speed -= (self.height * 30 / self.game.fps) / self.completion_ticks  # Gravity
        else:
            self.y_speed = self.z_speed  # Make the ball's vertical speed 0
            if not self.character:
                self.y = self.z  # Make the ball on the ground.
        self.hot_potato()


    # Called when a character throws the ball
    def throw(self, target_x, target_z, throw_type):
        distance = math.sqrt((target_x - self.x) ** 2 + (target_z - self.z) ** 2)  # Pythagoras theorem.
        total_speed = self.width * 60 / self.game.fps  # Set the speed as a constant.
        self.throw_type = throw_type
        if throw_type == "heavy_throw":  # Increase the speed if it's a heavy throw, decrease it if it's a pass.
            total_speed *= 1.5
        if self.passed:
            total_speed *= 0.5
        self.completion_ticks = distance / total_speed  # Work out how many ticks it will take to reach the target.
        # Work out the separate x and z components of the speed based on the x and z distances and the number of ticks.
        self.x_speed = (target_x - self.x) / self.completion_ticks
        self.z_speed = (target_z - self.z) / self.completion_ticks
        # self.height / 4 makes the trajectory look more realistic.
        self.y_speed = self.z_speed + (self.height / 4) * 30 / self.game.fps
        if throw_type == "light_throw":  # Make the trajectory more looping for light_throw.
            self.y_speed += (self.height / 4) * 30 / self.game.fps
```

Bounce

When the ball hits the ground, a player, or one of the edges of the screens, it bounces. When the ball hits the ground, the x and z speeds are multiplied by the surface friction constant which is determined by the stage the game is played on e.g. a snow stage has a friction constant of 0.3 but a grass stage has a friction constant of 0.7. The y speed becomes the z speed plus the vertical speed multiplied by the negative friction constant. If the ball bounces off a boundary or a player, the same applies but x or z speed are reversed rather than y speed.

When the ball collides with a character, the program checks if the character is in the catch action. If the character is, the character now has the ball. If not, the character is hit by the ball and loses health.

When the ball is vertically slow enough, it rolls rather than bounces. Its speeds are multiplied by surface friction every tick now, so the ball slows down quicker.

```
def bounce(self):
    if self.y < self.z < Window.height * .65:  # If the ball is touching the ground.
        # If the ball bounced last frame and is still going downwards i.e. the ball is very slow.
        if self.has_bounced and self.y_speed < self.z_speed:
            self.passed = False
            self.midair = False
            self.y_speed = self.z_speed
            self.y = self.z
            self.rolling = True
            return
        self.has_bounced = True
        self.throwing_team = None  # Once the ball has bounced, it doesn't do damage.
        self.x_speed *= self.surface_friction  # Doesn't have to be exponentiated by fps since it only happens once.
        self.z_speed *= self.surface_friction
        # Multiply the vertical speed by friction and then take it away from the z_speed.
        self.y_speed = self.z_speed - self.surface_friction * (self.y_speed - self.z_speed)
        self.y += self.y_speed
```

```
    else:
        self.has_bounced = False
    if self.z < 0 or self.z > Window.height * .64:  # If the ball has hit the upper or lower boundaries.
        self.z_speed *= -.8
        self.y_speed = self.z_speed + .8 * (self.y_speed + self.z_speed)
        self.z += self.z_speed
    if self.x < 0 or self.x > Window.width - self.width:  # If the ball has hit the left or right boundaries.
        self.x_speed *= -.8
        self.x += self.x_speed
```

AI

The next step is to create an AI class. To allow the player to switch characters, the AI class would have to be able to switch between acting like a player-controlled character and acting like an AI. For this, the AI class can inherit the character class and have a few extra methods and override some old methods. When the class is being controlled, it doesn't call the additional methods and the overridden methods only run the parent class method.

```
# Inherits the Character class. Adds computer-controlled functionality to the Character.
class AI(Character):
    def __init__(self, image_dict, ball, team):
        super().__init__(image_dict, ball, team)  # Initialise the parent class (Character) to get the same attributes.
        self.current_angle = math.radians(random.randint(0, 360))  # Start off with a random angle
        self.walk_tick = 0
        self.acted = False
        self.thrown = False
        self.prev_angle = 0
```

Run

The first method to be overridden is the run method. In the character class, the method receives the parameter "angle" which is given from the joystick and the method makes the character run towards this angle within the boundaries. For the AI, there is no input from a joystick. Instead the AI must base its angle on the situation it is in. If the ball is in the AI's side and is not mid-air plus the AI is closer to it than its teammates (otherwise all the AIs would run to the ball), the AI should run towards it. If the ball has been thrown at the AI by an enemy, the AI should run perpendicular to the angle from the AI to the ball. If the ball is being held by anyone or the AI is not close enough to run to the ball in the first situation, the AI should move around, changing its direction randomly at random but slightly large intervals (with small intervals, the AI's movement would look like it was vibrating rather than smoothly moving).

For the first two situations the program must first find the angle from the AI to the ball. This can be done by finding the arctan of the z distance from the AI to the ball over the x distance between the AI and the ball. For the first situation, nothing has to be done to this angle. For the second situation, the angle has to be changed by half pi. If the ball is moving towards below the character, the angle is increased by half pi, otherwise it is decreased by half pi. For the third situation, there is a walk_tick variable which is increased each time the character moves in the same angle. If the walk_tick reaches a random number between 5 and 15, a new angle is created and the walk_tick is reset to 0. Otherwise, the previous angle is used. Now the parent class run method is called with the parameter being the angle the method has created.

Random angle is actually a weighted random using a normal distribution. This is so that the AIs move in a random way but tend to move towards the back of the court since running to the front while the opponent has the ball is a bad idea. This makes the AI seem more intelligent and makes it more difficult to beat.

```
# Overrides the run method of Character. Will generate an angle based on the circumstances and then call the
# overridden run method with the generated angle.
def run(self, angle=None):
    if not self.selected:  # Ignore the overridden section if the character is selected.
        # First find the x and z distances of the character to the ball.
        dif_x = self.center_x - self.ball.x
        dif_z = self.z - self.ball.z
        angle = self.get_angle(dif_z, dif_x)  # Get the angle from the character to the ball.
        if not self.ball.throwing_team or self.ball.throwing_team == self.team:  # If the ball hasn't been thrown.
            self.thrown = False
        # If the ball has been thrown by the opponent
        if self.ball.throwing_team and self.ball.throwing_team != self.team:
            if self.thrown:  # If the angle has already been generated, don't create a new angle.
                angle = self.prev_angle
            else:
                # Find the direction the ball is travelling.
                ball_angle = self.get_angle(self.ball.z_speed, self.ball.x_speed)
```

```
            # We want the AI to try to move away from the ball. The best way to do this is to move perpendicular
            # to the ball's direction of travel. However, this is not realistic and the AIs will all move in the
            # same/opposite directions. Instead, the AI should move perpendicular to the thrower. However,
            # this still has the possibility that the ball is thrown to a different player and the AI runs into
            # the ball. To prevent this, the AI will run perpendicular to the thrower in the way that moves away
            # from the ball.
            if angle < ball_angle:
                angle -= math.pi / 2
            else:
                angle += math.pi / 2
            self.prev_angle = angle
            self.thrown = True
    elif self.ball.character or not self.bounds[0][0] + self.width / 10 < self.ball.x < \
            self.bounds[0][1] + self.width:  # If someone has the ball or the ball is in the other side.
        angle = self.random_angle()
    else:  # This means that the ball is on the AI's side and no-one has picked it up
        distance = math.sqrt(dif_x ** 2 + dif_z ** 2)  # Get the distance to the ball
        counter = 0
        for sprite in self.teammates:
            if math.sqrt((sprite.center_x - sprite.ball.x) ** 2 + (sprite.z - sprite.ball.z) ** 2) < \
                    distance:  # If the teammate is closer to the ball than the AI
                if counter == 1:  # If two teammates are closer then move randomly
                    angle = self.random_angle()
                else:
                    counter += 1
        if dif_x > 0:
            angle += math.pi  # The angle was away from the ball before now it is towards the ball.
    super().run(angle)  # Call the original run function with the generated angle.
```

```
# Generate a random angle but not every tick since this would just make the AI vibrate in place
def random_angle(self):
    # walk_tick is incremented every tick so it has to be divided by the fps to get a constant time.
    if self.walk_tick * 30 / self.game.fps < random.randint(10, 15):
        self.walk_tick += 1
    else:  # If the AI has run in the same direction for long enough.
        # We want the AIs to run randomly but weighted towards the back of their half.
        mean = 0
        # For team 1, pi radians as the angle is towards the back of their half whereas 0 is for team 2
        if self.team == 1:
            mean = math.pi
        self.current_angle = normal(mean, math.pi/2)  # Use the normal distribution with standard deviation as pi/2
        self.walk_tick = 0  # Reset the walk timer.
    return self.current_angle
```

Defend

When the ball is thrown at the AI, the AI should sometimes react with a defensive action (jump or catch). The program finds the angle from the ball to the AI and finds the angle of the ball travel. If these are close enough to each other, there is a 1/3 chance the AI jumps, a 1/3 chance the AI catches and a 1/3 chance the AI just moves. To do the actions, the parent class' methods for the actions are called.

```
# Jump, catch or do nothing. Only do the action once per throw
def defend(self):
    if self.ball.throwing_team and self.ball.throwing_team != self.team:
        if not self.acted:
            dif_x = self.center_x - self.ball.x
            dif_z = self.z - self.ball.z
            angle = self.get_angle(dif_z, dif_x)
            ball_angle = self.get_angle(self.ball.z_speed, self.ball.x_speed)
```

```
            if math.sqrt(dif_x ** 2 + dif_z ** 2) < self.width * 2:  # If the ball is close to the AI
                if abs(angle - ball_angle) < 0.1:  # If the ball is going towards the AI
                    chance = random.randint(0, 2)  # 1/3 chance for each action
                    if chance == 0:
                        super().jump()
                    elif chance == 1:
                        super().catch()
                self.acted = True
    else:
        self.acted = False  # Reset once the ball is no longer in midair.
```

Throw

If the AI has the ball. There is a chance that the AI throws the ball. The AI has an equal chance to heavy throw, light throw or pass and the AI could throw the ball at any time.

```
def random_throw(self):
        if self.has_ball and random.randint(0, self.game.fps) < 3:  # Only do sometimes when the character has the ball.
            chance = random.randint(0, 2)
            if chance < 1:
                super().light_throw()
            elif chance < 2:
                super().heavy_throw()
            else:
                super().pass_to_teammate()
```

Networking

Server

The server will work as both a connection manager and a game server. As a connection manager, the server works as a middleman between clients, passing on messages and storing data. As a game server, the server will take in game information from clients and distribute it to the other clients.

The server will be waiting for messages from clients and will then respond appropriately depending on the message. The clients will send messages based on the user's actions within the application e.g. when the user clicks on host, the client sends a message to the server and the server does an action based on this.

The server will have three dictionaries:

- clients – a dictionary with the keys being the usernames of the clients and the values being instances of the connection object between the server and the client.
- pending_lobbies – a dictionary with the keys being the lobby id and the values being dictionaries. The inner dictionaries will have keys as team names plus a key for host and a key for all_players. The values will be lists of all the usernames of the clients within that group e.g. team_1 has a list of all the users in team 1, all_players has a list of all the users in the pending lobby.
- lobbies – a dictionary with the keys being lobby id and the values being dictionaries. These dictionaries have keys as users in the lobby and values as a list of the characters the user has chosen.

The server waits for messages from the client. A method specific to the type of message is then called which does the correct response to the message. For example, if the client sends: "create_lobby", the method create_lobby is called in which the server makes some checks and creates a lobby. It then sends the client a message about the lobby so the client can join the lobby. The messages can also have parameters e.g. "add_friend;Username1" means that the client wants to add Username1 as a friend.

```
from twisted.internet import reactor, protocol
from os import makedirs, path
from online_images import send_images
from random import choice


# Class that handles online communication with clients. Inherits the Protocol class from twisted.
class Server(protocol.Protocol):
    username = None
    lobby = None
    team = None
    ingame = False
    images = b''
    previous_status = ""

    # Called when a message is received from a client.
    def dataReceived(self, data):
        try:
            messages = data.decode('utf-8').split("\n")
        except UnicodeDecodeError:
            messages = self.receive_images(data)
        for message in messages:
            if message.startswith("username") and not self.username:
                self.transport.setTcpNoDelay(True)
                self.username = message.replace("username;", "")
                self.factory.clients[self.username] = self
            elif message.startswith("wants_username"):
                if self.check_username(message):
                    self.transport.write("username_taken\n".encode("utf-8"))
                else:
                    self.transport.write("username_accepted\n".encode("utf-8"))
            elif message.startswith("add_friend"):
                if self.check_username(message):
                    self.transport.write("added_friend\n".encode("utf-8"))
                else:
                    self.transport.write("no_friend\n".encode("utf-8"))
```

```
        elif message.startswith("custom_characters"):
            self.custom_characters(message)
        elif message.startswith("friends"):
            self.get_friends_images(message)
        elif message.startswith("check_status"):
            self.friends_statuses(message)
        elif message == "list_lobbies":
            self.list_lobbies()
        elif message == "create_lobby" and not self.lobby:
            self.create_lobby()
        elif message.startswith("request"):
            self.pass_on_request(message)
        elif message.startswith("accept"):
            self.accept(message)
        elif message == "start":
            self.start()
        elif message == "leave":
            self.connectionLost(leaving=True)
        elif message.startswith("team"):
            team = message.partition(";")[-1]
            if len(self.factory.pending_lobbies[self.lobby][team]) == 3 or team == self.team:
                team = "centre"
            self.factory.pending_lobbies[self.lobby][self.team].remove(self.username)
            self.factory.pending_lobbies[self.lobby][team].append(self.username)
            self.team = team
            self.send_teams()
        elif message.startswith("characters"):
            self.character_selection(message)
        if message.startswith("update"):
            self.ingame = True
            self.update(message)
        else:
            self.ingame = False

    # Called when a client disconnects. This method handles them being removed from lobbies they were in.
    def connectionLost(self, reason=None, leaving=False):
```

```
        if self.lobby:
            if self.lobby in self.factory.lobbies:
                if self.username in self.factory.lobbies[self.lobby]:
                    message = self.previous_status.replace("update;", "").split(";;")
                    for i in range(len(message)):
                        message[i] = message[i].split(";")
                        if message[i][0] != "ball":
                            message[i][7] = "disconnected"
                        message[i] = ";".join(message[i])
                    message = "update;" + ";;".join(message)
                    self.update(message)
                    del self.factory.lobbies[self.lobby][self.username]
            # If they were the host of a pending lobby, either make the next player host or delete the lobby if there
            # are no other players.
            if self.factory.pending_lobbies[self.lobby]["host"] == self.username:
                if len(self.factory.pending_lobbies[self.lobby]["all_players"]) > 1:
                    self.factory.pending_lobbies[self.lobby]["host"] = \
                        self.factory.pending_lobbies[self.lobby]["all_players"][1]
                else:
                    del self.factory.pending_lobbies[self.lobby]
            self.factory.pending_lobbies[self.lobby][self.team].remove(self.username)
            self.factory.pending_lobbies[self.lobby]["all_players"].remove(self.username)
            self.send_teams()  # Update the teams for the other users.
            self.lobby = None
        if self.username and not leaving:
            del self.factory.clients[self.username]

    # Checks if the username has already been used.
    @staticmethod
    def check_username(message):
        username = message.partition(";")[-1]
        if path.isfile("Character Storage/" + username + ".txt"):
            return True
        else:
            return False
```

```
    # Checks if the server has the characters saved already and update the user's info file.
    def custom_characters(self, message):
        output = "unknown_characters;"
        characters = message.replace("custom_characters;", "").split(";")
        for character in characters:
            if not path.isdir("Character Storage/" + character):
                output += character + ";"
        with open("Character Storage/" + self.username + ".txt", "w+") as user_file:
            user_file.write("\n".join(characters))
        output += "\n"
        self.transport.write(output.encode("utf-8"))

    # Get the data of all the images of the user's friends that they don't have and send them.
    def get_friends_images(self, message):
        message = message.split(";;")[1:]
        friends = message[0].split(";")
        friend_images = message[1].split(";")
        characters = []
        for friend in friends:
            if friend == "":
                continue
            with open("Character Storage/" + friend + ".txt", "r") as images_file:
                characters.extend(images_file.read().split("\n"))
        characters = list(set(characters))  # Converting images to a set removes any duplicates
        for character in characters:
            if character in friend_images:
                characters.remove(character)
        data = send_images(characters, path="Character Storage")
        self.transport.write(data)

    # Format the message into images and save them.
    def receive_images(self, data):
        self.images += data
        messages = []
        if b'start' in data:
            messages = self.images.split(b'[delimiter]')
```

```
            messages = "".join(list(map(lambda x: x.decode("utf-8"), messages[:messages.index(b"start")]))).split("\n")
        if b'end' in data:
            self.images = self.images.split(b'[delimiter]')
            # Capture any messages that were sent before the images
            messages = ("".join(list(map(lambda x: x.decode("utf-8"), self.images[:self.images.index(b"start")]))) +
                        "".join(list(map(lambda x: x.decode("utf-8"),
                                         self.images[self.images.index(b"end") + 1:])))).split("\n")
            for image in self.images[self.images.index(b"start") + 1:self.images.index(b"end")]:
                try:
                    pathname = image.decode("utf-8").replace("Images", "Character Storage")
                except UnicodeDecodeError:
                    try:
                        with open(pathname, "w+b") as file:
                            file.write(image)
                    except FileNotFoundError:
                        makedirs("/".join(pathname.split("/")[:-1]))
                        with open(pathname, "w+b") as file:
                            file.write(image)
        self.transport.write("download_success\n".encode("utf-8"))
        return messages

    # Get the current statuses of the user's friends and send them.
    def friends_statuses(self, message):
        friends = message.split(";")[1:]
        output = "friends_statuses;;"
        for friend in friends:
            output += friend + ";"
            if friend in self.factory.clients:
                if self.factory.clients[friend].lobby:
                    if self.factory.clients[friend].ingame:
                        output += "In a game;;"
                    else:
                        output += "in_lobby-" + \
                                  self.factory.pending_lobbies[self.factory.clients[friend].lobby]["host"] \
                                  + "-" + str(len(self.factory.pending_lobbies[self.factory.clients[friend].lobby]
                                                  ["all_players"])) + ";;"
```

```
                else:
                    output += "Online;;"
            else:
                output += "Offline;;"
        output += "\n"
        self.transport.write(output.encode("utf-8"))

    # Send all the pending lobbies.
    def list_lobbies(self):
        hosts = []
        for lobby in self.factory.pending_lobbies:
            if lobby in self.factory.lobbies:  # If the players are in a game.
                continue
            if len(self.factory.pending_lobbies[lobby]["all_players"]) > 0:
                if not self.factory.clients[self.factory.pending_lobbies[lobby]["host"]].ingame:
                    hosts.append(self.factory.pending_lobbies[lobby]["host"] + ";" +
                                 str(len(self.factory.pending_lobbies[lobby]["all_players"])))
            else:
                del self.factory.pending_lobbies[lobby]
        self.transport.write(("lobbies;" + ";;".join(hosts) + "\n").encode("utf-8"))

    # Create a new pending lobby.
    def create_lobby(self):
        self.lobby = len(self.factory.pending_lobbies) + 1
        self.factory.pending_lobbies[self.lobby] = {}
        self.factory.pending_lobbies[self.lobby]["host"] = self.username
        self.factory.pending_lobbies[self.lobby]["team_1"] = []
        self.factory.pending_lobbies[self.lobby]["team_2"] = []
        self.factory.pending_lobbies[self.lobby]["centre"] = [self.username]
        self.factory.pending_lobbies[self.lobby]["all_players"] = [self.username]
        self.team = "centre"
        self.send_teams()

    # Pass a request to join from a user to the host of the lobby they want to join.
    def pass_on_request(self, message):
        host = message.partition(";")[-1]
```

```
        if host in self.factory.clients:
            if self.factory.clients[host].lobby:
                self.factory.clients[host].transport.write(("request;" + self.username + "\n").encode("utf-8"))
            else:
                self.transport.write("fail\n".encode("utf-8"))
        else:
            self.transport.write("fail\n".encode("utf-8"))

    # Accept a request to join a lobby.
    def accept(self, message):
        player_name = message.partition(";")[-1]
        if player_name in self.factory.clients:
            if not self.factory.clients[player_name].lobby and \
                    len(self.factory.pending_lobbies[self.lobby]["all_players"]) < 6:
                self.factory.pending_lobbies[self.lobby]["centre"].append(player_name)
                self.factory.pending_lobbies[self.lobby]["all_players"].append(player_name)
                self.factory.clients[player_name].team = "centre"
                self.factory.clients[player_name].lobby = self.lobby
                self.factory.clients[player_name].transport.write("accepted\n".encode("utf-8"))
                self.send_teams()
        else:
            self.transport.write("fail\n".encode("utf-8"))

    # Start a game from the pending lobby.
    def start(self):
        try:
            for name in self.factory.pending_lobbies[self.lobby]["all_players"]:
                try:
                    self.factory.clients[name].transport.write("start\n".encode("utf-8"))
                except KeyError:
                    self.factory.pending_lobbies[self.lobby][self.team].remove(name)
        except KeyError:
            self.transport.write("fail\n".encode("utf-8"))
        self.factory.lobbies[self.lobby] = {}
        self.factory.lobbies[self.lobby]["characters"] = 0
```

```
    # Send the updated teams to all the users in the pending lobby.
    def send_teams(self):
        try:
            for name in self.factory.pending_lobbies[self.lobby]["all_players"]:
                try:
                    self.factory.clients[name]. \
                        transport.write(("teams;" +
                                         ";".join(self.factory.pending_lobbies[self.lobby]["team_1"]) + ";;" +
                                         ";".join(self.factory.pending_lobbies[self.lobby]["centre"]) + ";;" +
                                         ";".join(self.factory.pending_lobbies[self.lobby]["team_2"]) + ";;" +
                                         self.factory.pending_lobbies[self.lobby]["host"] + "\n")
                                        .encode("utf-8"))
                except KeyError:
                    self.factory.pending_lobbies[self.lobby]["all_players"].remove(name)
        except KeyError:
            pass

    # Send all the characters selected to the users in the lobby.
    def character_selection(self, message):
        message = message.split(";")
        if self.username not in self.factory.lobbies[self.lobby]:
            self.factory.lobbies[self.lobby][self.username] = []
            for character in message[1:]:
                self.factory.lobbies[self.lobby][self.username].append(character)
                if character != "team_1" and character != "team_2":
                    self.factory.lobbies[self.lobby]["characters"] += 1
        if self.factory.lobbies[self.lobby]["characters"] == 6:  # Once all the characters have been chosen.
            background = choice(["Images/[0.7]background_mud.png", "Images/[0.6]background_grass.png",
                                 "Images/[0.7]background_night.png", "Images/[0.3]background_snow.png"])
            try:
                for name in self.factory.lobbies[self.lobby]:
                    if name == "characters":
                        continue
                    try:
                        output = ""
                        for user in self.factory.lobbies[self.lobby]:
```

```
                        if user == "characters":
                            continue
                        output += user + ";"
                        for character in self.factory.lobbies[self.lobby][user]:
                            output += character + ";"
                        output += ";"
                    self.factory.clients[name].transport.write(
                        ("characters_chosen;" + output + ";" + background + "\n").encode("utf-8"))
                except KeyError:
                    del self.factory.lobbies[self.lobby][name]
            except KeyError:
                self.transport.write("fail\n".encode("utf-8"))

    # Pass on the updates to all the users in the lobby.
    def update(self, message):
        self.previous_status = message
        try:
            for name in self.factory.lobbies[self.lobby]:
                if name == "characters" or name == self.username:
                    continue
                self.factory.clients[name].transport.write((message+"\n").encode("utf-8"))
        except KeyError:
            self.transport.write("fail\n".encode("utf-8"))


class ServerFactory(protocol.Factory):
    def __init__(self):
        self.protocol = Server
        self.clients = {}
        # self.pending_lobbies will be a dictionary with keys being lobbies and the values being dictionaries of each
        # lobby's teams and players.
        self.pending_lobbies = {}
        # self.lobbies will be a dictionary with keys being lobbies and values being dictionaries of each players
        # characters.
        self.lobbies = {}
```

```
# Start the server.
reactor.listenTCP(50000, ServerFactory())
reactor.run()
```

Client

The client works similarly to the server except most of the functions it calls are within the GUI itself e.g. if It receives data about the available lobbies, it then calls a function to display the available lobbies in the GUI.

The client also has methods for dealing with disconnects and attempting to reconnect. Here is the code for the client:

```
from misc import switch
from twisted.internet import protocol
from online_images import send_images
from os import makedirs
from twisted.internet.reactor import callLater


# Class that handles online communication with the server. Inherits the Protocol class from twisted.
class Client(protocol.Protocol):
    characters_received = False
    images = b''
    scheduled_timout = None

    # Called when the client connects to the server.
    def connectionMade(self):
        if self.scheduled_timout:
            self.scheduled_timout.cancel()
        self.transport.setTcpNoDelay(True)  # This prevents the Nagle algorithm which delays and coalesces messages
        self.factory.parent.transport = self.transport
        if self.username != "":  # If the user already has a username.
            self.factory.parent.initialise_connection()
        else:
            self.factory.parent.set_username()

    # Called when the client receives a message from the server.
    def dataReceived(self, data):
        try:
            # Messages always end in "\n" in case they get combined so they can be split up again.
```

```
            messages = data.decode('utf-8').split("\n")
        except UnicodeDecodeError:  # This means that the message is an image instead.
            messages = self.receive_images(data)
        for message in messages:
            if message.startswith("unknown_characters"):
                message = message.replace("unknown_characters;", "")
                if message == "":  # If there are no unknown characters, start.
                    self.factory.parent.create_buttons()
                else:
                    self.send_images(message.split(";"))
            elif message == "username_taken":
                self.factory.parent.username_taken()
            elif message == "username_accepted":
                self.factory.parent.username_accepted()
            elif message == "added_friend":
                self.factory.parent.added_friend()
            elif message == "no_friend":
                self.factory.parent.no_friend()
            elif message == "download_success":
                self.factory.parent.create_buttons()
            elif message.startswith("friends_statuses"):
                self.factory.parent.friends_statuses(message)
            elif message.startswith("teams"):
                message = message.replace("teams;", "")
                self.lobby_screen.update_slots(message)
            elif message.startswith("lobbies"):
                message = message.replace("lobbies;", "")
                self.join_screen.show_lobbies(message)
            elif message.startswith("request"):
                message = message.replace("request;", "")
                self.lobby_screen.request_received(message)
            elif message == "accepted":
                switch("Lobby", self.join_screen.manager, "down")
            elif message.startswith("start"):
                self.lobby_screen.start_game()
            elif message.startswith("characters_chosen"):
```

```
            if not self.characters_received:  # Only do this once.
                self.characters_received = True
                self.character_selection.enter_game(message)
        elif message.startswith("update"):
            self.game.receive_message(message.replace("update;", ""))

def send_images(self, characters):
    data = send_images(characters)
    self.transport.write(data)

def receive_images(self, data):
    self.images += data
    messages = []
    if b'start' in data:
        messages = self.images.split(b'[delimiter]')
        messages = ("".join(list(map(lambda x: x.decode("utf-8"), messages[:messages.index(b"start")])))).split(
            "\n")
    if b'end' in data:
        self.images = self.images.split(b'[delimiter]')
        # Capture any messages that were sent before the images
        messages = ("".join(list(map(lambda x: x.decode("utf-8"), self.images[:self.images.index(b"start")])))) +
                    "".join(list(map(lambda x: x.decode("utf-8"),
                                     self.images[self.images.index(b"end") + 1:])))).split("\n")
        for image in self.images[self.images.index(b"start") + 1:self.images.index(b"end")]:
            try:
                pathname = image.decode("utf-8").replace("Character Storage", "Images")
            except UnicodeDecodeError:
                try:
                    with open(pathname, "w+b") as file:
                        file.write(image)
                # w+ mode creates the file if it doesn't exist but doesn't create the directory if it doesn't exist.
                except FileNotFoundError:
                    makedirs("/".join(pathname.split("/")[:-1]))  # Create the directory.
                    with open(pathname, "w+b") as file:
                        file.write(image)
    return messages
```

```
# Factory for the client protocol. Inherits the ClientFactory class from twisted.
class ClientFactory(protocol.ClientFactory):
    protocol = Client

    def __init__(self, parent, username):
        self.protocol.username = username
        self.parent = parent

    def clientConnectionLost(self, connector, reason):
        connector.connect()  # Try to reconnect
        self.protocol.scheduled_timeout = callLater(5, connector.stopConnecting)  # Timeout after 5 seconds

    def clientConnectionFailed(self, connector, reason):
        self.parent.disconnected(reason.getErrorMessage())
```

Downloading and Uploading Characters

There were two different options for saving characters:

- When players are in the lobby, all the players download all the images from each other. Would cause a massive slowdown for everyone.
- When a user adds a friend, they download all of that friend's images. This means that user will only be able to see their friends' characters. This is ok though because those are the only ones that people really care about.

The second option made a lot more sense, so I decided to go with that one.

However, if the friend creates a new image, the user won't receive it. The solution to this is that whenever a user connects to the server, the user sends the names of all of their custom characters (the names are hashed so they should be unique to that character). The server checks if these characters are stored in the online file system. The server sends a message containing the names of all the images not already saved in the online file system. The user then uploads those characters only. The server stores the names of all the characters from that user in the info file created when the user first created a username.

In the same way, the user still needs to download any new characters their friends have created. To do that, the user has to send a list of friends and characters already downloaded from friends to the server whenever the user connects to the server. The server knows which characters each of the friends have uploaded because of the info file. The server then uploads all the characters that the user didn't already have.

Now whenever a user plays with a friend and the friend picks one of their custom characters, the user can see that custom character.

The images are sent in a single block. I created a start and an end so that any messages before and after can still be interpreted and used. Between each image is a delimiter.

```
from image_editing import make_dict


# Function for formatting the images to be sent to and from the client and server.
def send_images(characters, path="Images"):
    data = b"[delimiter]start[delimiter]"
```

```
    for character in characters:
        if character:
            image_dict = make_dict(character, path=path)
            for value in image_dict.values():
                for pathname in value:
                    pathname = pathname.replace("Images", path)
                    with open(pathname, "rb") as image:
                        data += pathname.encode("utf-8") + b'[delimiter]' + image.read() + b'[delimiter]'
    data += b'[delimiter]end[delimiter]'
    return data
```

When the images are received, receive_images is called. This splits up the block of images into single images and saves them in the correct structure. It also takes messages before and after the block and interprets them.

```
# Format the message into images and save them.
def receive_images(self, data):
    self.images += data
    messages = []
    if b'start' in data:
        messages = self.images.split(b'[delimiter]')
        messages = "".join(list(map(lambda x: x.decode("utf-8"), messages[:messages.index(b"start")]))).split("\n")
    if b'end' in data:
        self.images = self.images.split(b'[delimiter]')
        # Capture any messages that were sent before the images
        messages = ("".join(list(map(lambda x: x.decode("utf-8"), self.images[:self.images.index(b"start")]))) +
                    "".join(list(map(lambda x: x.decode("utf-8"),
                                     self.images[self.images.index(b"end") + 1:])))).split("\n")
        for image in self.images[self.images.index(b"start") + 1:self.images.index(b"end")]:
            try:
                pathname = image.decode("utf-8").replace("Images", "Character Storage")
            except UnicodeDecodeError:
                try:
```

```
                with open(pathname, "w+b") as file:
                    file.write(image)
            except FileNotFoundError:
                makedirs("/".join(pathname.split("/")[:-1]))
                with open(pathname, "w+b") as file:
                    file.write(image)
    self.transport.write("download_success\n".encode("utf-8"))
    return messages
```

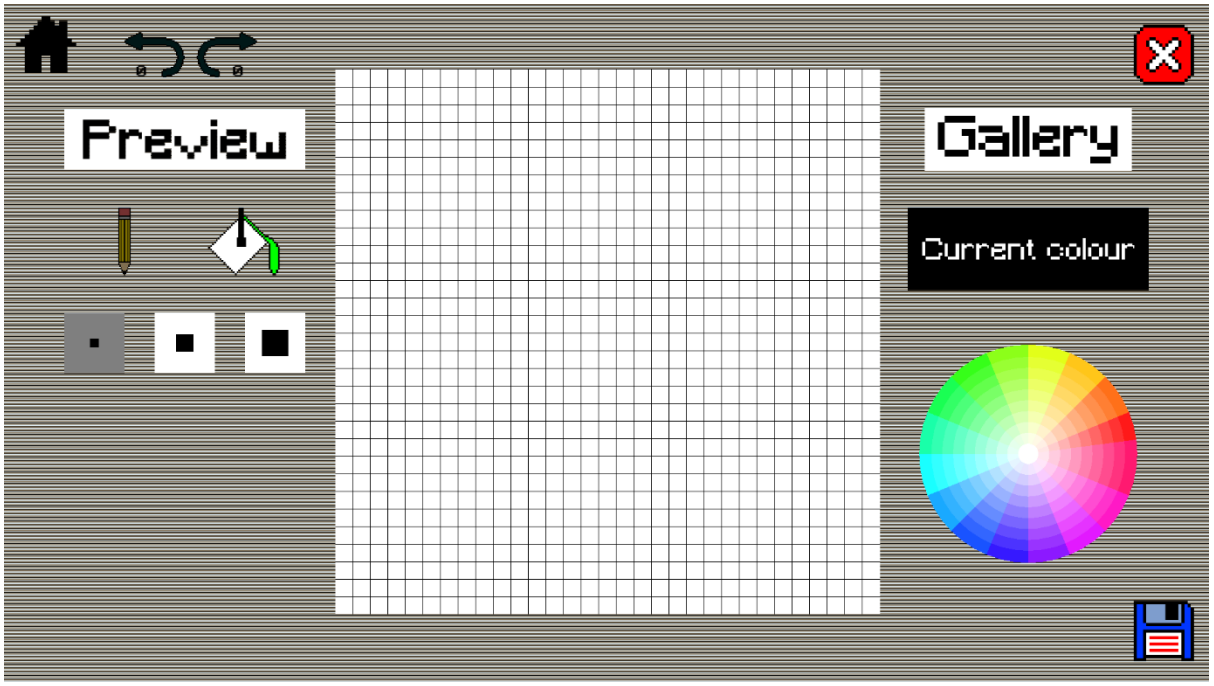
User Interface Screenshots

Main Menu

Left button is for character creation, middle is play and right is change controls.

Character Creation

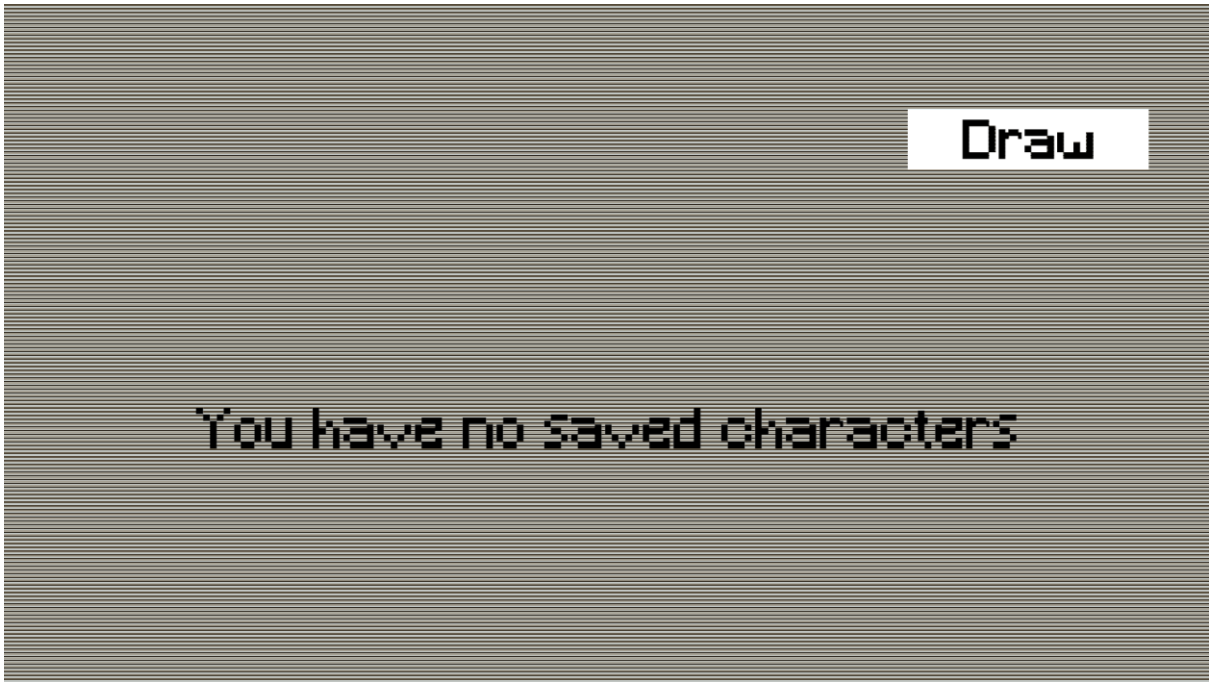
First screen of character creation.



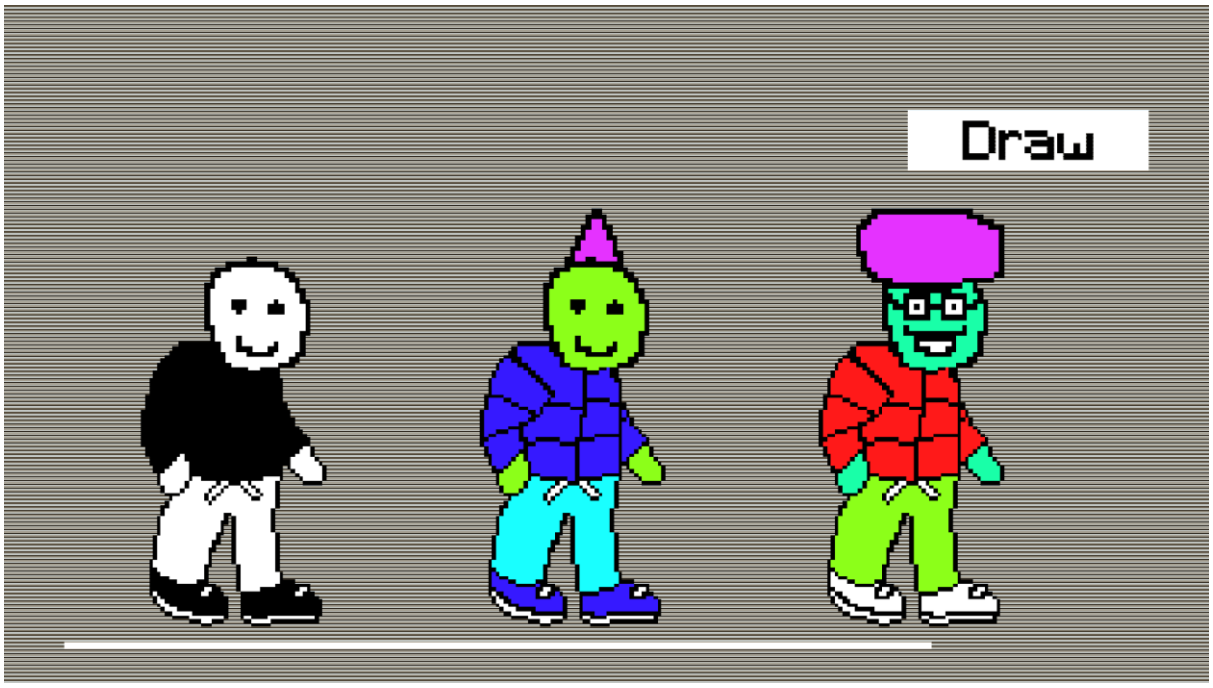
Canvas screen of character creation. Pressing the pencil when it is already selected toggles it to eraser.

Preview

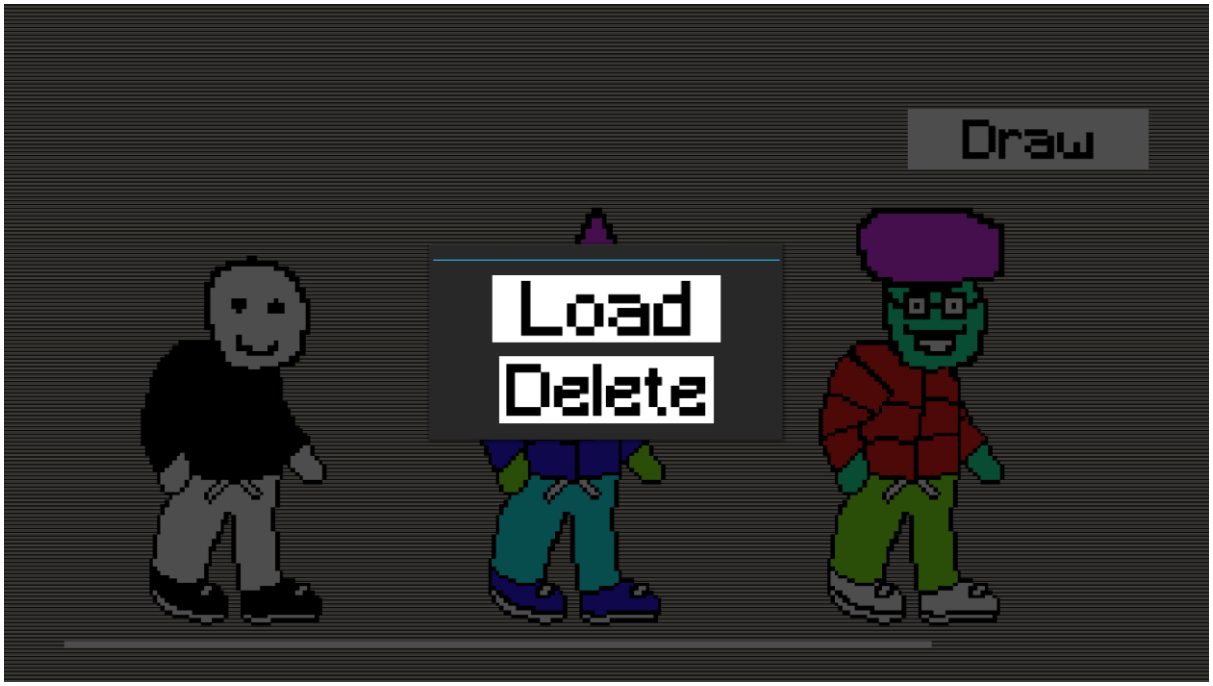
Gallery



Gallery with no saved characters.

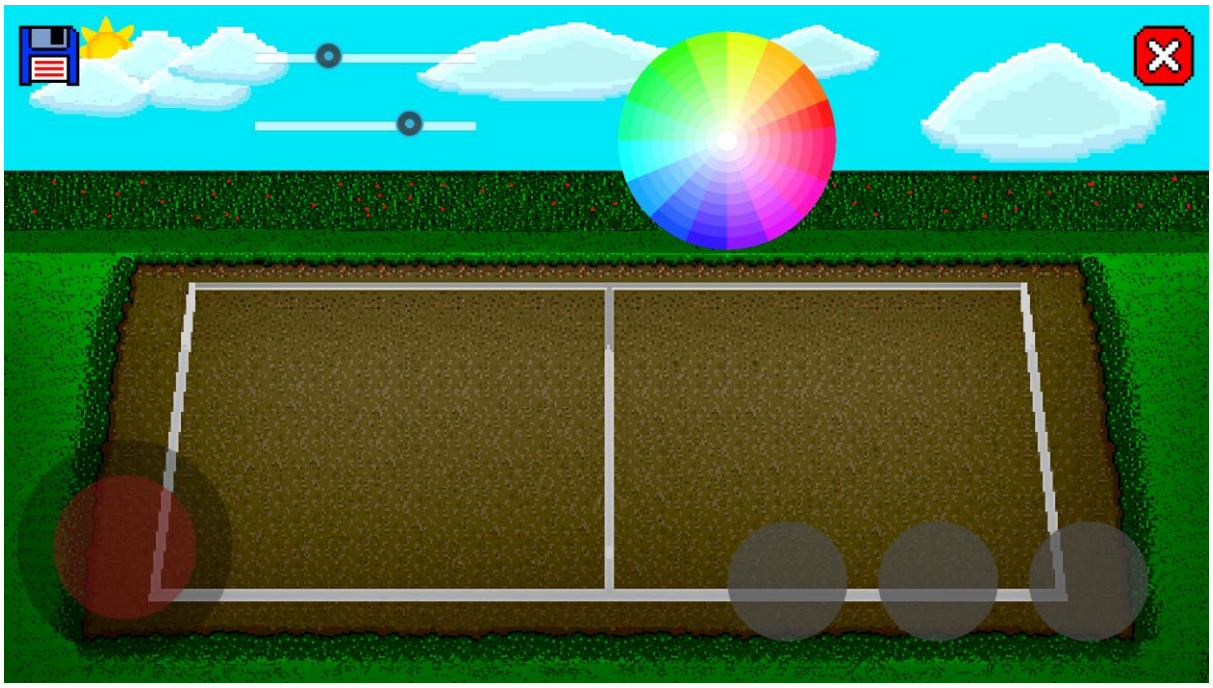


Gallery with saved characters. User can drag horizontally to scroll through all their characters.

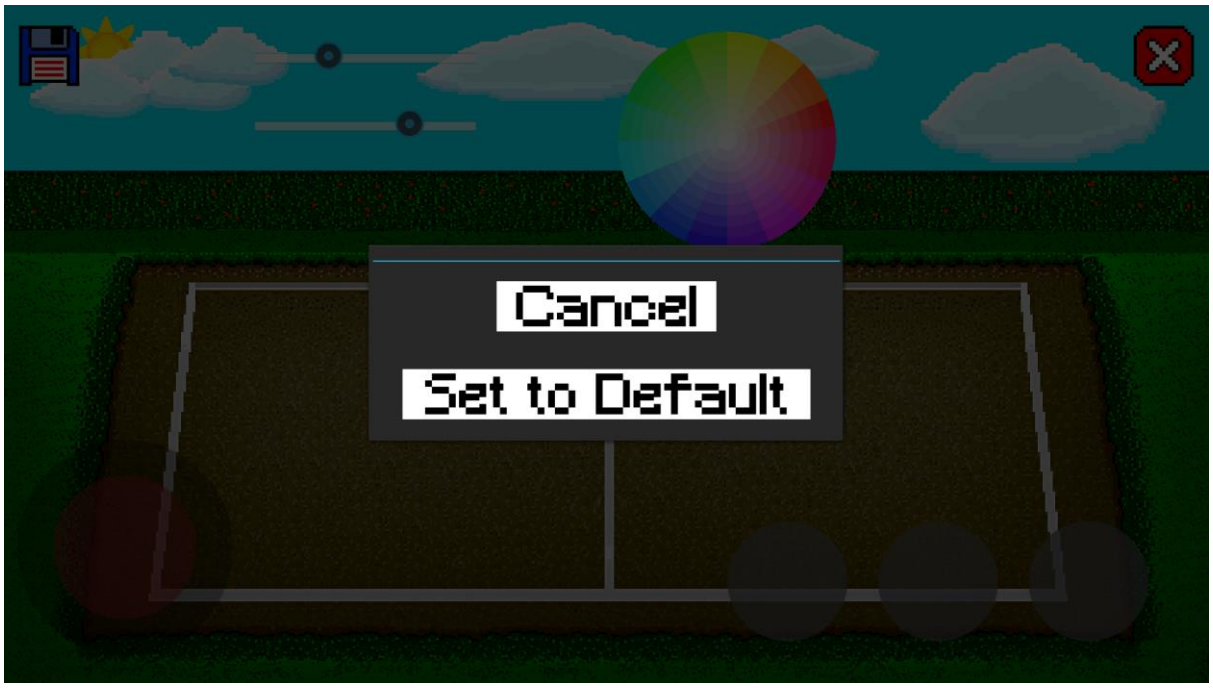


When pressing one of the characters, get a popup for options.

Change Controls

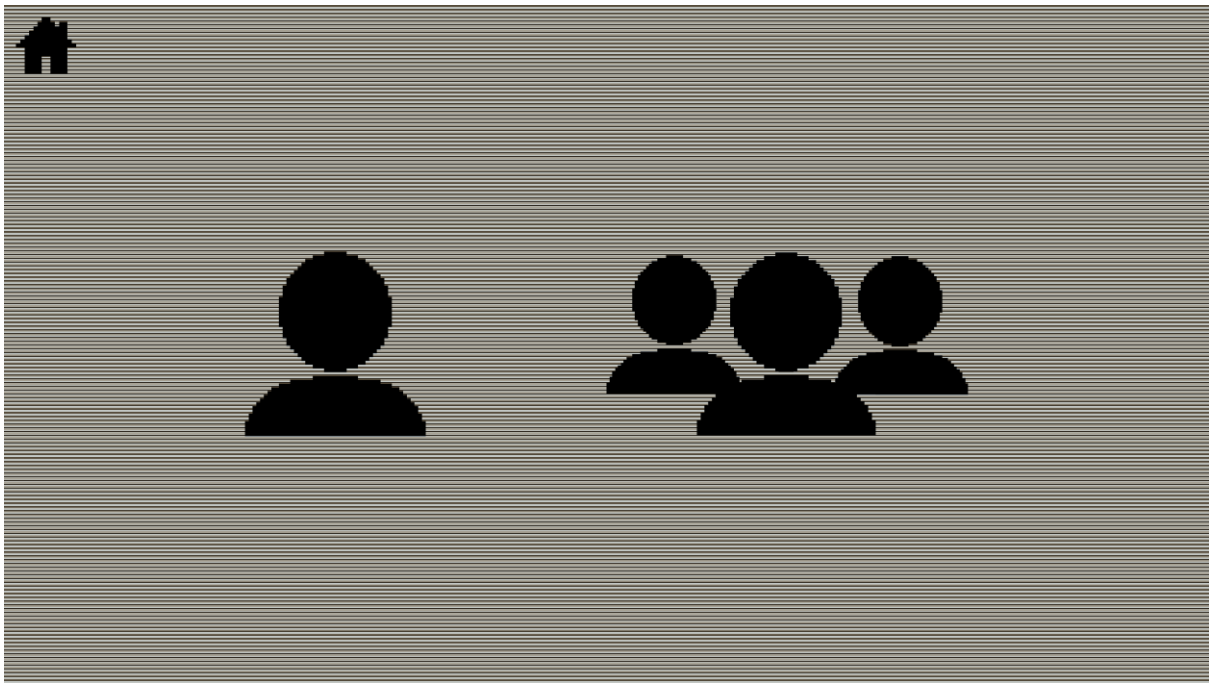


User can move around the buttons and joystick. The sliders are for size and opacity with size on the top and opacity below it.



Popup from pressing the X in the top right.

Play

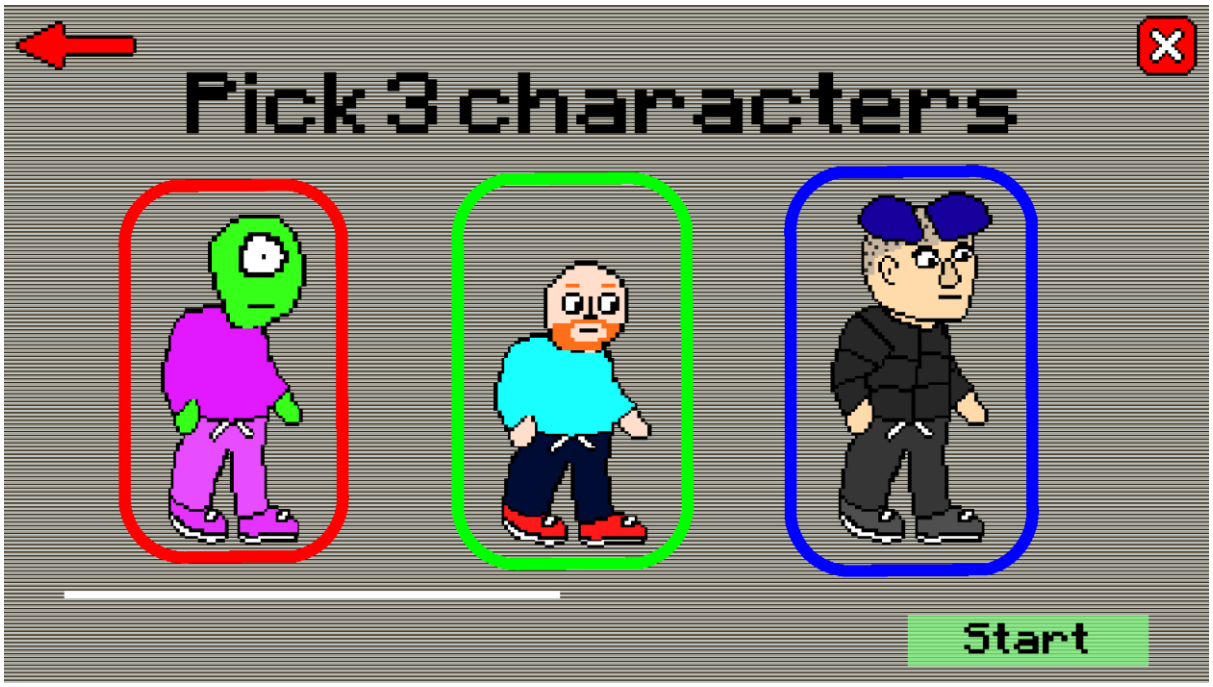


Screen from pressing the play button on the main menu. The left option is single-player and the right option is multi-player.

Character Selection

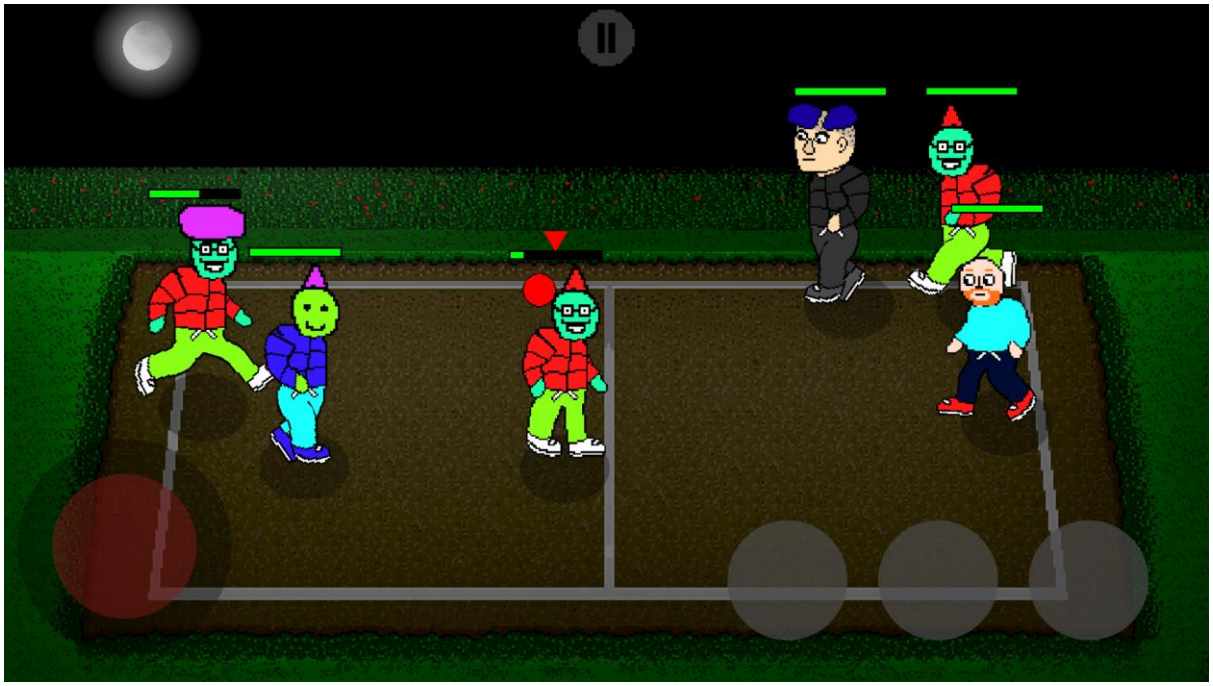
Character election screen also has a scroll feature like the gallery.

Contains both in-built characters and custom characters.

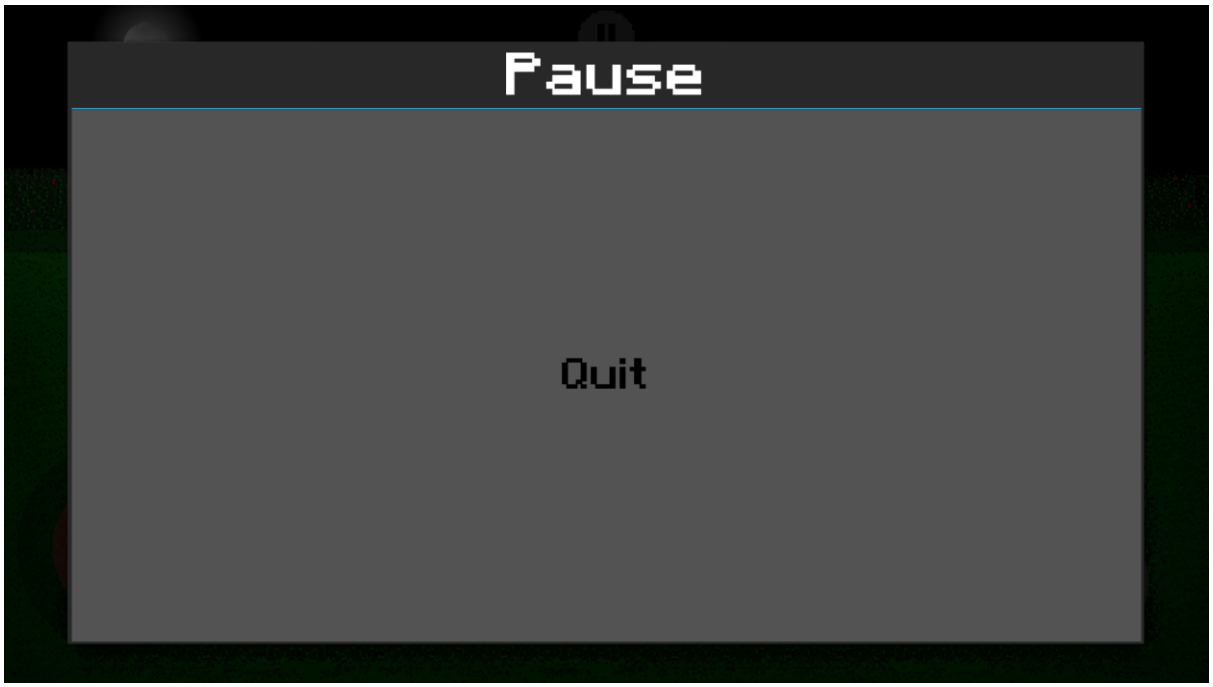


3 different coloured outlines.

Gameplay



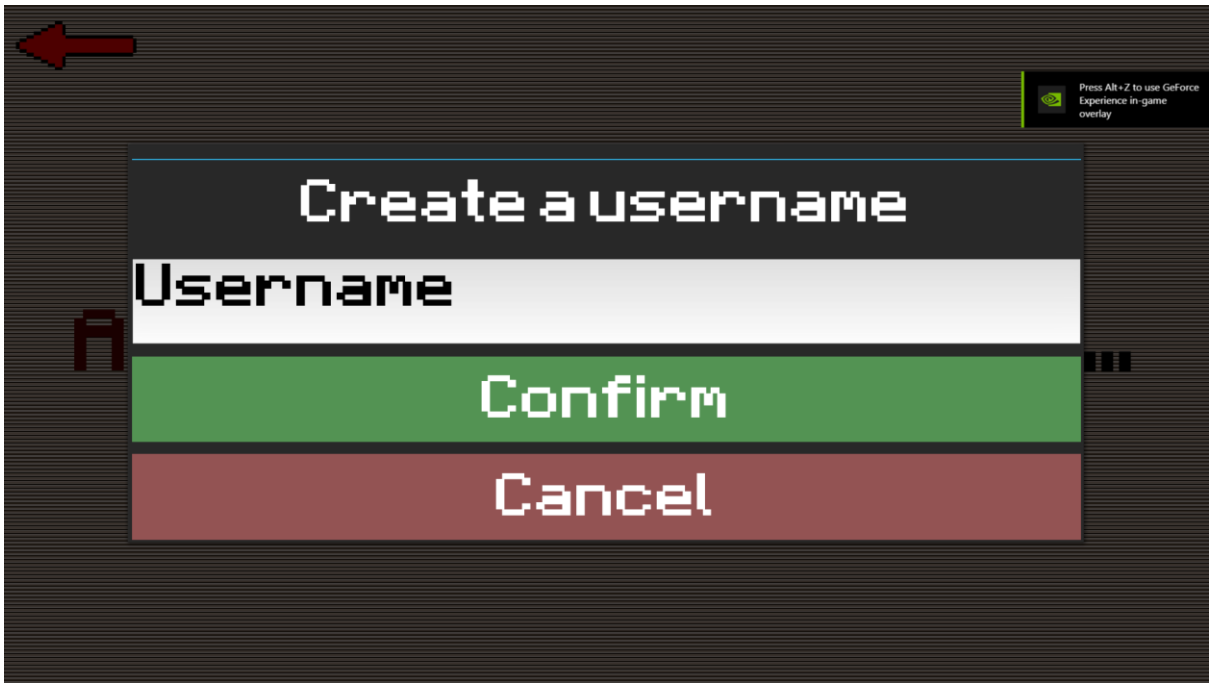
Gameplay with the night background.



The pause menu just has a big quit button.

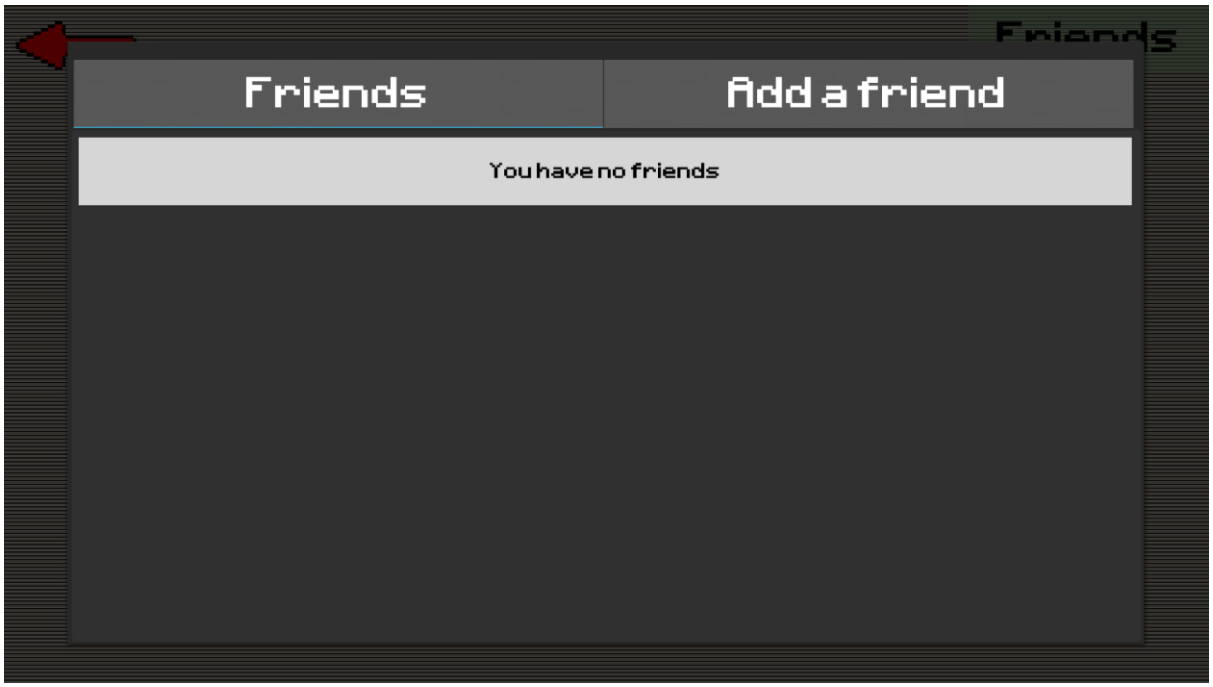
Multiplayer

When connecting to the server, this message appears. The red flows through the text as an animation.

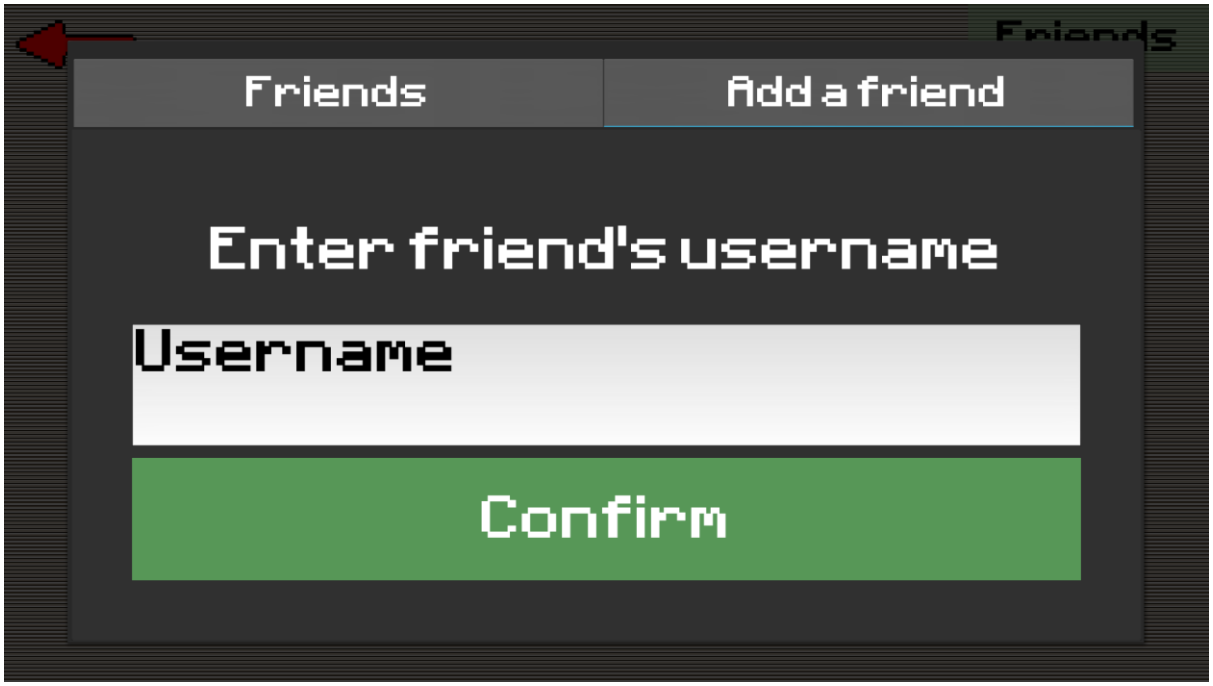


Once connected, if the user hasn't made a username yet.

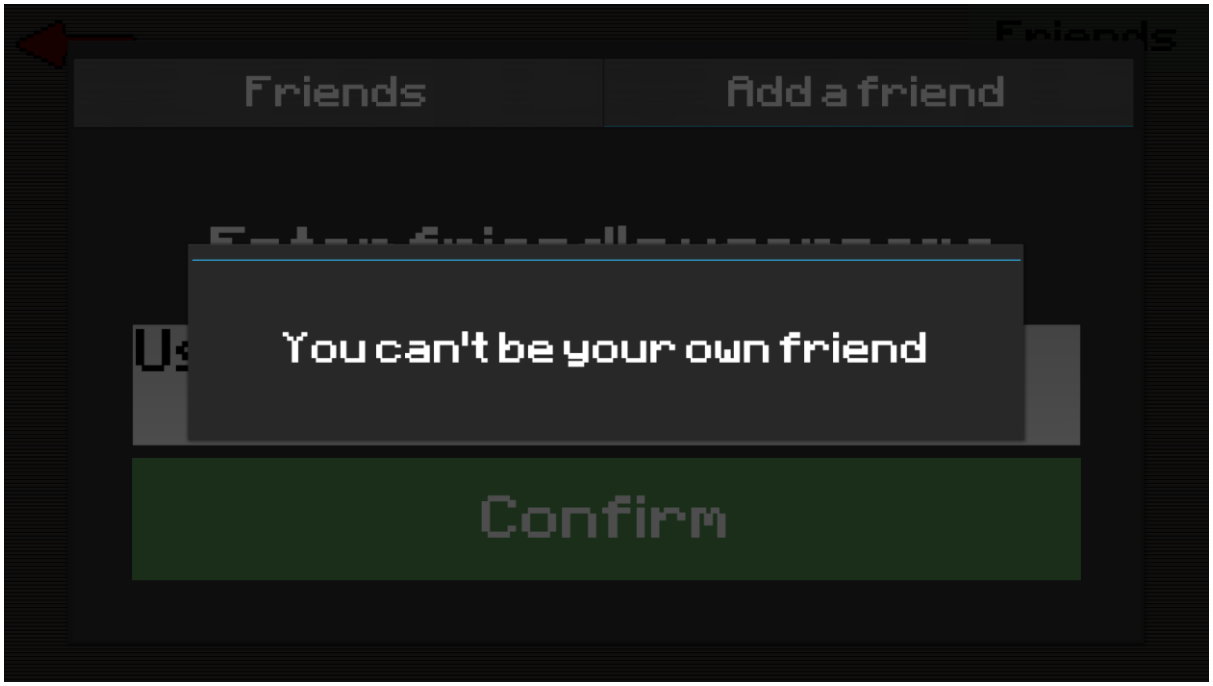
This is the screen once connected and with a username.



Friends menu with no friends added.



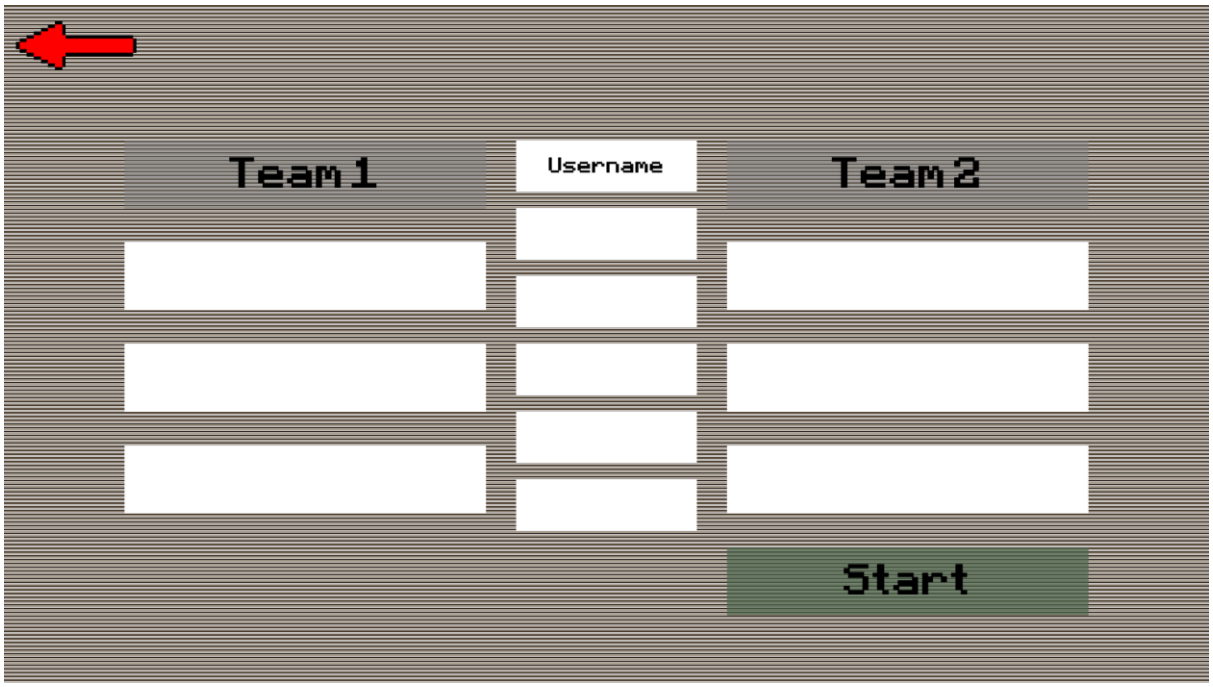
Add a new friend.



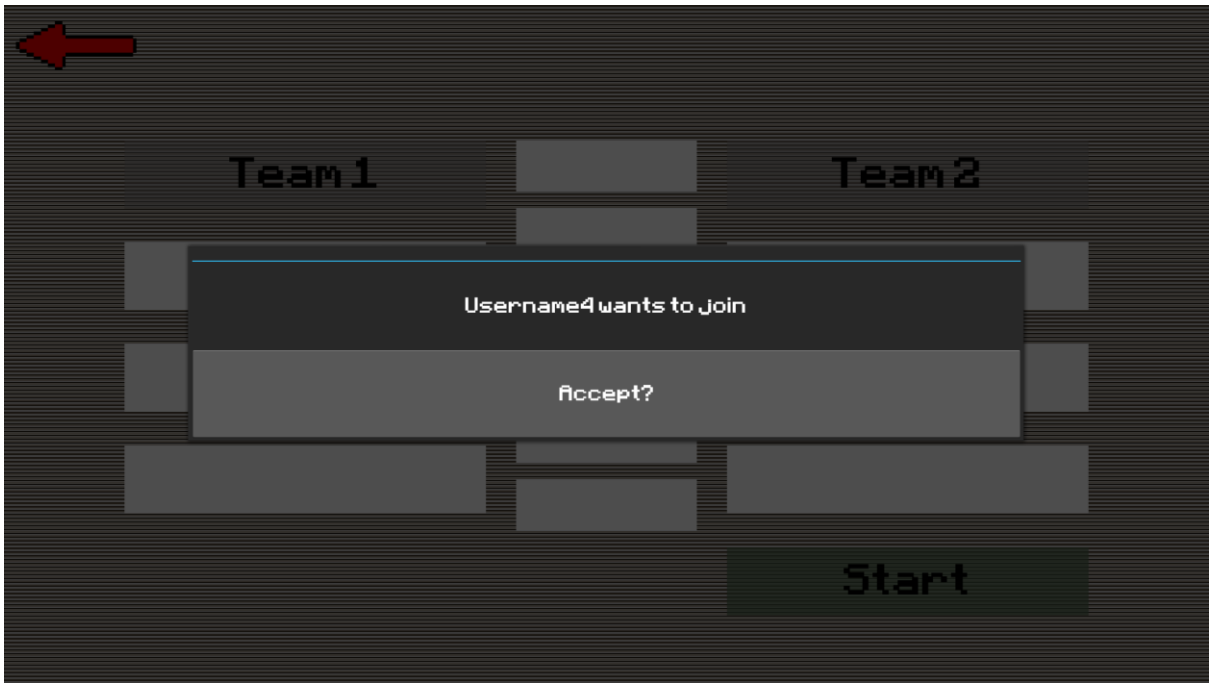
If the user enters their own username.

Friends list when user has some friends added.

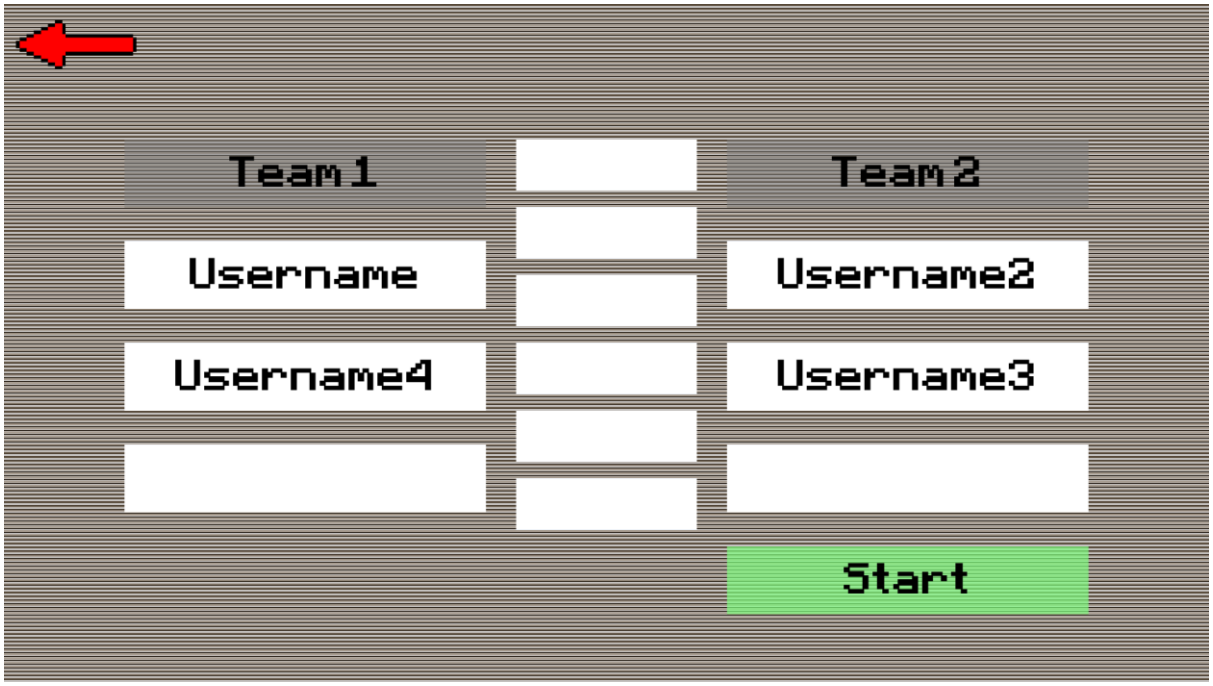
Lobby



Lobby with just the host.



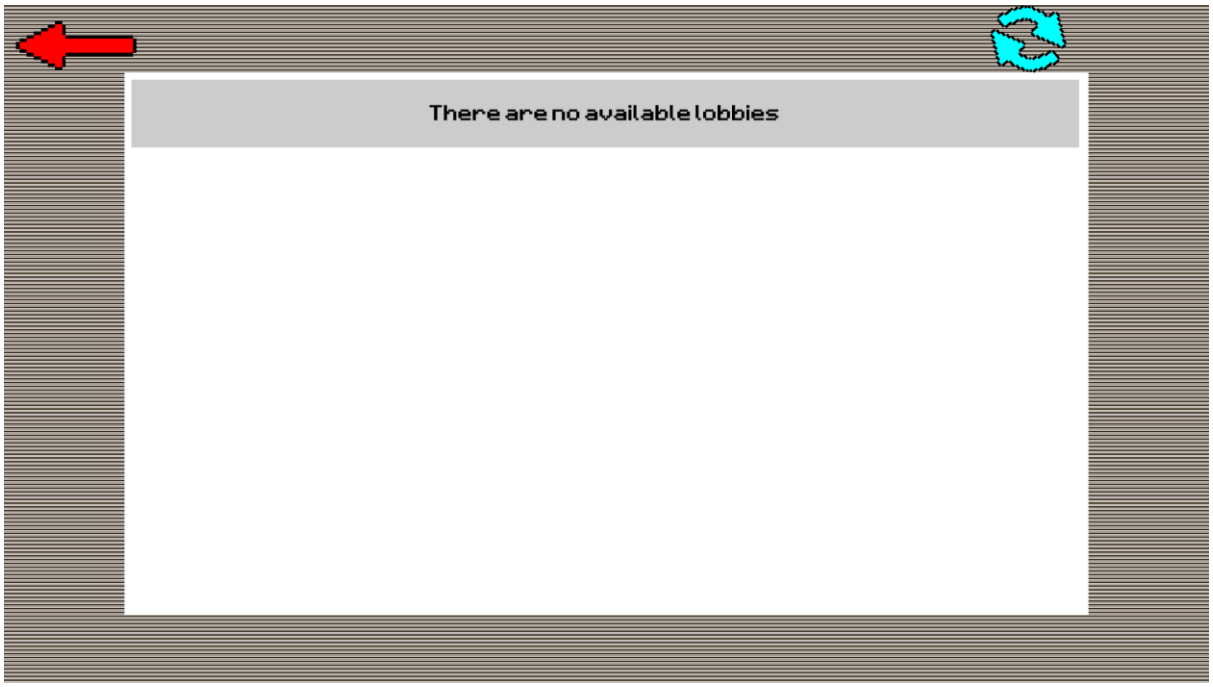
Host's screen when someone wants to join their lobby.



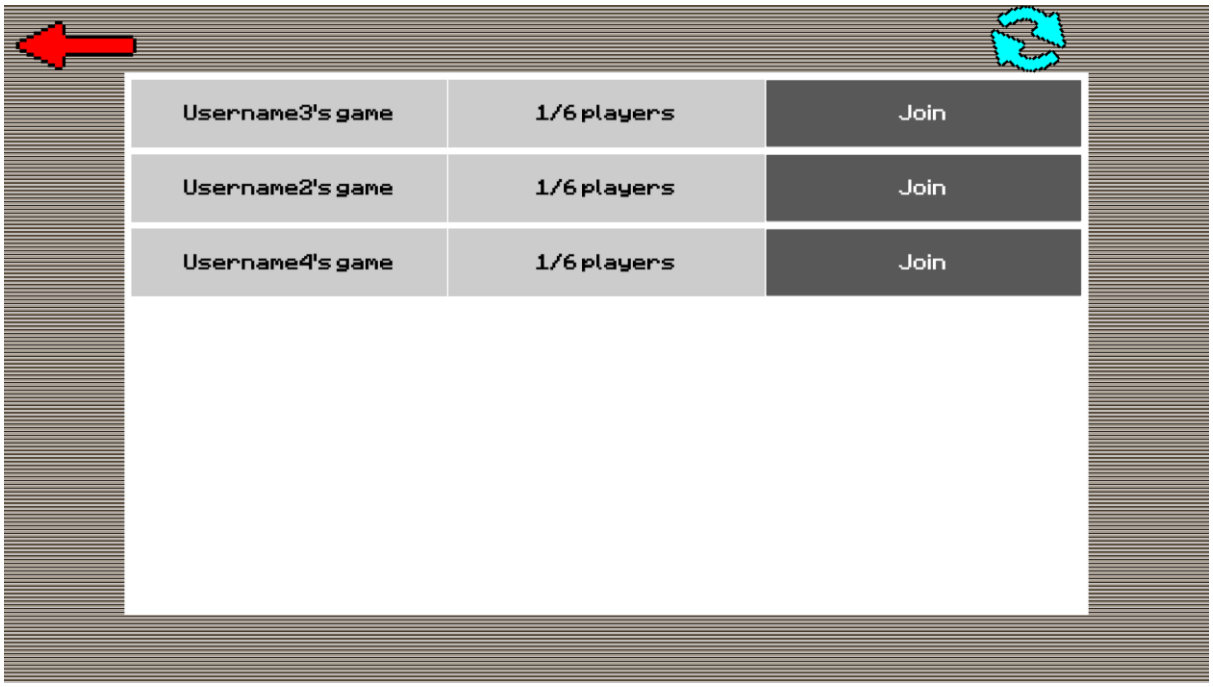
Lobby with 4 players. Since there are no players in the centre and there are players on both teams, the start button is no longer greyed out.

Character selection now has a timer and only asks for 2 characters from this user since they have a teammate.

Join



Join screen with no available lobbies.



Join screen with some lobbies. The table can scroll like the gallery and character selection except it is vertical rather than horizontal.

Testing

I did extensive testing while programming the game during each section as I believe that this is the most efficient way to test. Due to this, most if not all the tests should be successful.

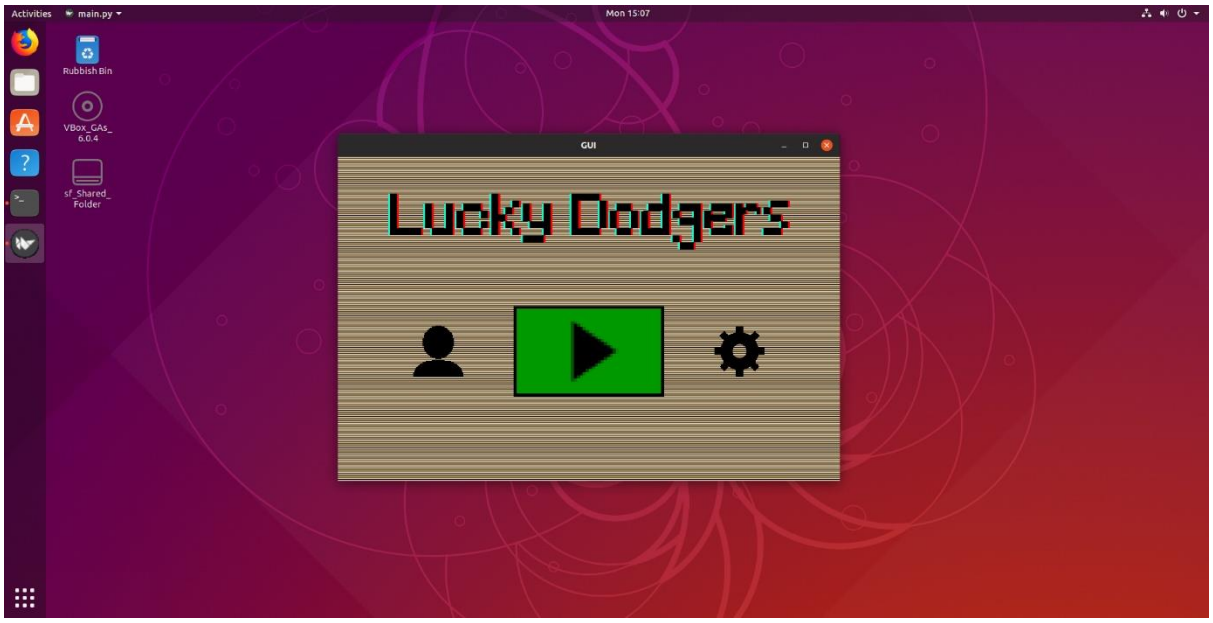
Test 1

Test No.	Objective No.	Description	Expected Outcome
1	1	Run the game on Windows, Linux and Android.	Should work on all of the platforms with no issues.

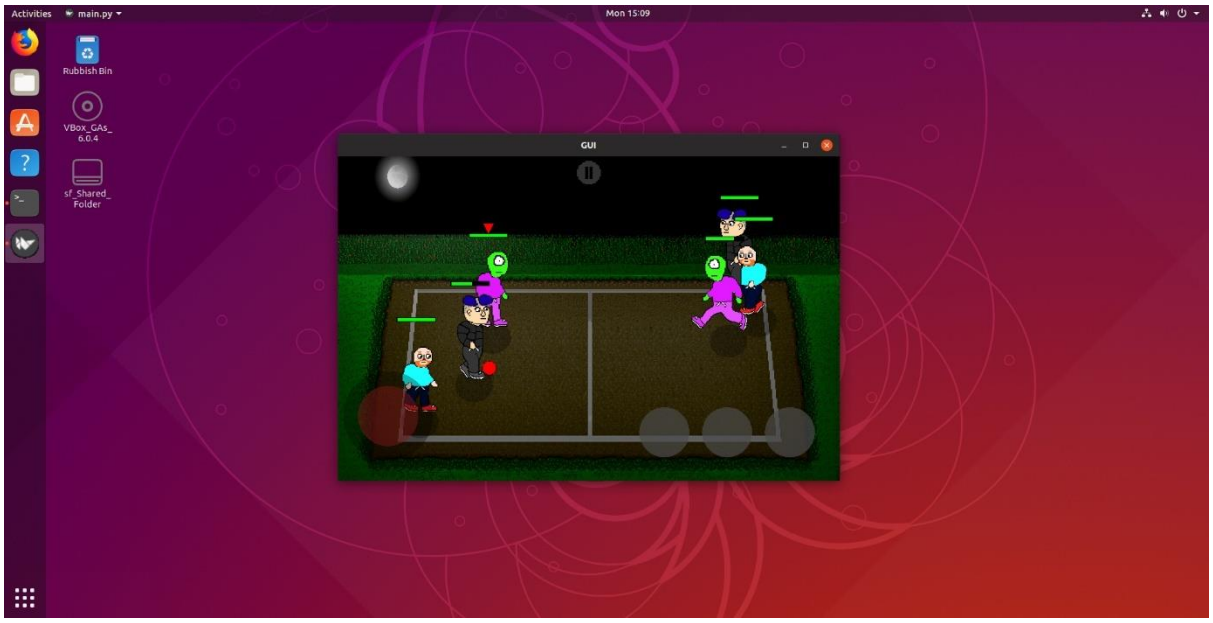
Android:

Windows:

This was done on an Ubuntu virtual machine



Within the game:



Test 1 was successful since the game ran on Windows, Linux and Android.

Test 2

Test No.	Objective No.	Description	Expected Outcome
2	2	Let the client use the user interface and get feedback.	The client should find the UI intuitive.

Client feedback: “The design was pretty good. I liked the animations and the buttons looked nice. Some of the buttons weren’t obvious in what they did but overall it was good.”

I would consider this a success for test 2.

Test 3

Test No.	Objective No.	Description	Expected Outcome
3	2	Test the time it takes to transition between screens.	Should be under a second.

I added a line to print the time when a button is pressed and when the screen has finished changing. This is done with the line:

```
print(time.time())
```

Here is the result:

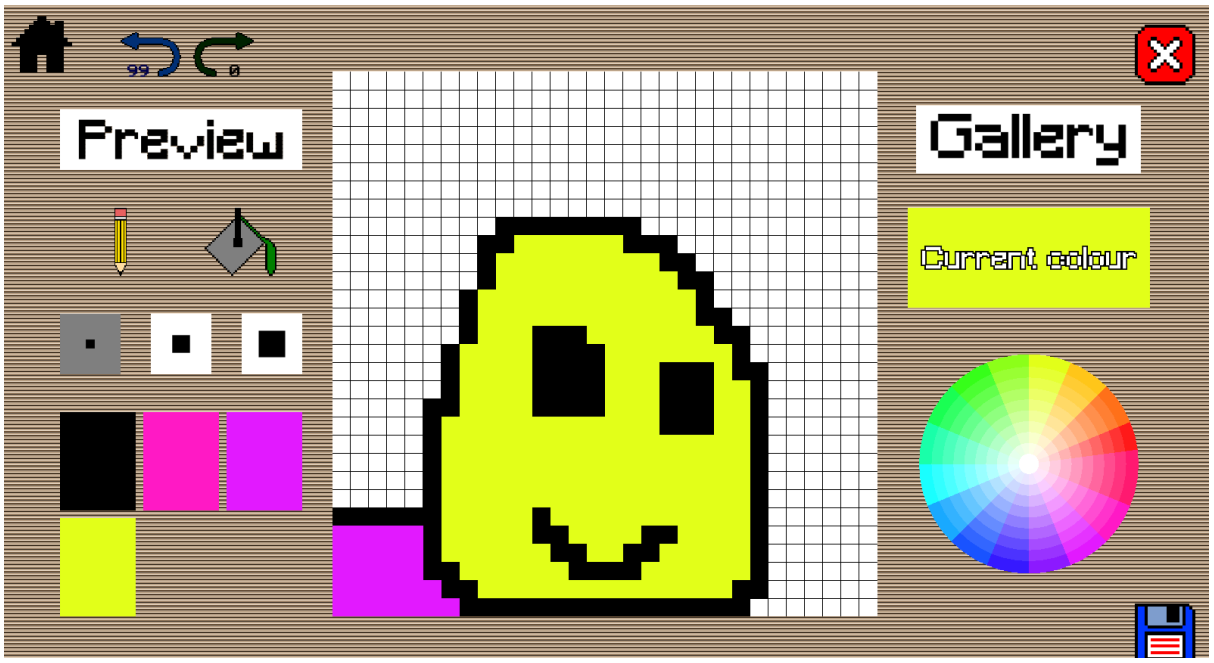
```
1578324117.6499
1578324118.0510354
```

This is in seconds so the time it took to switch screens was very close to 0.4 seconds. This is under a second so test 3 is a success.

Test 4

Test No.	Objective No.	Description	Expected Outcome
4	3	Test drawing new faces for multiple characters.	The new faces should be applied to the characters.

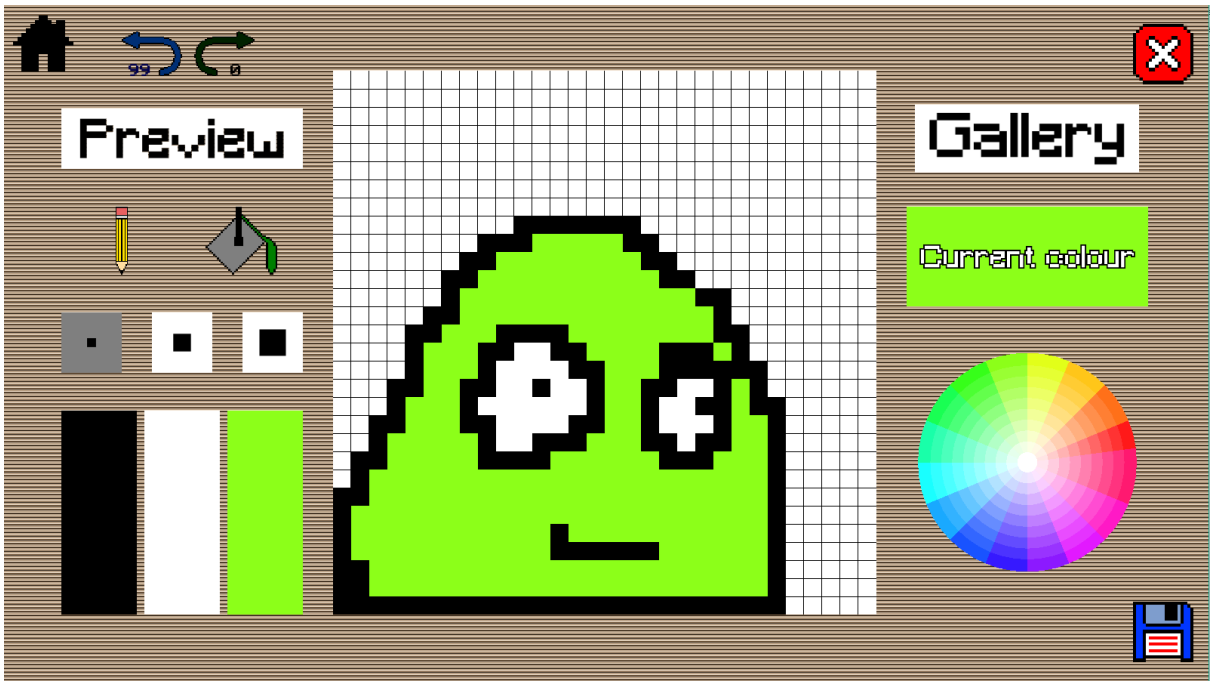
Test character 1:



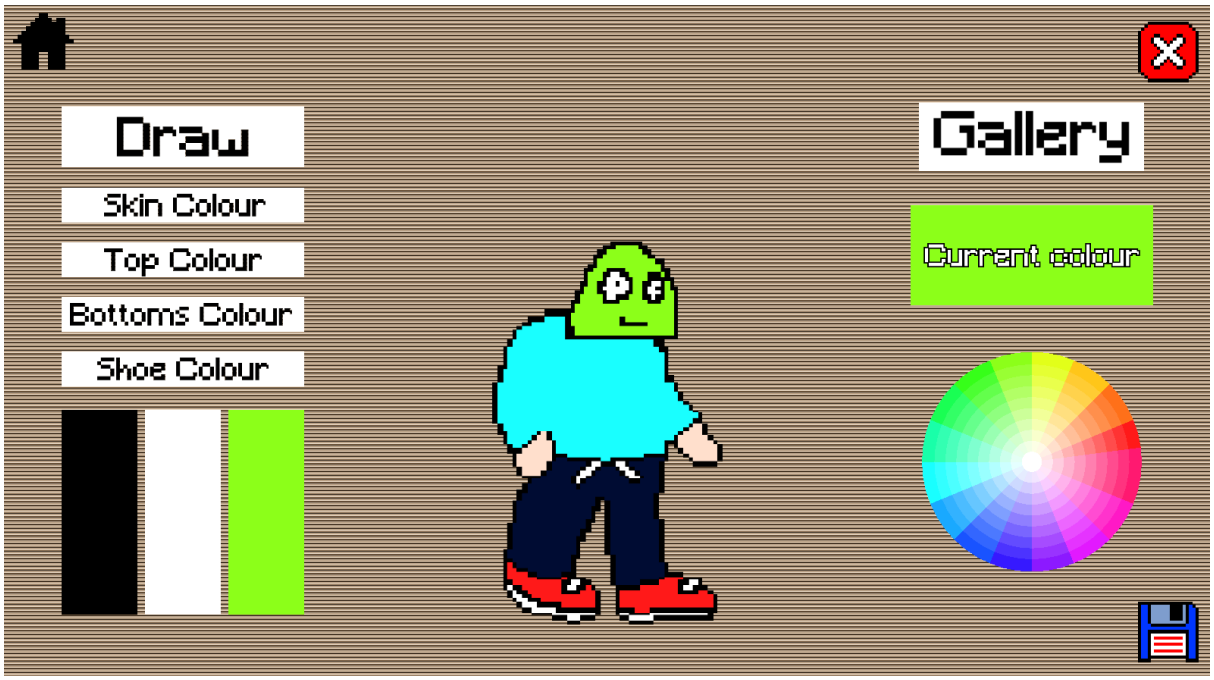
I made a simple head for the character. Here is the result:

The character creation worked fine for this character.

Test character 2:



Made another simple head. Here is the resulting character:



As you can see, the character creation worked for two different heads on two different bodies. Test 4 is a success.

Test 5

Test No.	Objective No.	Description	Expected Outcome
5	3	Let the client use the canvas and get feedback.	The client should find the canvas responsive.

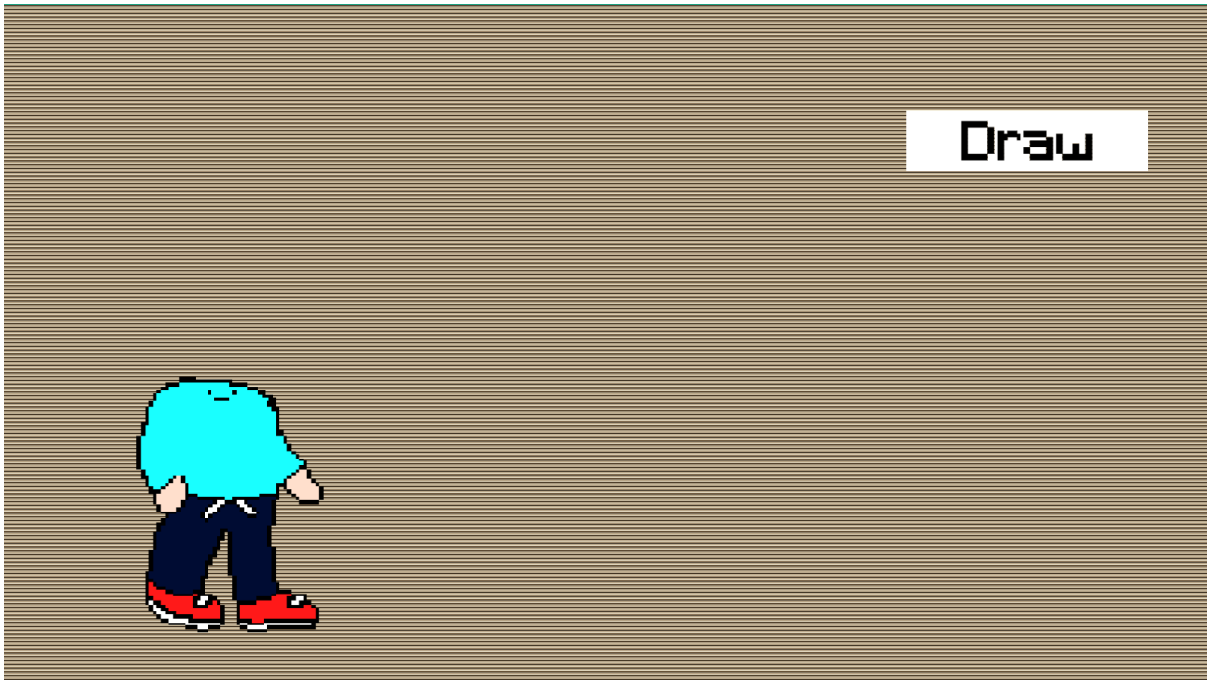
Client feedback: “The canvas was perfectly responsive. Wherever I touched, it worked fine.”

Test 5 was a success.

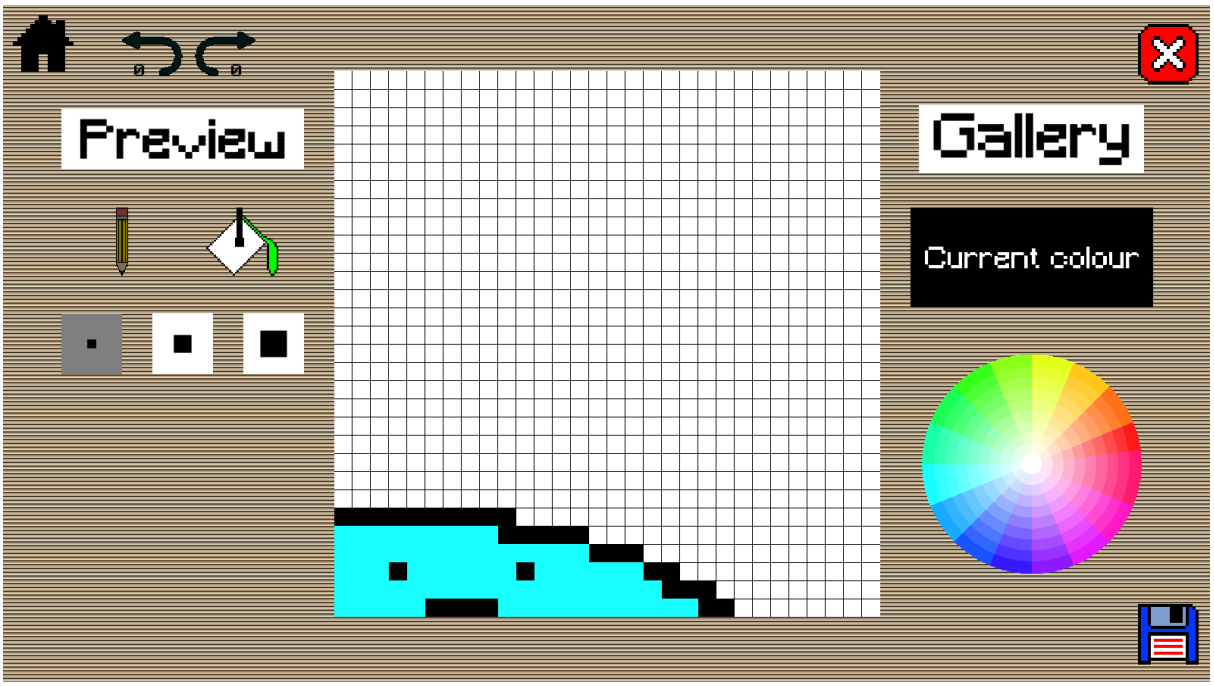
Test 6

Test No.	Objective No.	Description	Expected Outcome
6	4	Test loading a custom character.	Character should be able to be loaded and reused.

I created a new character and saved it. It is in the gallery now:



After pressing on the character and clicking load:

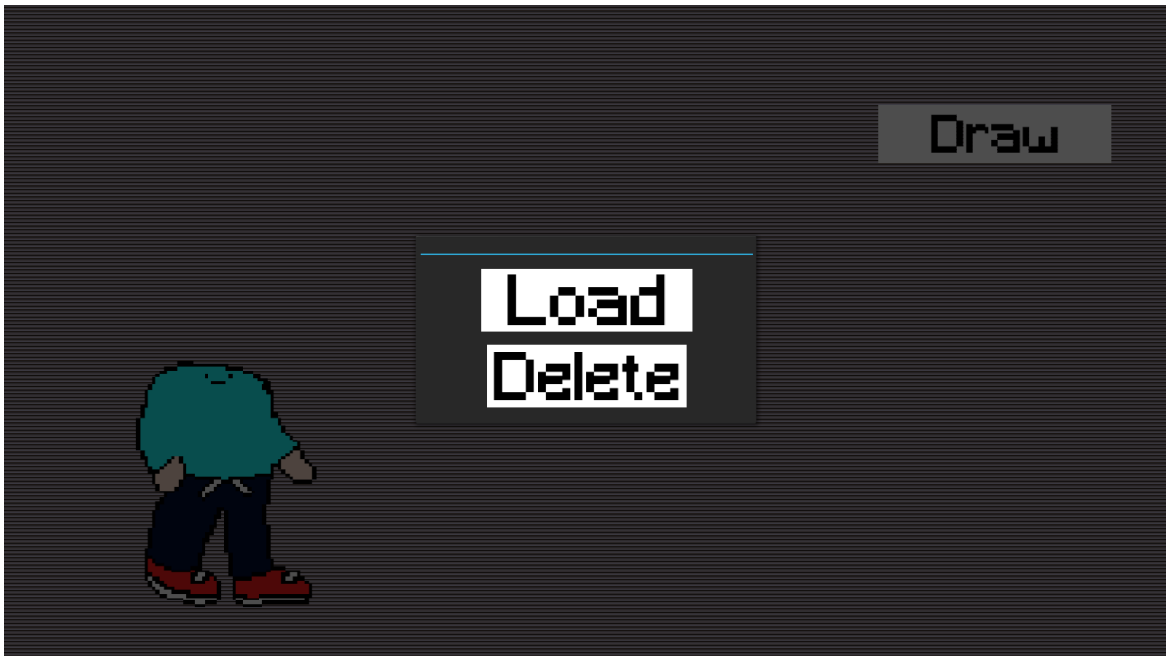


The character is loaded, and I can edit it. Test 6 was a success.

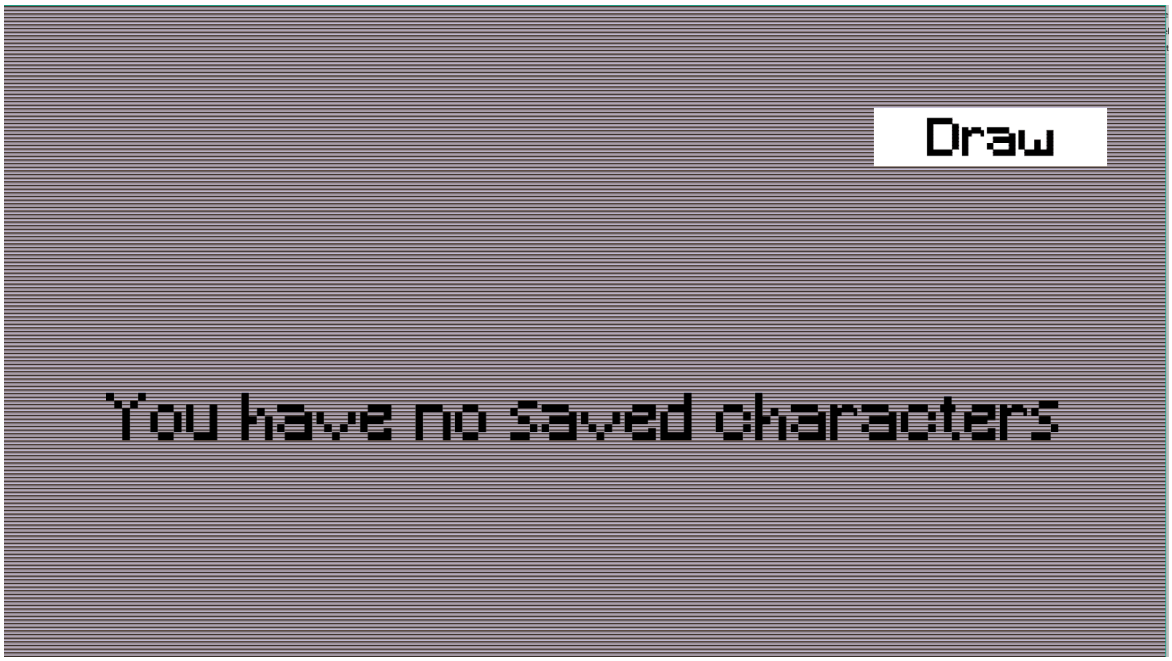
Test 7

Test No.	Objective No.	Description	Expected Outcome
7	4	Test deleting a custom character.	Once deleted, character should be completely gone from the game.

Before deleting the character:



After deleting the character:



The character is completely gone from the game so test 7 is a success.

Test 8

Test No.	Objective No.	Description	Expected Outcome
8	5	Try previewing a new character.	Old character with new character's head should appear.

Test 8 was demonstrated a success in test 4 (the screenshots showed previews of characters).

Test 9

Test No.	Objective No.	Description	Expected Outcome
9	5	Test saving a new character.	Save should take less than 10 seconds.

I used the same method for timing as I did in test 3. I created a character with the canvas completely full and with all the body parts' colours changed to get the slowest possible save. This was the result:

```
1578328333.2857416
1578328334.9075158
```

The time it took was just over 1.6 seconds. This was hugely less than expected although the test was done on my PC so it was faster than it would be on Android.

When testing on Android, I had to test just by counting because there is no print output. I counted the save to be about 5 seconds. This is slower than on Windows but still a lot less than expected.

Test 9 was successful because the save time was less than expected.

Test 10

Test No.	Objective No.	Description	Expected Outcome
10	6	Write a program to test the hash that tests the speed and number of collisions.	One hash should take half a second, there should be very few collisions.

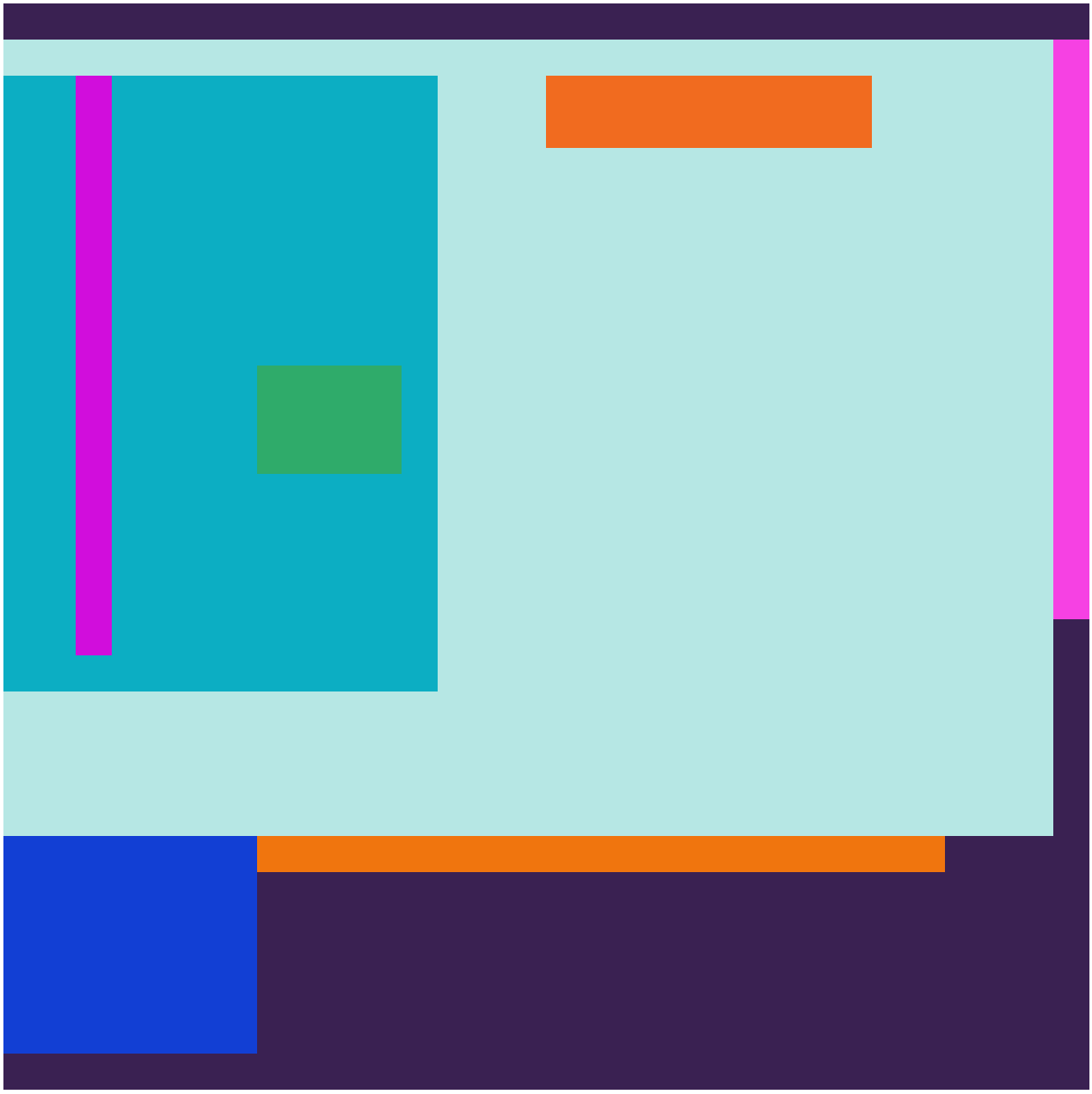
Using the method for timing from before isn't good enough for this because the hash is too fast. The result ends up being inconsistent. To find a more precise value I had to use python's inbuilt timeit function to see how long the hash takes. Here is the result:

```
0.0001340999999999981
```

The hash took a tenth of a millisecond to complete. This means that the hash has no noticeable effect on saving time at all.

To test how collision-proof the hash was, I had to write a program which generates data that resembles drawings by users.

Here is an example image made by the program:



The number of different shapes within an image generated can vary from 1 to 30.

Here is the hash testing program:

```
from image_editing import hash_image
from random import random, randint
import numpy as np
import cv2


hashes_dict = {}  # Use a dictionary because finding duplicates is quick.
for i in range(1000000):  # Generate a million different images.
    section_colors = {'skin_colour': [random(), random(), random(), 1.0],  # Pick random colours for each section.
                      'top_colour': [random(), random(), random(), 1.0],
                      'bottoms_colour': [random(), random(), random(), 1.0],
                      'shoe_colour': [random(), random(), random(), 1.0]}
    images = []
    shapes = randint(1, 30)  # Create up to 30 random shapes and then combine them to make an image.
    for j in range(shapes):
        colors = []
        # The 3rd dimension is for colour, it is 1 instead of 4 for rgba because I want each value (r, g and b) to be
        # random so I have to do them separately.
        shape = [randint(1, int(30)), randint(1, int(30)), 1]
        for k in range(3):  # Create an array for r, g and b.
            colors.append(np.full(shape, randint(0, 255)))  # Create an array with random 3rd dimension.
        colors.append(np.full(shape, 255))  # Create the alpha bit of the array. We want it 255 to make the shape opaque
        images.append(np.concatenate(colors, axis=2))  # Combine all the colours together to make one shape.
    canvas = np.zeros((30, 30, 4))  # Create a blank 30x30 image array.
    for image in images:
        if image.shape[0] == 30:
            r_y = 0
        if image.shape[1] == 30:
            r_x = 0
        r_y = randint(0, 30 - image.shape[0])  # Make a random position for the shape to go.
        r_x = randint(0, 30 - image.shape[1])
        canvas[r_y:r_y+image.shape[0], r_x:r_x+image.shape[1], :] = image  # Put the image in the canvas.
    h = hash_image(canvas, section_colors, "kirk")  # Hash the image.
    if h in hashes_dict:  # If there is a collision, end the program.
        print("FAIL")
        break
```

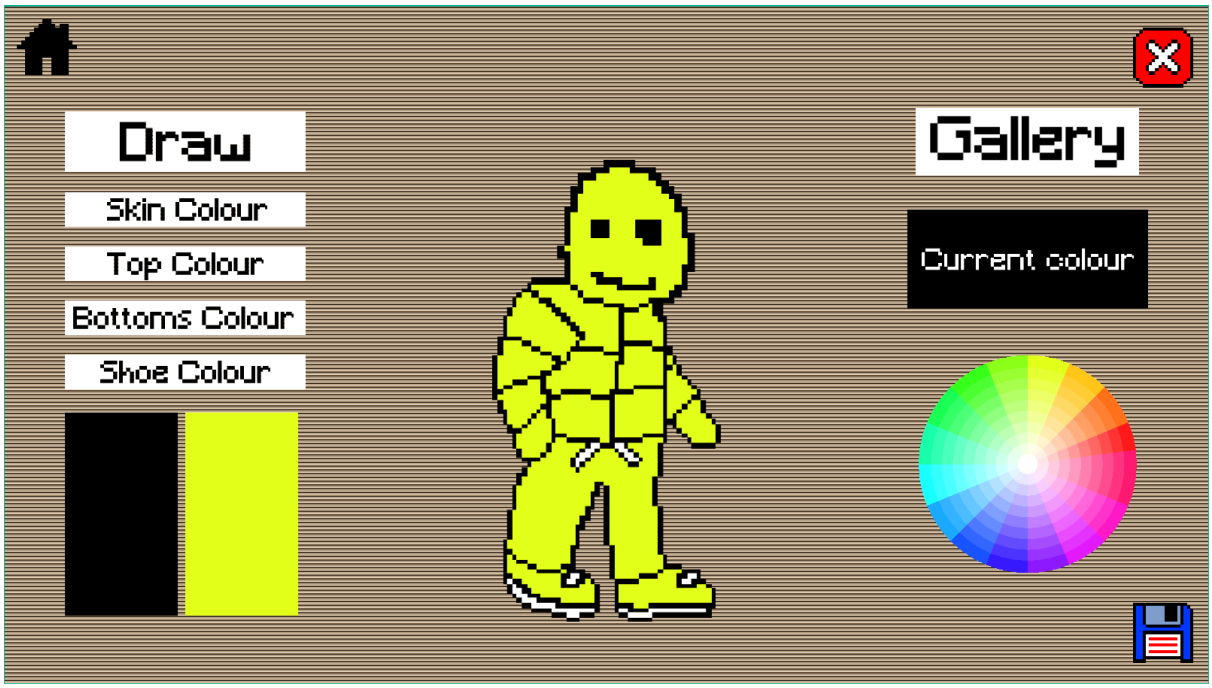
```
    else:
        hashes_dict[h] = "hashed"
```

When run, the program generated zero collisions. This means that the hash algorithm is very good especially considering that there were a million different images generated. Due to this and the hash function being very fast, test 10 is a success.

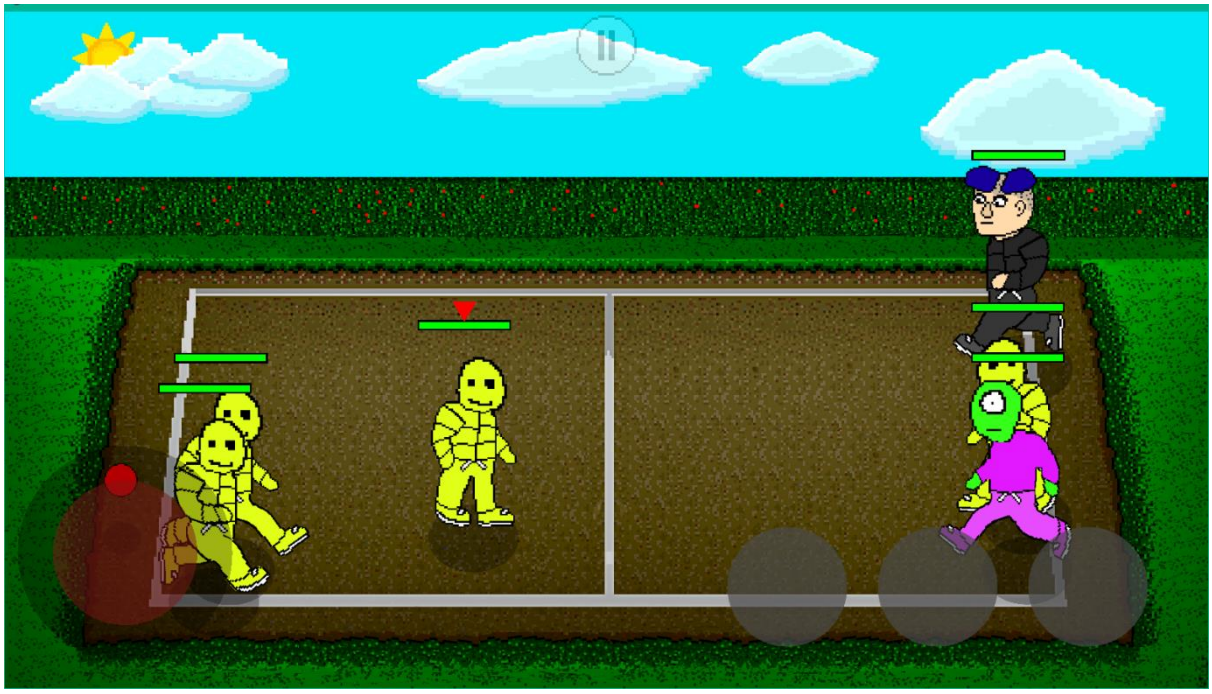
Test 11

Test No.	Objective No.	Description	Expected Outcome
11	7	Play a game with custom characters and pre-made characters.	The custom characters should work with no inconsistencies.

Made a new character:



When used in-game there were no issues found with the character. Test 11 is a success.



Test 12

Test No.	Objective No.	Description	Expected Outcome
12	8	Observe the frame rate and game speed throughout a game.	The frame rate should be changing but the game speed should be constant.

For this test I printed out fps at every tick. Game speed has to be judged by eye because there is no way to calculate it. Here are the frame rates:

```
157
158
157
158
157
156
157
156
157
156
157
158
157
158
159
158
159
158
159
160
159
160
161
160
161
160
```

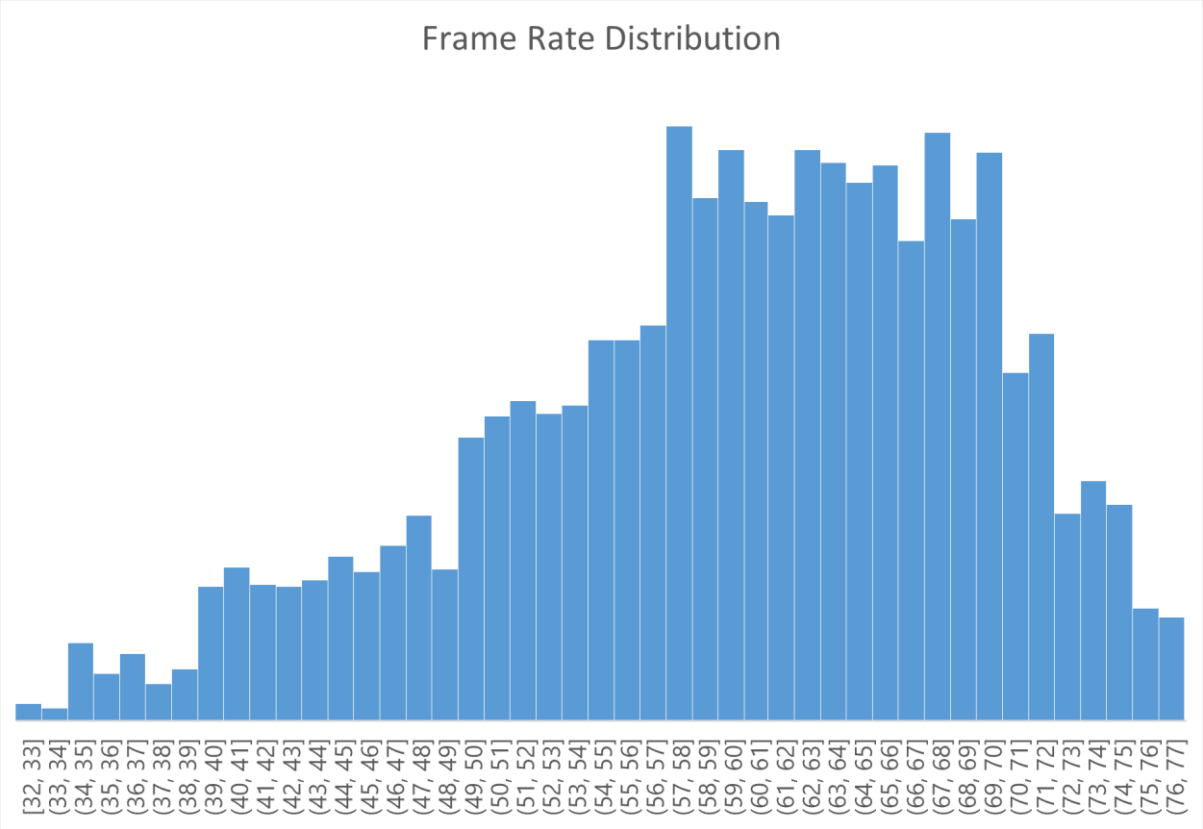
As you can see, the frame rate is constantly adapting but is consistently around 160. The game speed seemed to be constant the whole time which is expected especially when the fps is consistent since slowdown wouldn't occur.

Due to this, test 12 is a success.

Test 13

Test No.	Objective No.	Description	Expected Outcome
13	9	Run the game on my phone (Google Pixel XL) and observe the frame rate.	Should run at between 30 and 60 fps.

For this test, I had to store the frame rates in a csv file since there is no console to print to. I then made a histogram from the values in the csv file using Excel to visually represent the data. Here it is:



The frame rate followed a somewhat normal distribution with a heavy negative skew. This is not surprising because frame rate should be fairly consistent with a hard limit for the maximum but not for the minimum. The lowest fps was 32 and the highest was 77. The median was 61 fps. This is very impressive considering my phone is over 3 years old. Therefore, I would consider test 13 a success.

Test 14 and 16

Test No.	Objective No.	Description	Expected Outcome
14	10	Let the client play the game and get feedback.	The client should enjoy the game.
16	12	Let the client play the game and get feedback.	The client should find the AI fair to play against.

Client feedback: “That was really fun. Catching the ball was quite difficult but it was very satisfying to do. The AI was hard to beat at first but after a few goes I could beat it. It was nice to have a challenge because usually in games, the AI is way too easy to beat and it gets boring.”

I would consider both tests successful from this feedback.

Test 15

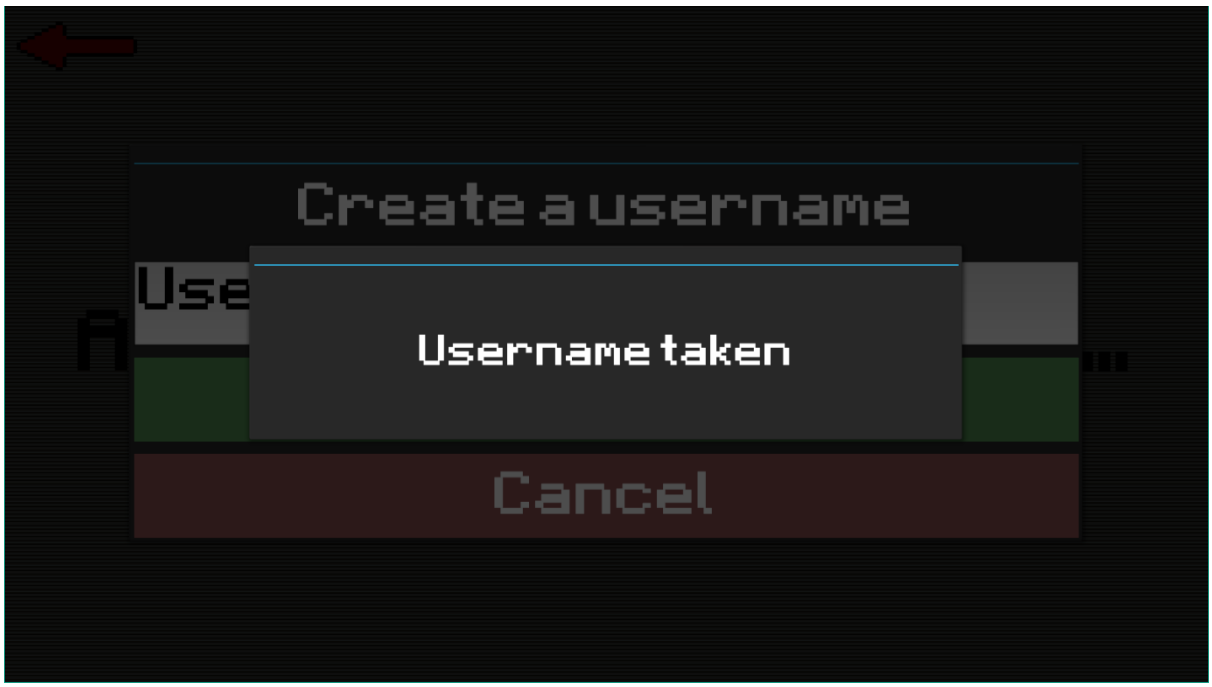
Test No.	Objective No.	Description	Expected Outcome
15	11	Complete all the other tests.	No major bugs should occur during any of the tests.

There were only some minor bugs which I quickly fixed in some of the tests so test 15 was successful.

Test 17

Test No.	Objective No.	Description	Expected Outcome
17	13	Connect to the server and create a username.	If the username is not taken, the server should accept it.

If username is taken:



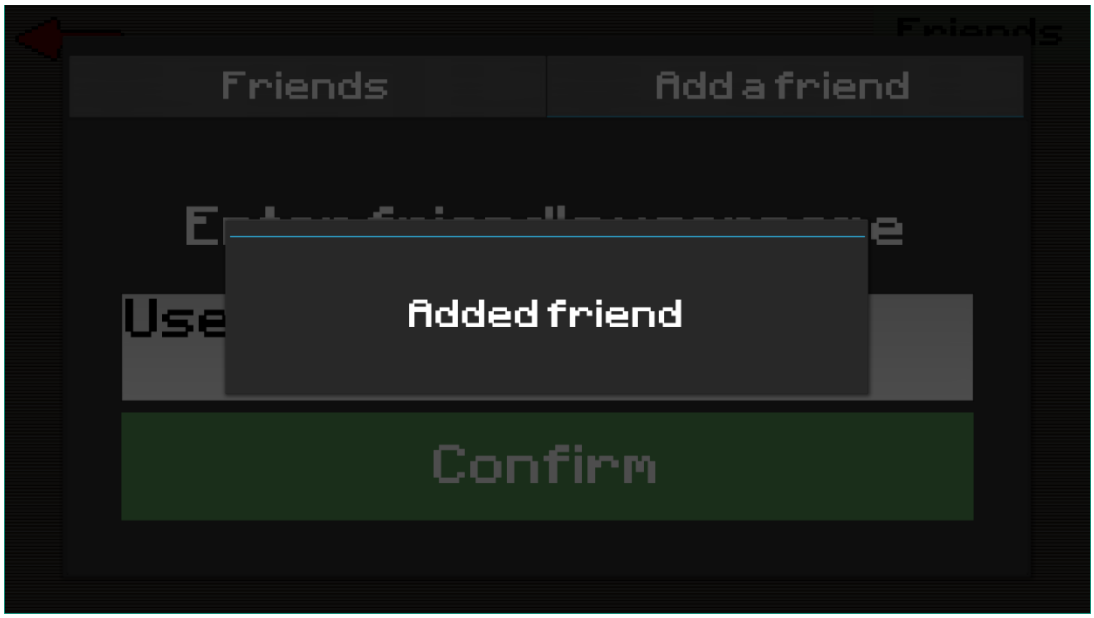
If username isn't taken:

Test 17 was successful.

Test 18 and 19

Test No.	Objective No.	Description	Expected Outcome
18	14	Add a friend.	The friends should now be in the friend's list.
19	14	Check the friend's list.	It should show all of the user's friends and their statuses.

Add a friend:



Friends list now looks like this after adding two friends:

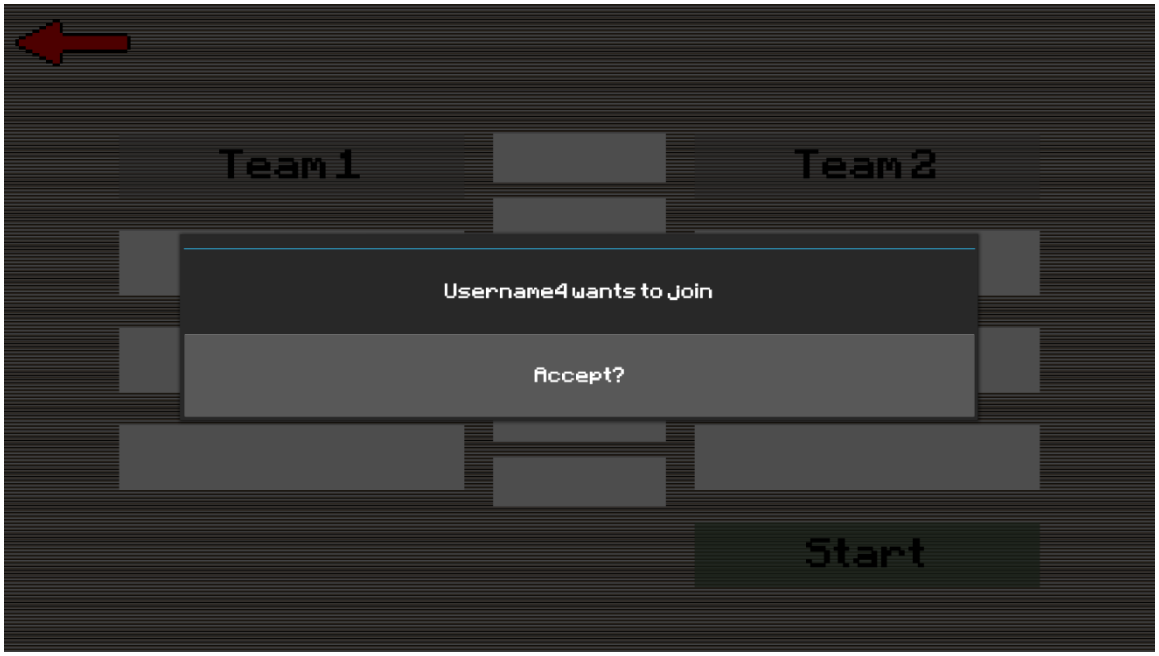
Tests 18 and 19 are successful.

Test 20

Test No.	Objective No.	Description	Expected Outcome
20	15	Join a friend from the friend's list.	A request should pop up on the host of the game's screen.

Here is the friends list:

After pressing the join button next to Username2, this appears on Username2's screen:



Test 20 is a success.

Test 21, 22 and 23

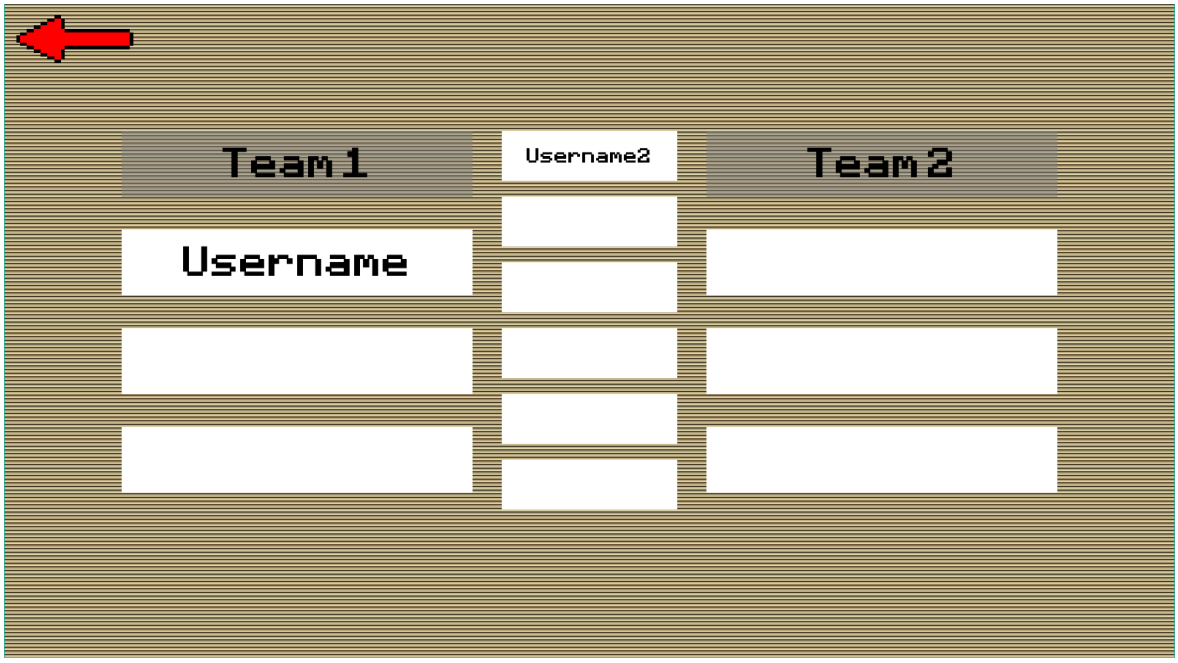
Test No.	Objective No.	Description	Expected Outcome
21	16	See all the available lobbies.	A table of lobbies should be generated.
22	17	Host a lobby.	The user should be in a lobby on their own until someone joins.
23	18	Use a second instance of the game to request to join a hosted lobby.	The host should be able to accept or deny the request.

These three tests were completed and succeeded in the user interface screenshots section in technical solution.

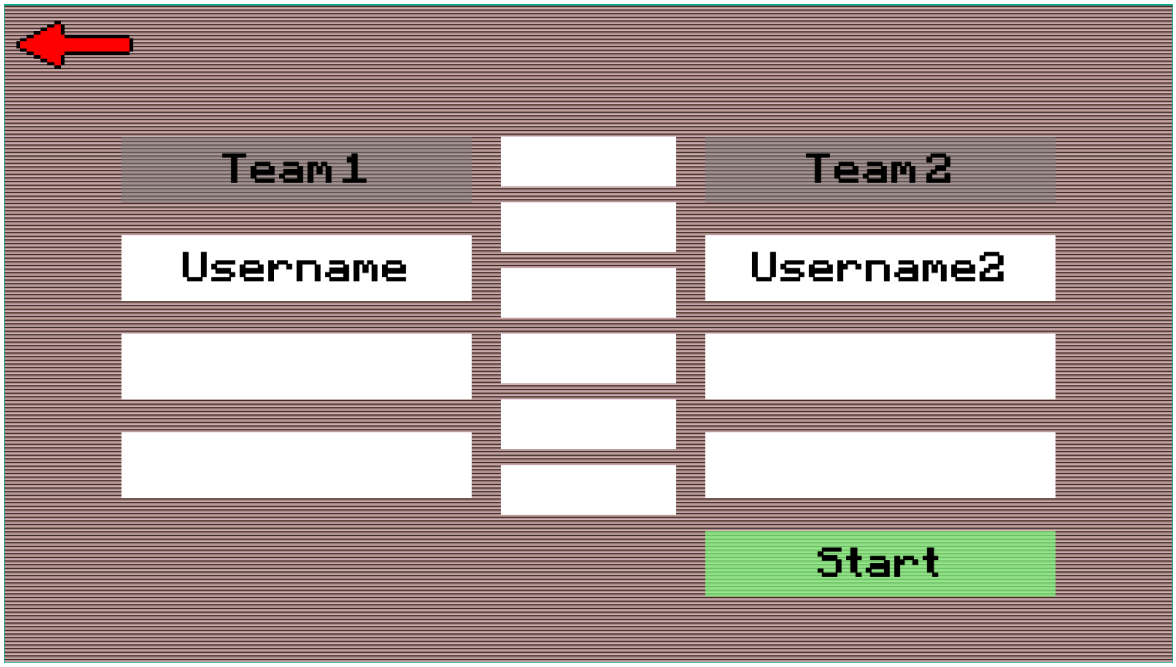
Test 24

Test No.	Objective No.	Description	Expected Outcome
24	19	Change teams within the lobby screen.	The teams should be updated on the screen of everyone in the lobby.

Before changing teams. This is Username2's screen:



After changing teams on Username2's device. This is Username's screen:

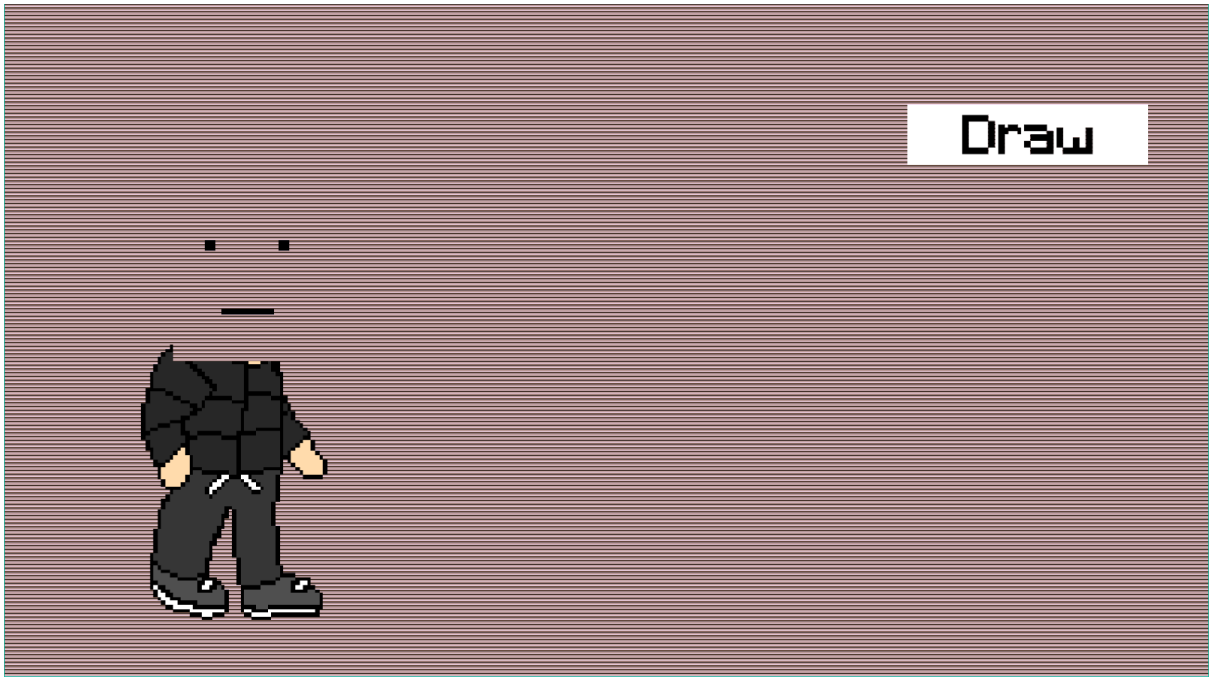


Test 24 is a success.

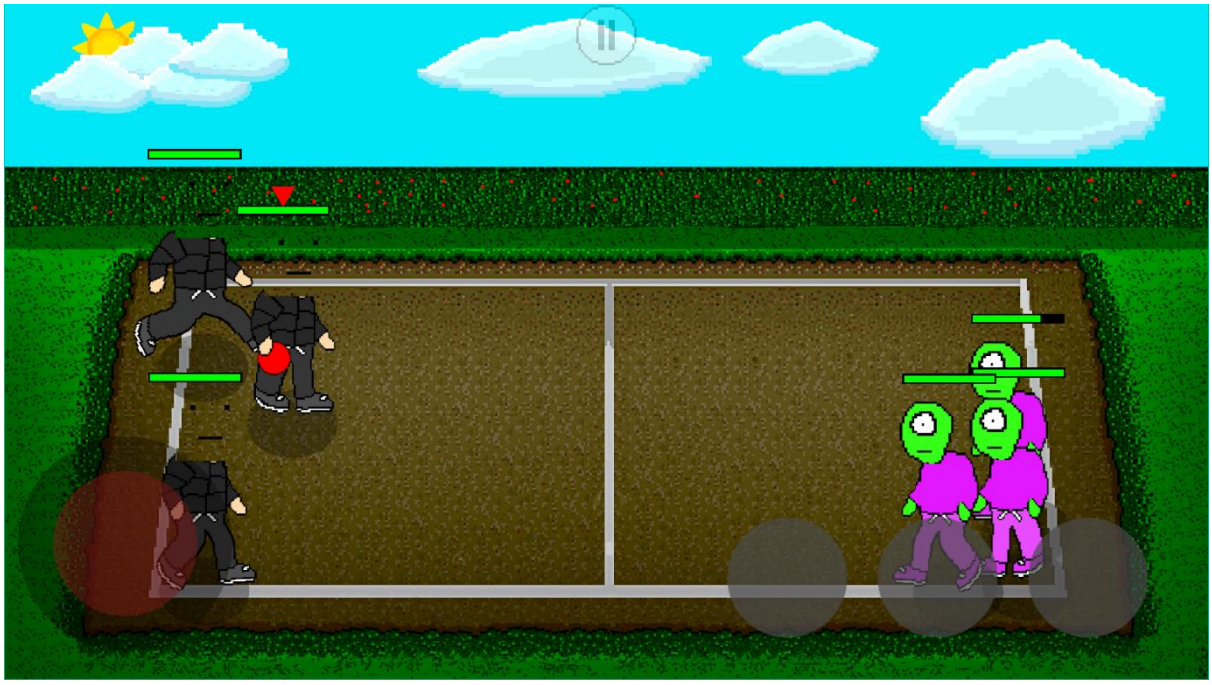
Test 25

Test No.	Objective No.	Description	Expected Outcome
25	20	Play a game with a friend who made a custom character.	The custom character should appear on both player's screens.

Created a character:



Added a friend and started a game with them:



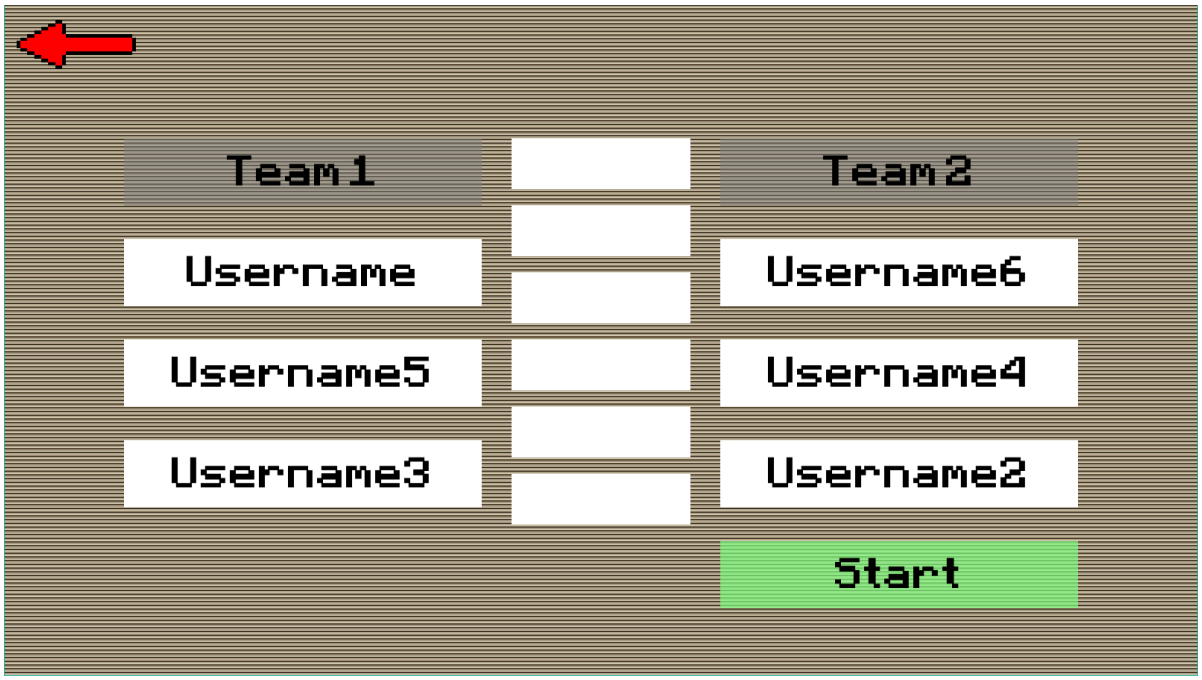
This is on the other player's screen hence the cursor being above the player on the right. As you can see, the custom character is on this screen as well.

Test 25 is successful.

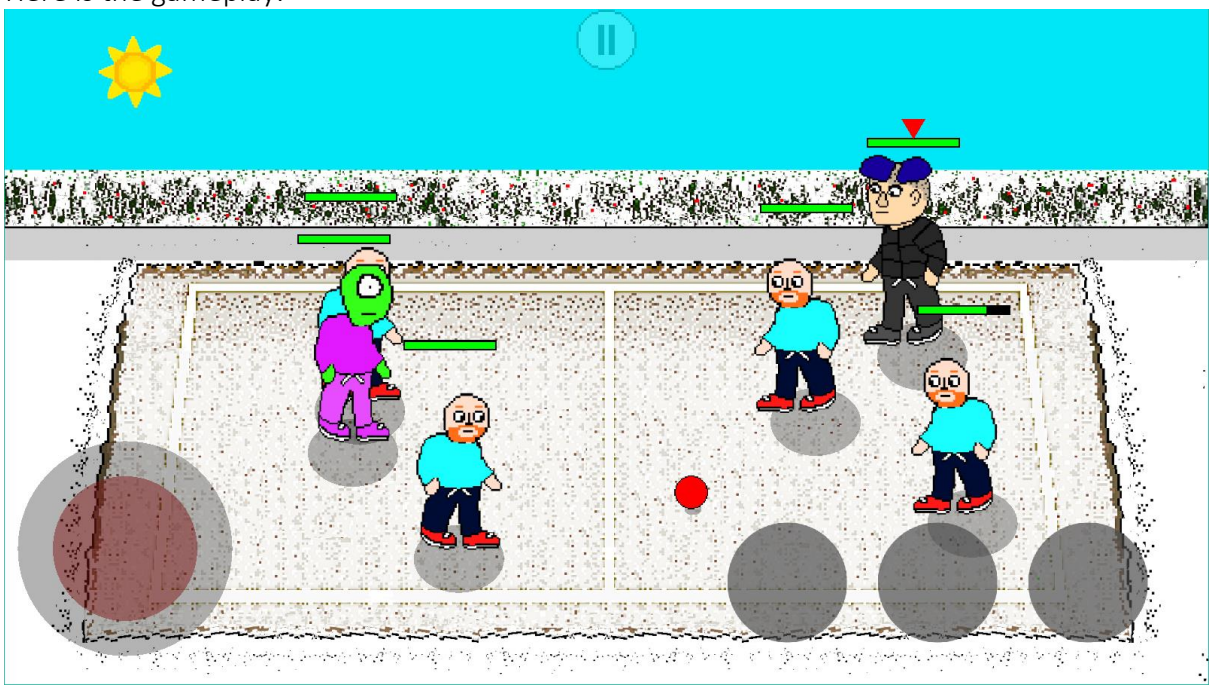
Test 26

Test No.	Objective No.	Description	Expected Outcome
26	21	Start a match with 6 instances of the game.	The game should run normally with all six players being able to play.

Here is the full lobby:



Here is the gameplay:

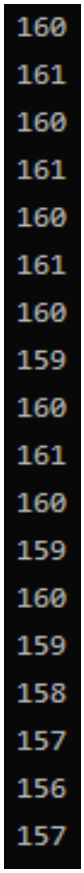


The gameplay worked perfectly fine with each device controlling one character.

Test 27

Test No.	Objective No.	Description	Expected Outcome
27	22	Compare frame rates in online and offline.	They should be about the same.

Using the same method as test 12, here are the results:

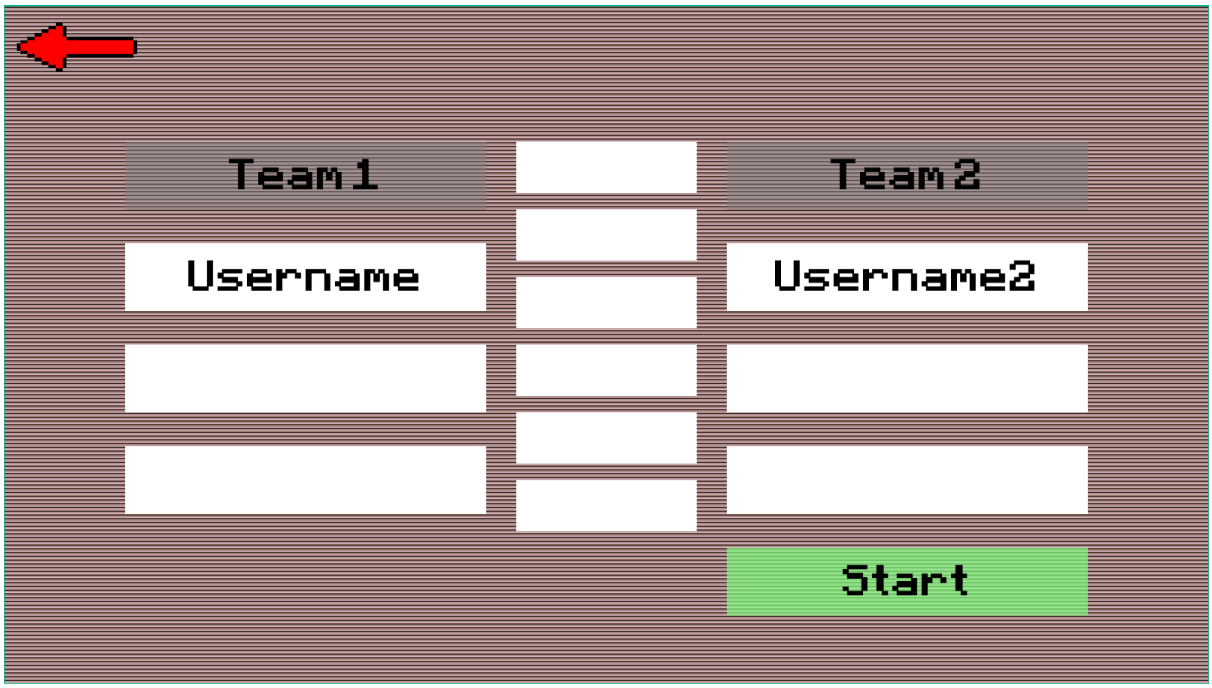


Once again, the fps stayed at around 160. This demonstrates that online multiplayer doesn't have a noticeable effect on how well the game runs. Test 27 is a success.

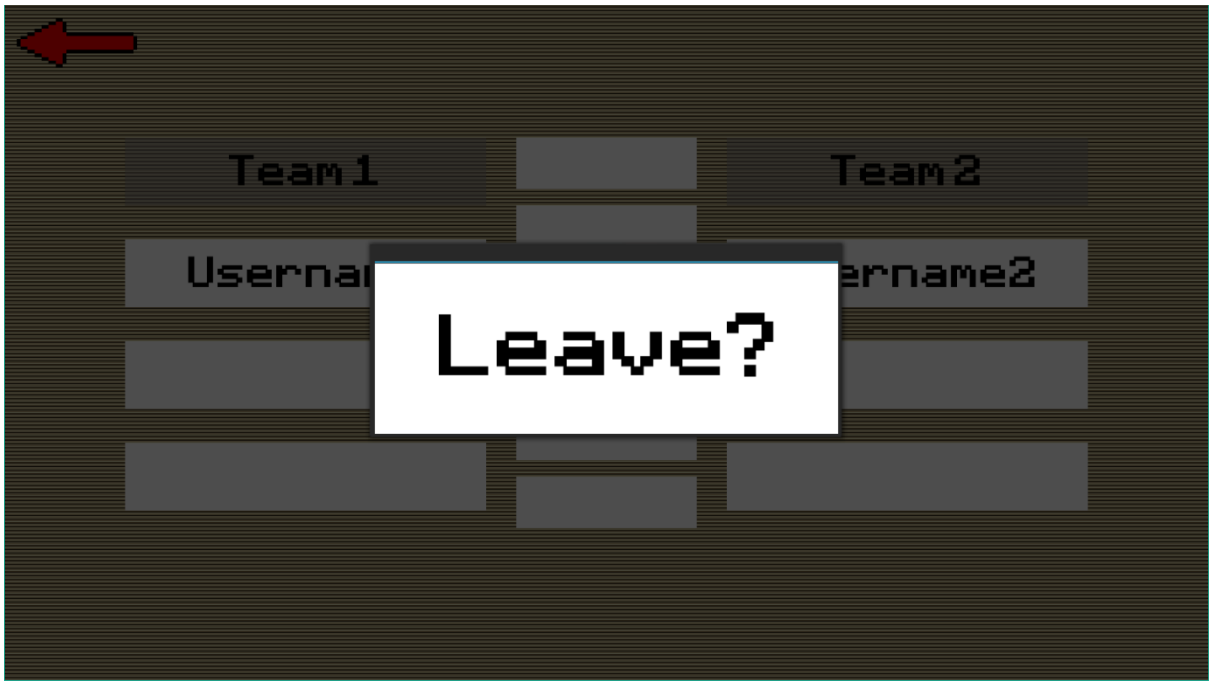
Test 28

Test No.	Objective No.	Description	Expected Outcome
28	23	Disconnect a device while connected to the server.	The server should notify any players interacting with the disconnected player.

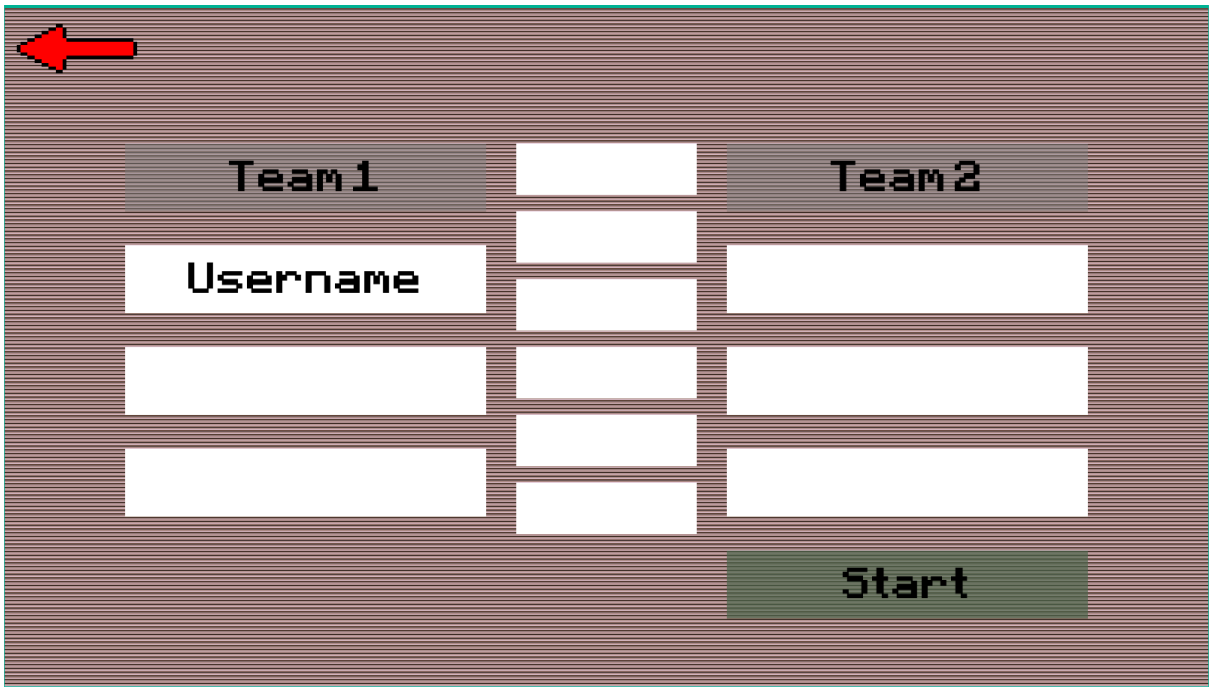
Lobby from Username's screen:



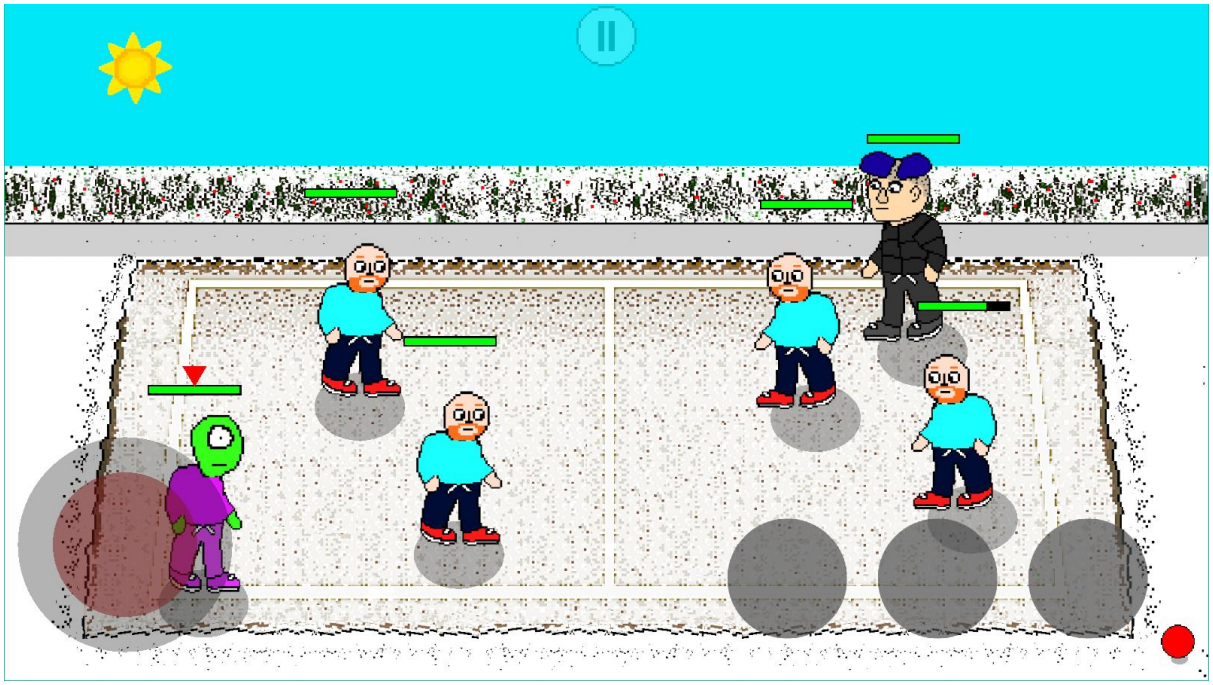
Leave on Username2's device:



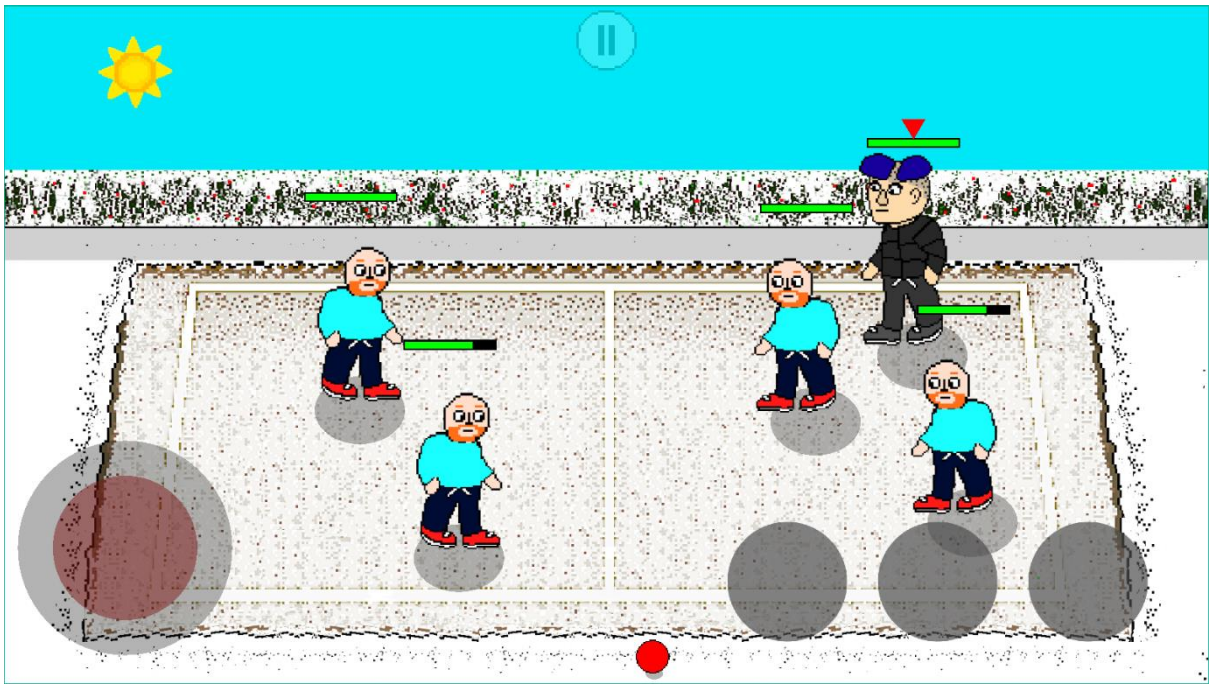
The lobby is updated on Username's screen:



During an online multiplayer game:



When the user disconnects, the server sends a message to the other players that the character(s) controlled by the user have been knocked out. Here is what it looks like now:



Test 29

Test No.	Objective No.	Description	Expected Outcome
29	24	Disconnect and reconnect quickly.	The player should be able to resume playing.

I tested this on my phone by switching from Wi-Fi to mobile data while in a lobby. I was able to continue being in the lobby and could change teams again once I reconnected. Test 29 was a success.

Video Evidence

Single-Player

Here is the link to a video showing the single-player mode in Lucky Dodgers:

https://youtu.be/cyni641Sgag

Multiplayer

Here is the link to a video showing the multiplayer in Lucky Dodgers:

https://youtu.be/LtwC4kv-2rk

Evaluation

This project has been quite challenging but very enjoyable to complete. I spread out the process over a long time so it didn't feel overwhelming and it was more satisfying to complete. I learnt a lot about many different programming concepts such as client server networking and image manipulation. Additionally, I learned about the importance of optimisation and use of efficient algorithms because I had to make the game run well on slower devices.

Some of the limitations I initially thought would be problematic were avoided fairly well. The performance was good even on older phones and I would consider anything over 20 fps good enough for the game so it would be adequate for even older phones than my own. Also, the saving was surprisingly quick with it only taking about 5 seconds. This is nice because if it took too long, users may think the app had crashed or is not responding.

All of the tests were a success, so I believe that I did a good job at completing the objectives I set.

Client Feedback

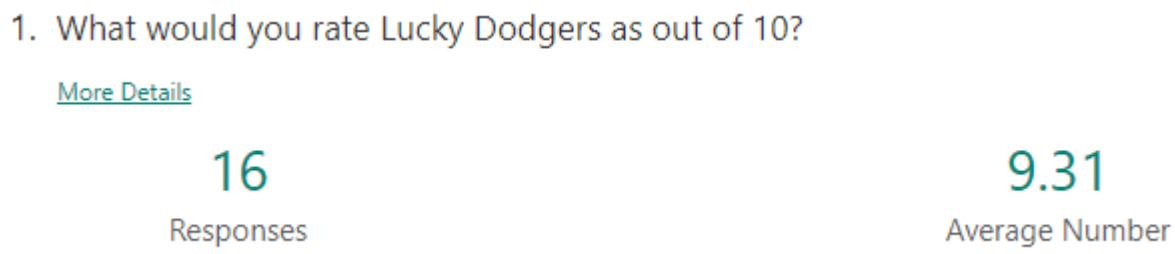
"All the objectives seem to have been met and I'm very happy with the game so I would consider it a success. I was especially impressed with the multiplayer because that was really fun and made it seem like a proper professionally made game.

One thing that I felt was missing was sound. Usually, games have some kind of sound but this didn't have any. This was only a minor issue because I would usually play on my phone while out in public so I would have the sound off anyway."

The feedback was overall very positive, so I am very happy with that. Having sound in the game was something I completely forgot about during the creation of the game. I'm quite disappointed to have forgotten about the sound as this would have been quite easy to implement.

Feedback survey

I sent a quick survey asking to rate the game out of 10 to some friends and family whom I played the game with. There were 16 responses. Here is the result:



An average rating of 9.31 is amazing. I am very happy with this. I did only ask friends and family so the rating is probably a bit higher than it should be, but it is still a very good score.

Improvements

Overall, I am satisfied with how the project went although there are a few improvements I could have made if I had more time:

- The character creation is good, although it would have been nice to be able to change the body as well as the head. I considered this too hard when starting the project, but I believe that with the knowledge I gained over the course of the project, I could now make the character creation have full body changing. This would have been done by having each pixel on the body having an assigned vector movement for each animation. There would still be a few issues with this though. Users could make blank characters, so they are invisible.
- I am running the server from my home PC. This is because I didn't want to spend money on an online service when I could just use my PC. However, this does somewhat limit scalability. Although the server doesn't have to do much processing, with very large numbers of concurrent users, it could start to struggle.
- For drawing on the canvas, I could have added a zoom feature to make it easier for people on smaller screens. This would be quite difficult to implement though because it would require the canvas data structure to be done differently.
- As brought up by the client, I should have added some sound effects to the game. The sounds would have been quite easy to add to the game. The main reason why I didn't add them was because I had always played mobile games without sound and due to this, I didn't even think about it.

Appendix

Main

main.py

This is the program for starting the app and adding a background.

```
from kivy.app import App
from kivy.uix.screenmanager import ScreenManager
from kivy.uix.floatlayout import FloatLayout
from kivy.core.window import Window
from kivy.graphics import Rectangle, Color
from random import uniform
from main_menu import MainMenu

# Create an instance of the ScreenManager class. This is used to switch between screens.
manager = ScreenManager()
# The FloatLayout class is a layout that allows you to place widgets inside it with relative or definite coordinates and
# sizes (size_hint and pos_hint are relative, size and pos are definite).
float_layout = FloatLayout()


class GUI(App):
    def build(self):
        with Window.canvas:
            Color(uniform(.8, 1), uniform(.8, 1), uniform(.8, 1))  # Randomly slightly tint the background
            Rectangle(source="Images/menu_bg.png", size=(Window.width, Window.height))  # Create a background
            Color(1, 1, 1)  # Set the canvas colour back to white
        float_layout.add_widget(manager)
        manager.add_widget(MainMenu(name="Main Menu"))
```

```
        return float_layout


if __name__ == "__main__":
    GUI().run()
```

misc.py

This is a collection of functions and classes that need to be used by many different files.

```
from kivy.uix.behaviors import ButtonBehavior
from kivy.uix.image import Image
import os


# Class that combines the image widget with a button
class ImageButton(ButtonBehavior, Image):
    pass


class Menu(ImageButton):  # This is a button that returns to the main menu
    def __init__(self, direction):
        super().__init__()
        self.pos_hint = {"x": .01, "top": .99}
        self.size_hint = (0.05, 0.1)
        self.source = "Images/home.png"
        self.on_press = lambda: switch("Main Menu", self.parent.parent.manager, direction)


# Function for switching between screens. If the screen hasn't been created yet, create it.
def switch(screen, manager, direction, screen_type=None):
    if not manager.has_screen(screen):
        manager.add_widget(screen_type(name=screen))
    manager.transition.direction = direction
    manager.current = screen


def get_characters():  # Function that finds all the saved characters and creates a dictionary with preview images.
    characters = {}
```

```
    character_folders = next(os.walk("Images"))[1]  # Finds all the subdirectories in Images
    for character in character_folders:
        characters[character] = "Images/" + character + "/preview.png"
    return characters


def set_color(widget, color):
    widget.color = color
```

main_menu.py

This is the program for the main menu screen.

```
from kivy.uix.screenmanager import Screen
from kivy.uix.floatlayout import FloatLayout
from kivy.uix.image import Image
from misc import switch, ImageButton
from controls_changer import ControlsChanger
from creation_station import CreationStation
from play import Play
from kivy.utils import platform


class MainMenu(Screen):
    def __init__(self, name):
        super().__init__()  # Initialises the screen (parent class)
        self.name = name  # self.name is used by manager to differentiate between screens.
        float_layout = FloatLayout()  # Float layout is where buttons and other widgets are placed.
        # Create the title
        title = Image(pos_hint={"x": 0.1, "y": 0.65}, size_hint=(0.8, 0.3), source="Images/title.png")
        # Create a button called play. pos_hint determines the position of the button relative to the size of the
        # window. size_hint is the same but for size. source is the location of the image file of the button.
        play = ImageButton(pos_hint={"x": 0.35, "y": 0.25}, size_hint=(0.3, 0.3), source="Images/play.png")
        options = ImageButton(pos_hint={"x": 0.75, "y": 0.3}, size_hint=(0.1, 0.2), source="Images/options.png")
        creation_station = ImageButton(pos_hint={"x": 0.15, "y": 0.3}, size_hint=(0.1, 0.2),
                                       source="Images/creation_station.png")
        # When the play button is pressed, it calls the function "switch".
        play.bind(on_press=lambda button: switch("Play", self.manager, "down", screen_type=Play))
        options.bind(on_press=lambda button: switch("Controls Changer", self.manager, "left",
                                                    screen_type=ControlsChanger))
        creation_station.bind(on_press=lambda button: switch("Creation Station", self.manager,
                                                             "right", screen_type=CreationStation))
        self.add_widget(float_layout)  # Adding the float layout to the screen.
        float_layout.add_widget(title)
        float_layout.add_widget(play)
```

```
        float_layout.add_widget(options)
        float_layout.add_widget(creation_station)
        if platform == "android":
            start_button = ImageButton()
            start_button.bind(on_press=self.remove_widget)
            self.add_widget(start_button)
```

Character Creation

image_editing.py

This is a collection of functions for changing the images for character creation.

```
from os import walk, makedirs
import cv2
import numpy as np
import pickle
from pathlib import Path
from shutil import copy
import base64


# A function to create a dictionary for the character specified with the keys being the actions of the character and the
# values being lists of the pathnames of each image within that state.
def make_dict(character_name, path="Images"):
    image_dict = {}  # This will be a dictionary containing lists of pathnames
    for subdir, dirs, files in walk(path + "/" + character_name):  # Finds each file in the character name folder
        temp_subdir = subdir.split("/")
        # os.walk uses backslashes but this doesnt work on android so these lines turn everything into forward slashes
        temp_subdir.extend(temp_subdir[1].split("\\"))
        del (temp_subdir[1])
        temp_subdir.remove(path)
        temp_subdir.remove(character_name)
        temp_subdir = "".join(temp_subdir)  # Theses 4 lines are for formatting the path into just the subdir name
        if not temp_subdir:
            continue
        image_dict[temp_subdir] = list()
        for i in range(len(files)):
            # Add the file path to the dictionary
            image_dict[temp_subdir].append("Images/" + character_name + "/" + temp_subdir + "/" + str(i + 1) + ".png")
```

```
    return image_dict


# Basically the opposite of the make_dict function because this uses a dictionary to create folders with images saved.
def make_new_character_directories(character_name, image_dict, drawing, old_character_name):
    char_path = "Images/" + character_name + "/"
    makedirs(char_path)
    saveable_drawing = {}
    for pos in drawing:
        # Turn the values into tuples containing rgba rather than kivy Color objects so it can be pickled
        if drawing[pos].rgba != [0, 0, 0, 0]:
            saveable_drawing[pos] = drawing[pos].rgba
    copy("Images/" + old_character_name + "/info.txt", char_path)
    with open(char_path + "rects_colors.pickle", "wb") as output_file:
        pickle.dump(saveable_drawing, output_file)  # Save the drawing so it can be edited later by the player
    for key in image_dict:
        key_path = char_path + key + "/"
        makedirs(key_path)
        for i, image in enumerate(image_dict[key]):
            save_image(image, key_path + str(i + 1) + ".png")
            if key == "idle" and i == 0:  # This is for making a cropped preview image for the character
                cropped_image = crop_image(image)
                save_image(cv2.resize(cropped_image, (cropped_image.shape[1] * 5, cropped_image.shape[0] * 5),
                                      interpolation=cv2.INTER_NEAREST), char_path + "preview.png")


def drawing_to_image(drawing):  # Turns the initial drawing in the format of a dict with pos as keys to image
    temp_list = []
    # Make the dictionary into a 1D list and multiply each colour value by 255 because kivy uses a 0-1 range for rgba
    for key in drawing:
        temp_list.append(list(map(lambda x: x * 255, drawing[key].rgba)))
    im_arr = np.array(temp_list)  # Make the 1D list into a 1D image array
    # noinspection PyArgumentList
    im_arr = np.flipud(im_arr.reshape(30, 30, 4))  # Reshape the image array to become 3D
    return im_arr
```

```
def get_info(folder, character=True):  # Access the information file for the character
    info_dict = {}
    if character:
        pathname = "Images/" + folder + "/info.txt"
    else:
        pathname = folder
    with open(pathname, "r") as info:
        info_list = info.read().split("\n")
        for i in range(0, len(info_list), 2):
            info_dict[info_list[i]] = info_list[i + 1].split(";")
            try:
                info_dict[info_list[i]] = list(map(int, info_dict[info_list[i]]))
            except ValueError:
                pass
    return info_dict


def crop_image(im_arr, hashing=False):  # Only want to crop width since this is constant except for when hashing
    # Turn the values of the rgba axis into a combination of all of them, so an empty pixel will have a value of 0.
    image_data = im_arr.max(axis=2)
    # Find columns where all pixels has a value > 0, so not an empty column or row
    non_empty_columns = np.where(image_data.max(axis=0) > 0)[0]
    # Create the dimensions containing the first and last non_empty row
    crop_box = (min(non_empty_columns), max(non_empty_columns))
    im_arr = im_arr[:, crop_box[0]:crop_box[1] + 1, :]  # Crop the image array
    if hashing:  # Crop rows as well
        non_empty_rows = np.where(image_data.max(axis=1) > 0)[0]
        crop_box = (min(non_empty_rows), max(non_empty_rows))
        im_arr = im_arr[crop_box[0]:crop_box[1] + 1, :, :]
    return im_arr


def make_image_array(pathname):
    # cv2.imread creates an image array from a pathname. The -1 is to make it keep alpha channel.
    # The slicing afterwards is there because cv2 use bgra rather than rgba so this is converting it to rgba.
```

```
    # IMPORTANT: The image array is in the format [y][x][colour] not [x][y][colour] like expected.
    # x and y are integers and color is a list of 4 integers up to 255.
    im_arr = cv2.imread(pathname, -1)[:, :, [2, 1, 0, 3]]
    return im_arr


def save_image(im_arr, pathname):
    # Slicing required for same reason as in make_image_array
    cv2.imwrite(pathname, im_arr[:, :, [2, 1, 0, 3]], [cv2.IMWRITE_PNG_COMPRESSION, 9])


def image_search(im_arr, head_dimensions):
    # This function finds the uppermost pixel and then calculates using head dimensions, where the head is.
    image_y, image_x = im_arr.shape[:2]
    for y in range(-head_dimensions[0], image_y - head_dimensions[1]):
        for x in range(-head_dimensions[2], image_x - head_dimensions[3]):
            test = (im_arr[y, x] == (0, 0, 0, 255))
            if test.all():
                return [y + head_dimensions[0], y + head_dimensions[1], x + head_dimensions[2], x + head_dimensions[3]]


# Replace the pixels where the head was with the new image
def combine_images(filename, head_dimensions, replacement_im_array):
    im_array = make_image_array(filename)
    head_section = image_search(im_array, head_dimensions)
    im_array[head_section[0]:head_section[1], head_section[2]:head_section[3]] = replacement_im_array
    return im_array


def find_sections(character_name):  # This is only used for new characters upon creation, never called by the program
    character = make_dict(character_name)
    info = get_info(character_name)
    head_dimensions = info["head_dimensions"]
    section_locations = {}
    for key in character:  # Go through each animation in the character directory
        for file in character[key]:  # Go through each image in the animation
```

```
            section_locations[file] = {}
            # Remove the head, np.zeros creates an empty array
            im_arr = combine_images(file, head_dimensions, np.zeros((head_dimensions[1] - head_dimensions[0],
                                                                     head_dimensions[3] - head_dimensions[2], 4)))
            for section in info:  # Go through each section
                if section.endswith("colour"):  # If this is a section_colour and not other info like head_dimensions
                    section_locations[file][section] = {}  # Create a new dictionary for this section
                    color = info[section]  # Get the color of that section
                    for x in range(0, im_arr.shape[1]):  # Go through each column of pixels in the image
                        section_locations[file][section][x] = list()  # Create a list for that column
                        first_pixel = True  # This is for the first pixel of each column of the section color.
                        for y in range(0, im_arr.shape[0]):  # Go through each pixel in the column
                            test = (im_arr[y, x] == color)
                            if test.all() and first_pixel:  # If this pixel is the right colour and is the first
                                section_locations[file][section][x].append(y)  # Add it to the list for that column
                                first_pixel = False
                            elif not test.all() and not first_pixel:
                                section_locations[file][section][x].append(y)  # Add it to the list for that column
                                first_pixel = True
                        if not section_locations[file][section][x]:
                            del section_locations[file][section][x]
    with open("Images/" + character_name + "/section_locations.pickle", "wb") as output_file:
        pickle.dump(section_locations, output_file)  # Serialise the dictionary and save it


def swap_colors(im_arr, section_colors, section_locations, original_pathname):
    section_locations = section_locations[original_pathname]  # Get the section locations for this file
    for section in section_locations:  # Go through each section
        if section_colors[section]:  # If the section colors list contains a value for this section
            for x in section_locations[section]:  # Go through each column
                for i in range(0, len(section_locations[section][x]), 2):  # Get every odd value
                    # Get every value between the first and last in a column
                    for y in range(section_locations[section][x][i], section_locations[section][x][i + 1]):
                        # Set the pixel to the new colour
                        im_arr[y, x] = list(map(lambda num: num * 255, section_colors[section]))
    return im_arr
```

```python
def hash_image(image, section_colors, name):
    # Cropping the image both makes the hashing faster and decreases the similarities between images
    try:
        image = crop_image(image, hashing=True)
        hash_output = int.from_bytes(image.tobytes(),
                                     "little")  # Convert the image array to bytes and then convert to int
        for i in range(3, 5):
            # XOR hash with the itself bit shifted to the left by i and then multiply the hash by i
            hash_output ^= hash_output << i
            hash_output *= i
        # At this point hash_output is a huge number (approximately 10^70000). We need to reduce the size of
        # hash_output to the hash table size. The best way to do this is with modulo. The hash will be used for
        # directory names so the character limit is 248 on Windows and 255 on Linux. However, we don't need the hash
        # to be that long to prevent collisions and the larger the number we modulo by the slower the hash will be. A
        # more reasonable size would be about 50 characters. The final hash won't be an integer (this would make the
        # table size 10^50 if we wanted 50 characters). Instead the hash will be converted to base 64. This means
        # that the table size can be 64^50 which is approximately 2x10^90. When using modulo for hashing we want to
        # modulo by a coprime of the initial number. This is to deal with any patterns that may occur in the number
        # which would result in collisions otherwise. Fortunately it isn't hard to find a prime close to 64^50
        # because 2^n - 1 is always prime (Mersenne primes).
        hash_output %= 64 ** 50 - 1
        byte_length = (hash_output.bit_length() + 7) // 8  # This is needed for converting an integer to bytes
        hash_output = hash_output.to_bytes(byte_length, "little")
    except ValueError:  # This means that the image is empty.
        hash_output = b""
    # The previous section of the hash only dealt with the head of the character but now we have to take into account
    # the colours of the character's skin, clothes and shoes.
    for section in section_colors.values():
        if section:  # If the user hasn't picked a new colour, section will be None
            # Round each value to 2 d.p. and concatenate as a string
            section = "".join(map(lambda x: str(round(x, 2)), section[:-1]))
        else:
            section = "0"
        hash_output += section.encode("utf-8")
```

```
    hash_output += name.encode("utf-8")  # name is the name of the character's body which the new character is based on
    # '/' is not allowed in filenames so replace with '_'
    return base64.b64encode(hash_output).decode("utf-8").replace("/", "_")


def main(old_character_name, replacement_drawing, section_colors, section_locations, preview=False):
    new_character_name = None
    old_character = make_dict(old_character_name)
    info = get_info(old_character_name)
    head_dimensions = info["head_dimensions"]
    replacement_image = drawing_to_image(replacement_drawing)
    # Resize the image to the head dimensions
    replacement_im_array = cv2.resize(replacement_image, (head_dimensions[3] - head_dimensions[2],
                                                          head_dimensions[1] - head_dimensions[0]),
                                      # This gets rid of interpolation which would make the image blurry
                                      interpolation=cv2.INTER_NEAREST)
    if preview:
        combined_image = combine_images(old_character["idle"][0], head_dimensions, replacement_im_array)
        color_swapped_image = swap_colors(combined_image, section_colors, section_locations,
                                          original_pathname=old_character["idle"][0])
        cropped_image = crop_image(color_swapped_image)
        # Resize image so it can be seen at full size on the screen
        resized_image = cv2.resize(cropped_image, (cropped_image.shape[1] * 5, cropped_image.shape[0] * 5),
                                   interpolation=cv2.INTER_NEAREST)
        save_image(resized_image, "Images/preview_image.png")
        return
    final_images = {}  # This will be a dictionary of lists containing images
    for key in old_character:
        final_images[key] = []
        for filename in old_character[key]:
            combined_image = combine_images(filename, head_dimensions, replacement_im_array)
            combined_image = swap_colors(combined_image, section_colors, section_locations, original_pathname=filename)
            final_images[key].append(combined_image)
            if key == "idle" and len(final_images[key]) == 1:
                new_character_name = hash_image(replacement_image, section_colors, old_character_name)
                if Path("Images/" + new_character_name).is_dir():  # If this character already exists
```

```
                return
    make_new_character_directories(new_character_name, final_images, replacement_drawing, old_character_name)
    return
```

paint_widget.py

This is a program for the canvas for character creation.

```
from kivy.uix.widget import Widget
from kivy.core.window import Window
from kivy.graphics import Color, Rectangle, Line, Rotate, PushMatrix, PopMatrix
from collections import OrderedDict, deque
import math
from copy import copy


def rotate_button(button):
    with button.canvas.before:
        PushMatrix()  # Without push and pop matrix, the rotate function rotates all widgets.
        Rotate(angle=180, origin=button.center)
    with button.canvas:
        PopMatrix()


# Function used to round to the nearest something that's not a power of 10.
def specific_rounding(value, rounding_number, subtractor=0):
    # The rounding is done to the nearest rounding_number so if rounding_number = 4, rounding is done to the nearest 4.
    # This is because round(x / 4) * 4 rounds x the nearest 4. This is used in the program to round the touch position
    # to the nearest rectangle to be painted on. The subtractor is needed because the rounding needs to be done relative
    # to the position from the start of the widget but the value given is relative to the position of the screen.
    return round((value - subtractor) / rounding_number) * rounding_number + subtractor


# This is the canvas that the player draws on. Inherits the widget class from kivy.
class PaintWidget(Widget):
    def __init__(self):
        super().__init__()
        self.register_event_type('on_draw')
        width = min(Window.width*0.45, Window.height*0.9)
        self.pos = ((Window.width-width)/2, (Window.height-width)/2)  # Set the widget's position based on its size.
```

```
        self.rect_width = int(width / 30)
        self.rounding_number = float(self.rect_width)  # The number things will be rounded to.
        self.size = (specific_rounding(width, self.rounding_number),
                     specific_rounding(width, self.rounding_number))
        self.tool = "pencil"  # Current tool in use is pencil.
        self.rects_colors = OrderedDict()  # An ordered dictionary containing the colours of each dot.
        self.undo_list = deque(maxlen=99)  # A list with a limit of 99 elements.
        self.redo_list = deque(maxlen=99)
        self.pencil_state = "pencil"  # Determines whether pencil button is in pencil or rubber mode.
        self.color = (1, 1, 1, 1)  # Initial colour is black.
        self.pencil_size = 1  # Number of rects drawn by a single tap of the pencil.
        with self.canvas:
            Rectangle(pos=self.pos, size=self.size)  # Create a background.
            Color(0, 0, 0)  # Set line color as black.
            # Make a grid.
            for i in range(int(self.y + self.rect_width), int(self.y + self.height), self.rect_width):
                Line(points=[self.x, i, self.x + self.width, i])
            for i in range(int(self.x + self.rect_width), int(self.x + self.width), self.rect_width):
                Line(points=[i, self.y, i, self.y + self.height])
            # Create all the dots and make them transparent.
            for pos_y in range(int(self.y), int(self.y + self.height), self.rect_width):
                for pos_x in range(int(self.x), int(self.x + self.width), self.rect_width):
                    # Ensure that the grid being made is 30x30.
                    if pos_y == int(self.y) + self.rect_width * 30 or pos_x == int(self.x) + self.rect_width * 30:
                        continue
                    self.rects_colors[(pos_x, pos_y)] = Color(0, 0, 0, 0)
                    Rectangle(pos=(pos_x, pos_y), size=(self.rect_width, self.rect_width))
        self.prev_touch = None

    def on_touch_down(self, touch):  # This method is called when the screen is touched.
        self.size = (round(Window.width * 0.4, -1), round(Window.height * 0.8, -1))  # Resets the size of the canvas.
        # Get the rounded pos
        pos = (specific_rounding(touch.x-self.rect_width/2.0, self.rounding_number, subtractor=self.x),
               specific_rounding(touch.y-self.rect_width/2.0, self.rounding_number, subtractor=self.y))
        if pos in self.rects_colors.keys():  # Ensure that the click was in the grid.
            if self.rects_colors[pos].rgba != self.color or self.tool == "rubber":
```

```
            if self.tool == "rubber" and self.rects_colors[pos].a == 0:
                return
            self.redo_list.clear()
            self.parent.parent.update_undo_nums()
            if self.tool == "pencil":
                self.pencil(pos)
            elif self.tool == "rubber":
                self.pencil(pos, erase=True)
            else:
                self.undo_list.append(self.copy_rects_colors())
                self.fill(pos, self.rects_colors[pos].rgba)

    def on_touch_move(self, touch):  # This method is called when the player holds down and moves their finger.
        # Kivy's drag detection struggles with quick movements. What happens is that there is a large gap between two
        # touches. This results in large blank spaces on the canvas between two squares drawn in. I have to simulate
        # touches between the previous touch and the current touch.
        if self.prev_touch:
            if self.prev_touch.opos == touch.opos:  # If the previous touch was part of the same drag as this touch.
                # Find the distance between the previous touch and the current touch using Pythagoras.
                dist = math.sqrt((touch.x-self.prev_touch.x)**2 + (touch.y-self.prev_touch.y)**2)
                # The simulated touches will be self.rect_width/2 apart.
                # If the distance is less than this, we don't need simulated touches.
                if dist > self.rect_width / 2:
                    # Find the angle between the previous touch and the current touch.
                    angle = math.atan2((touch.y-self.prev_touch.y), (touch.x-self.prev_touch.x))
                    count = int(dist // (self.rect_width/2))  # The number of simulated touches needed.
                    x_increment = self.rect_width/2 * math.cos(angle)
                    y_increment = self.rect_width/2 * math.sin(angle)
                    for i in range(count):
                        # self.on_touch_down takes a touch object as a parameter and uses the x and y properties of the
                        # touch object. We need to create a fake object with these properties to call the function.
                        new_touch = type('new_touch', (object,), {'x': self.prev_touch.x + x_increment * i,
                                                                   'y': self.prev_touch.y + y_increment * i})()
                        self.on_touch_down(new_touch)
        self.on_touch_down(touch)
        self.prev_touch = copy(touch)  # Have to copy it so it doesn't get overwritten by the next call.
```

```
def set_tool(self, button, other_button, tool):
    if tool == "pencil" and self.tool != "fill":  # If the player clicks on pencil while pencil/rubber is active.
        if self.tool == "pencil":
            self.tool = "rubber"
        else:
            self.tool = "pencil"
        rotate_button(button)
        self.pencil_state = self.tool
    elif tool == "pencil" and self.tool == "fill":  # If the player clicks on pencil while fill is active.
        self.tool = self.pencil_state  # Switch self.tool to whatever mode the pencil was in (either pencil/rubber).
    else:
        self.tool = tool
    button.color = (.5, .5, .5, 1)
    other_button.color = (1, 1, 1, 1)

def set_pencil(self, button, other_buttons, size):  # Used to set the dot size of the pencil.
    self.pencil_size = size
    button.color = (.5, .5, .5, 1)
    for other_button in other_buttons:
        other_button.color = (1, 1, 1, 1)

def set_color(self, color):
    self.color = color  # Sets self.color as the colour picked by the player on the ColorPicker widget.

def pencil(self, pos, erase=False, color=None):
    # Nested loops to create a square with height and width pencil_size.
    if not color:
        color = self.color
    self.undo_list.append({})
    for x in range(self.pencil_size):
        for y in range(self.pencil_size):
            # If the current pos is out of bounds, skip to the next pos.
            if (pos[0]+self.rect_width*x, pos[1]+self.rect_width*y) not in self.rects_colors.keys():
                continue
            self.undo_list[-1][(pos[0]+self.rect_width*x, pos[1]+self.rect_width*y)] = \
```

```
                self.rects_colors[(pos[0]+self.rect_width*x, pos[1]+self.rect_width*y)].rgba
            if erase:
                # Makes the rectangle at pos transparent.
                self.rects_colors[(pos[0]+self.rect_width*x, pos[1]+self.rect_width*y)].rgba = (0, 0, 0, 0)
            else:
                self.draw((pos[0]+self.rect_width*x, pos[1]+self.rect_width*y), color)  # Call the draw method.

    def draw(self, pos, color):
        self.rects_colors[pos].rgba = color  # Makes the rectangle at pos have self.color as its color.
        self.dispatch("on_draw")

    def on_draw(self):  # This is an event that is dispatched when a pixel is drawn.
        pass

    def fill(self, pos, init_color, color=None):
        if not color:
            color = self.color
        if pos not in self.rects_colors.keys():  # If the current pos is out of bounds.
            return
        if init_color == color:  # If the rectangle is already the desired colour.
            return
        # If the rectangle is a different colour to the first rectangle that was clicked on.
        if self.rects_colors[pos].rgba != init_color:
            return
        self.draw(pos, color)  # Fill in that rectangle.
        # Repeat the function with the next rectangle to the left, right, up and down.
        self.fill((pos[0]+self.rect_width, pos[1]), init_color, color=color)
        self.fill((pos[0]-self.rect_width, pos[1]), init_color, color=color)
        self.fill((pos[0], pos[1]+self.rect_width), init_color, color=color)
        self.fill((pos[0], pos[1]-self.rect_width), init_color, color=color)

    def undo(self, button, other_button, start_list, end_list):  # If button is undo, other_button is redo etc.
        if start_list:
            temp_rects_colors = start_list.pop()
            end_list.append({})
            for pos in temp_rects_colors:
```

```
                end_list[-1][pos] = self.rects_colors[pos].rgba
                self.rects_colors[pos].rgba = temp_rects_colors[pos]
            other_button.color = (1, 1, 1, 1)
        if not start_list:
            button.color = (.7, .7, 0, 1)
        self.parent.parent.update_undo_nums()

    def copy_rects_colors(self):  # Used to copy the colours of self.rects_colors and returning the result.
        temp_rects_colors = {}
        for pos in self.rects_colors:
            temp_rects_colors[pos] = self.rects_colors[pos].rgba
        return temp_rects_colors
```

creation_station.py

This is the program for the overall character creation.

```
from kivy.uix.screenmanager import Screen, NoTransition, SlideTransition
from kivy.uix.floatlayout import FloatLayout
from kivy.uix.gridlayout import GridLayout
from kivy.uix.colorpicker import ColorPicker
from kivy.uix.button import Button
from kivy.uix.label import Label
from kivy.uix.image import Image
from kivy.uix.textinput import TextInput
from kivy.uix.popup import Popup
from kivy.core.window import Window
from paint_widget import PaintWidget
from gallery import Gallery
from misc import ImageButton, Menu, get_characters, switch
from os.path import isfile
import image_editing
import pickle


# This is the screen for creating new characters. Inherits the Screen class from kivy.
class CreationStation(Screen):
    def __init__(self, name):
        super().__init__()
        self.name = name
        self.character = ""
        self.section_colors = {"skin_colour": None, "top_colour": None, "bottoms_colour": None, "shoe_colour": None}
        self.section_locations = None
        self.start_layout = FloatLayout()  # Layout that will show before a character is selected
        start_guide = Image(pos_hint={"x": .1, "y": .75}, size_hint=(0.8, 0.2), source="Images/pick_char.png")
        characters = get_characters()
        character_picker = GridLayout(pos_hint={"x": .05, "y": .05}, size_hint=(0.9, 0.75), rows=1)
        for character in characters:  # Create an ImageButton for each character
            # Only create a button if the character is pre-made.
```

```
    if not isfile("Images/" + character + "/section_locations.pickle"):
        continue
    character_picker.add_widget(ImageButton(source=characters[character],
                                            on_press=self.set_character))
self.start_layout.add_widget(start_guide)
self.start_layout.add_widget(character_picker)
self.float_layout = FloatLayout()
self.drawing_canvas = PaintWidget()
self.drawing_canvas.bind(on_draw=lambda canvas: self.on_draw(used_colors_layout, color_picker.color,
                                                             color_picker))
menu = Menu("left")
pencil = ImageButton(pos_hint={"x": .05, "y": .6}, size_hint=(0.1, 0.1),
                     source="Images/pencil.png", color=(.5, .5, .5, 1))
pencil.bind(on_press=lambda button: self.drawing_canvas.set_tool(button, fill, "pencil"))
fill = ImageButton(pos_hint={"x": .15, "y": .6}, size_hint=(0.1, 0.1), source="Images/fill.png")
fill.bind(on_press=lambda button: self.drawing_canvas.set_tool(button, pencil, "fill"))
color_picker = ColorPicker(pos_hint={"x": .75, "y": .1}, size_hint=(0.2, 0.6), color=(0, 0, 0, 1))
# Get rid of the values and sliders on the box above the color wheel
color_picker.children[0].children[1].clear_widgets()
# Add an image to the box above the color wheel
color_picker.children[0].children[1].add_widget(Image(source="Images/current_colour.png"))
# When color changes, call the set color function of the drawing canvas
color_picker.bind(color=lambda picker, color: self.drawing_canvas.set_color(color))
small_pencil = ImageButton(pos_hint={"x": .05, "y": .45}, size_hint=(0.05, 0.1),
                           source="Images/small_pencil.png", color=(.5, .5, .5, 1))
small_pencil.bind(on_press=lambda button: self.drawing_canvas.set_pencil(button, [medium_pencil,
                                                                                  large_pencil], 1))
medium_pencil = ImageButton(pos_hint={"x": .125, "y": .45}, size_hint=(0.05, 0.1),
                            source="Images/medium_pencil.png")
medium_pencil.bind(on_press=lambda button: self.drawing_canvas.set_pencil(button, [small_pencil,
                                                                                   large_pencil], 2))
large_pencil = ImageButton(pos_hint={"x": .2, "y": .45}, size_hint=(0.05, 0.1),
                           source="Images/large_pencil.png")
large_pencil.bind(on_press=lambda button: self.drawing_canvas.set_pencil(button, [small_pencil,
                                                                                  medium_pencil], 3))
used_colors_layout = GridLayout(pos_hint={"x": .05, "y": .1}, size_hint=(0.2, 0.3), cols=3, spacing=[10, 10])
```

```
save = ImageButton(pos_hint={"x": .9375, "y": .025}, size_hint=(0.05, 0.1), source="Images/save.png")
save.bind(on_press=lambda button: saving_popup.open())
save.bind(on_release=lambda button: self.on_save(self.character, self.drawing_canvas.rects_colors,
                                                 self.section_colors, save_popup, saving_popup))
preview_button = ImageButton(pos_hint={"x": .05, "y": .75}, size_hint=(0.2, 0.1), source="Images/preview.png")
preview_button.bind(on_press=lambda button: self.get_preview(paint_mode, preview_mode, preview,
                                                             self.drawing_canvas.rects_colors))
self.undo_button = ImageButton(pos_hint={"x": .1, "y": .875}, size_hint=(0.05, 0.1),
                                source="Images/undo.png", color=(.7, .5, .2, 1))
self.undo_button.bind(on_press=lambda button: self.drawing_canvas.undo(button, self.redo_button,
                                                                       self.drawing_canvas.undo_list,
                                                                       self.drawing_canvas.redo_list))
self.redo_button = ImageButton(pos_hint={"x": .16, "y": .875}, size_hint=(0.05, 0.1),
                                source="Images/redo.png", color=(.7, .5, .2, 1))
self.redo_button.bind(on_press=lambda button: self.drawing_canvas.undo(button, self.undo_button,
                                                                       self.drawing_canvas.redo_list,
                                                                       self.drawing_canvas.undo_list))
self.undo_num = Label(pos_hint={"x": .11, "y": .89}, size_hint=(0.01, 0.02),
                      text=str(len(self.drawing_canvas.undo_list)), color=(0, 0, 0, 1),
                      font_name="pixel_font.otf", font_size=Window.height * 0.02)
self.redo_num = Label(pos_hint={"right": .2, "y": .89}, size_hint=(0.01, 0.02),
                      text=str(len(self.drawing_canvas.redo_list)), color=(0, 0, 0, 1),
                      font_name="pixel_font.otf", font_size=Window.height * 0.02)
restart_confirmation = ImageButton(source="Images/restart.png")
restart_confirmation.bind(on_press=self.restart)
restart_popup = Popup(pos_hint={"x": .3, "y": .35}, size_hint=(.4, .3),
                      content=restart_confirmation, title="")
restart_button = ImageButton(pos_hint={"x": .9375, "y": .875}, size_hint=(0.05, 0.1),
                             source="Images/cancel.png")
restart_button.bind(on_press=lambda button: restart_popup.open())
gallery_button = ImageButton(pos_hint={"x": .75, "y": .75}, size_hint=(.2, .1), source="Images/gallery.png")
gallery_button.bind(on_press=lambda button: switch("Gallery", self.manager, "up", screen_type=Gallery))
preview = Image(pos_hint={"center_x": .5, "y": .05}, size_hint=(0.4, 0.8))
draw = ImageButton(pos_hint={"x": .05, "y": .75}, size_hint=(0.2, 0.1), source="Images/draw.png")
draw.bind(on_press=lambda button: self.toggle_mode(preview_mode, paint_mode))
skin_color = ImageButton(pos_hint={"x": .05, "y": .65}, size_hint=(0.2, 0.1), source="Images/skin_colour.png")
```

```
        skin_color.bind(on_press=lambda button: self.change_color("skin_colour", color_picker.color,
                                                                  preview, self.drawing_canvas.rects_colors,
                                                                  used_colors_layout, color_picker.color, color_picker))
        top_color = ImageButton(pos_hint={"x": .05, "y": .57}, size_hint=(0.2, 0.1), source="Images/top_colour.png")
        top_color.bind(on_press=lambda button: self.change_color("top_colour", color_picker.color,
                                                                 preview, self.drawing_canvas.rects_colors,
                                                                 used_colors_layout, color_picker.color, color_picker))
        bottoms_color = ImageButton(pos_hint={"x": .05, "y": .49}, size_hint=(0.2, 0.1),
                                    source="Images/bottoms_colour.png")
        bottoms_color.bind(on_press=lambda button: self.change_color("bottoms_colour", color_picker.color,
                                                                     preview, self.drawing_canvas.rects_colors,
                                                                     used_colors_layout, color_picker.color,
                                                                     color_picker))
        shoe_color = ImageButton(pos_hint={"x": .05, "y": .41}, size_hint=(0.2, 0.1), source="Images/shoe_colour.png")
        shoe_color.bind(on_press=lambda button: self.change_color("shoe_colour", color_picker.color,
                                                                  preview, self.drawing_canvas.rects_colors,
                                                                  used_colors_layout, color_picker.color, color_picker))
        save_popup = Popup(pos_hint={"x": .3, "y": .35}, size_hint=(.4, .3),
                           content=Image(source="Images/saved.png"), title="")
        saving_popup = Popup(pos_hint={"x": .3, "y": .35}, size_hint=(.4, .3),
                             content=Image(source="Images/saving.png"), title="")
        paint_mode = [self.drawing_canvas, menu, pencil, fill, color_picker, small_pencil, preview_button, save,
                      self.undo_button, self.redo_button, gallery_button, medium_pencil, large_pencil,
                      used_colors_layout, restart_button, self.undo_num, self.redo_num]
        preview_mode = [color_picker, used_colors_layout, save, preview, restart_button,
                        draw, skin_color, top_color, bottoms_color, shoe_color, menu, gallery_button]
        self.add_widget(self.start_layout)
        self.start_layout.add_widget(Menu("left"))
        for widget in paint_mode:
            self.float_layout.add_widget(widget)

    def set_character(self, character):
        self.character = character.source.split("/")[1]
        self.remove_widget(self.start_layout)
        self.add_widget(self.float_layout)
        with open("Images/" + self.character + "/section_locations.pickle", "rb") as char_file:
```

```
        # Load the serialised dictionary into self.section_locations
        self.section_locations = pickle.load(char_file)

    def toggle_mode(self, current_mode, next_mode):
        for widget in current_mode:
            self.float_layout.remove_widget(widget)
        for widget in next_mode:
            self.float_layout.add_widget(widget)

    def get_preview(self, current_mode, next_mode, preview, drawing):
        image_editing.main(self.character, drawing, self.section_colors, self.section_locations, preview=True)
        preview.source = "Images/preview_image.png"
        preview.reload()  # This is need to update the image because the source didn't change.
        self.toggle_mode(current_mode, next_mode)

    def change_color(self, section, new_color, preview, drawing, colors_layout, color, color_picker):
        if self.section_colors[section] == new_color:  # If the requested color is already the section color, return.
            return
        # Have to add the [:] so it is a copy, otherwise the section color changes whenever new_color changes
        # This leads to all the sections having the same color at all times
        self.color_used(colors_layout, color, color_picker)
        self.section_colors[section] = new_color[:]
        image_editing.main(self.character, drawing, self.section_colors, self.section_locations, preview=True)
        preview.reload()  # This is need to update the image because the source didn't change.

    def on_save(self, old_character_name,  replacement_drawing, section_colors, save_popup, saving_popup):
        image_editing.main(old_character_name, replacement_drawing, section_colors, self.section_locations)
        saving_popup.dismiss()
        save_popup.open()
        if not self.manager.has_screen("Gallery"):
            self.manager.add_widget(Gallery(name="Gallery"))
        self.manager.get_screen("Gallery").changes_made = True

    def restart(self, button=None):
        if button:
            # Kivy popups contain a grid layout then a box layout and then the content so 3 parent levels are needed
```

```
            button.parent.parent.parent.dismiss()
            # This is here because otherwise it is possible to double press the button which causes an error
            button.parent.remove_widget(button)
        manager = self.manager
        manager.transition = NoTransition()
        manager.remove_widget(self)  # Remove this screen
        # Create a fresh new screen (python garbage collection makes sure the old screen doesn't use up any memory so
        # you can restart as much as wanted with no memory errors.
        manager.add_widget(CreationStation(self.name))
        manager.current = self.name
        manager.transition = SlideTransition()

    def load_character(self, character_name):
        manager = self.manager
        self.restart()
        with open("Images/" + character_name + "/rects_colors.pickle", "rb") as char_file:
            # Load the serialised dictionary into self.section_locations
            temp_rects_colors = pickle.load(char_file)
        for pos in temp_rects_colors:
            manager.current_screen.drawing_canvas.rects_colors[pos].rgba = temp_rects_colors[pos]

    def on_draw(self, used_colors_layout, color, color_picker):
        self.color_used(used_colors_layout, color, color_picker)
        self.update_undo_nums()

    # Method for updating the numbers next to the undo and redo buttons and change the colours of the buttons.
    def update_undo_nums(self):
        for undo_widgets in [[self.undo_num, self.undo_button, self.drawing_canvas.undo_list],
                             [self.redo_num, self.redo_button, self.drawing_canvas.redo_list]]:
            undo_widgets[0].text = str(len(undo_widgets[2]))
            if undo_widgets[0].text == "0":
                undo_widgets[0].color = (0, 0, 0, 1)
                undo_widgets[1].color = (.7, .7, 0, 1)
            else:
                undo_widgets[0].color = (0, 0, .3, 1)
                undo_widgets[1].color = (1, 1, 1, 1)
```

```
    def color_used(self, grid_layout, color, color_picker):
        color_picker.color = color  # Set the colour of the color_picker to the chosen colour
        for button in grid_layout.children:
            if button.background_color == color:  # If this colour has been used before leave the function
                return
        new_btn = Button(background_normal="", background_color=color)  # Create a new button with the chosen colour
        # When pressed, the button calls a function which sets the colour as the button's colour
        new_btn.bind(on_press=lambda x: self.color_btn_pressed(new_btn.background_color, color_picker))
        grid_layout.add_widget(new_btn)

    @staticmethod
    def color_btn_pressed(color, color_picker):
        color_picker.color = color  # Set the colour of the color_picker to the chosen colour
```

gallery.py

This is the program for the gallery screen.

```
from kivy.uix.screenmanager import Screen
from kivy.uix.floatlayout import FloatLayout
from kivy.uix.gridlayout import GridLayout
from kivy.uix.scrollview import ScrollView
from kivy.uix.popup import Popup
from kivy.uix.image import Image
from kivy.core.window import Window
from os.path import isfile
from shutil import rmtree
from misc import switch, get_characters, ImageButton


class Gallery(Screen):
    def __init__(self, name):
        super().__init__()
        self.name = name
        self.changes_made = True  # Used so the gallery only updates if changes have been made (save has occurred)
        character_scroll = ScrollView(pos_hint={"x": .05, "y": .05}, size_hint=(.9, 0.65), bar_color=(1, 1, 1, 1),
                                      bar_inactive_color=(1, 1, 1, .6), bar_width=Window.height/100)
        self.character_picker = GridLayout(size_hint=(None, 1), rows=1)
        self.popup_layout = GridLayout(rows=2, spacing=[0, Window.height/50], padding=[0, Window.height/50])
        Popup(pos_hint={"x": .35, "y": .35}, size_hint=(.3, .3),  # This will pop up when a character is selected
              content=self.popup_layout, title="")
        character_scroll.add_widget(self.character_picker)
        draw = ImageButton(pos_hint={"x": .75, "y": .75}, size_hint=(.2, .1), source="Images/draw.png")
        draw.bind(on_press=lambda button: switch("Creation Station", self.manager, "down"))
        float_layout = FloatLayout()
        self.add_widget(float_layout)
        float_layout.add_widget(draw)
        float_layout.add_widget(character_scroll)

    def on_pre_enter(self):  # Gets called when the screenmanager starts switching to this screen.
```

```
        if self.changes_made:
            characters = get_characters()
            self.character_picker.clear_widgets()
            for character in characters:  # Create an ImageButton for each character
                # Only create a button if the character has a saved head.
                if not isfile("Images/" + character + "/rects_colors.pickle"):
                    continue
                char_button = ImageButton(source=characters[character])
                char_button.bind(on_press=self.character_pressed)
                char_button.reload()
                self.character_picker.add_widget(char_button)
            if len(self.character_picker.children) == 0:
                self.character_picker.add_widget(Image(source="Images/no_saved_chars.png"))
                self.character_picker.width = Window.width * .9
            else:
                self.character_picker.width = len(self.character_picker.children) * Window.height * 0.5
            self.changes_made = False

    def character_pressed(self, button):
        character_name = button.source.split("/")[1]
        self.popup_layout.clear_widgets()
        self.popup_layout.add_widget(ImageButton(source="Images/load.png",
                                                 on_press=lambda btn: self.popup_layout.parent.parent.parent.dismiss(),
                                                 on_release=lambda btn: self.manager.get_screen(
                                                     "Creation Station").load_character(character_name)))
        self.popup_layout.add_widget(ImageButton(source="Images/delete.png",
                                                 on_press=lambda btn: self.delete_char(character_name)))
        self.popup_layout.parent.parent.parent.open()  # This opens the popup

    def delete_char(self, character_name):
        self.popup_layout.parent.parent.parent.dismiss()  # This dismisses the popup
        while True:
            try:
                rmtree("Images/" + character_name)
                break
            except OSError:
```

```
            pass
        self.changes_made = True
        self.on_pre_enter()  # Reload the gallery
```

Gameplay

play.py

This is the program for the play screen.

```
from kivy.uix.floatlayout import FloatLayout
from kivy.uix.screenmanager import Screen
from character_selection import CharacterSelection
from misc import Menu, ImageButton, switch
from multiplayer import Multiplayer


class Play(Screen):
    def __init__(self, name):
        super().__init__()
        self.name = name
        float_layout = FloatLayout()
        menu = Menu("up")
        single_player = ImageButton(pos_hint={"x": .2, "y": .35}, size_hint=(.15, .3),
                                    source="Images/creation_station.png")
        single_player.bind(on_press=lambda button: switch("Character Selection", self.manager, "down",
                                                          screen_type=CharacterSelection))
        multiplayer = ImageButton(pos_hint={"x": .5, "y": .35}, size_hint=(.3, .3), source="Images/multiplayer.png")
        multiplayer.bind(on_press=lambda button: switch("Multiplayer", self.manager, "left", screen_type=Multiplayer))
        self.add_widget(float_layout)
        float_layout.add_widget(menu)
        float_layout.add_widget(single_player)
        float_layout.add_widget(multiplayer)
```

character_selection.py

This is the program for the character selection screen.

```
from kivy.uix.screenmanager import Screen
from kivy.uix.floatlayout import FloatLayout
from kivy.uix.gridlayout import GridLayout
from kivy.uix.scrollview import ScrollView
from kivy.uix.label import Label
from kivy.uix.button import Button
from kivy.uix.popup import Popup
from kivy.core.window import Window
from kivy.graphics import Color, Line
from kivy.clock import Clock
from misc import ImageButton, switch, get_characters
from game import Game
from random import choice
from os.path import isfile


# A class for the character selection screen. Inherits the Screen class from kivy. Has optional parameters for online.
class CharacterSelection(Screen):
    def __init__(self, name, character_choices=3, multiplayer=False, protocol=None, transport=None, username=None,
                 team=None):
        super().__init__()
        self.name = name
        self.init_character_choices = character_choices
        self.character_choices = character_choices
        self.multiplayer = multiplayer
        self.protocol = protocol
        self.team = team
        if protocol:
            self.protocol.character_selection = self
        self.transport = transport
        self.username = username
        self.characters = []
```

```
        float_layout = FloatLayout()
        leave_button = Button(background_normal='', background_color=(1, 1, 1, 1), text="Leave?",
                              font_name="pixel_font.otf", color=(0, 0, 0, 1), font_size=Window.height * 0.1)
        leave_button.bind(on_press=lambda button: self.leave(leave_popup))
        leave_popup = Popup(pos_hint={"x": .3, "y": .35}, size_hint=(.4, .3),
                            content=leave_button, title="")
        back = ImageButton(pos_hint={"x": .01, "top": .99}, size_hint=(0.1, 0.1), source="Images/back.png")
        back.bind(on_press=lambda button: leave_popup.open())
        cancel = ImageButton(pos_hint={"x": .94, "top": .99}, size_hint=(0.05, 0.1), source="Images/cancel.png")
        cancel.bind(on_press=self.cancel)
        pick_label = Label(pos_hint={"x": .15, "y": .7}, size_hint=(0.7, 0.3),
                           text="Pick " + str(self.character_choices) + " character" +
                                ("s" if self.character_choices > 1 else ""),
                           font_name="pixel_font.otf", color=(0, 0, 0, 1), font_size=Window.height * 0.1)
        character_scroll = ScrollView(pos_hint={"x": .05, "y": .125}, size_hint=(.9, .7), bar_color=(1, 1, 1, 1),
                                      bar_inactive_color=(1, 1, 1, .6), bar_width=Window.height / 100)
        self.character_picker = GridLayout(padding=[0, Window.height*.1, 0, Window.height*.05], size_hint=(None, 1),
                                           rows=1)
        character_scroll.add_widget(self.character_picker)
        self.start_button = Button(size_hint=(.2, .075), pos_hint={"x": .75, "y": .025}, background_normal='',
                                   background_color=(.3, .4, .3, .7), text="Start", font_name="pixel_font.otf",
                                   color=(0, 0, 0, 1), font_size=Window.height * 0.05, background_down="")
        self.start_button.bind(on_press=lambda button: self.start())
        self.add_widget(float_layout)
        float_layout.add_widget(back)
        float_layout.add_widget(cancel)
        float_layout.add_widget(character_scroll)
        float_layout.add_widget(pick_label)
        float_layout.add_widget(self.start_button)

    # Method for reducing the timer by one and automatically starting if the timer goes to zero.
    def countdown(self, timer):
        timer.text = str(int(timer.text) - 1)
        if int(timer.text) < 1:
            self.countdown_schedule.cancel()
            for i in range(self.character_choices):
```

```
            self.characters.append(choice(list(get_characters().keys())))  # Choose a random character.
        self.start_button.background_color = (.5, 1, .5, .7)
        self.start()

    def on_pre_enter(self):  # Gets called when the screenmanager starts switching to this screen.
        if self.multiplayer:
            self.timer = Label(size_hint=(.1, .1), pos_hint={"x": .025, "y": .05}, text="10",
                               font_name="pixel_font.otf", color=(0, 0, 0, 1), font_size=Window.height * 0.1)
            self.add_widget(self.timer)
            self.countdown_schedule = Clock.schedule_interval(lambda instance: self.countdown(self.timer), 1)
        characters = get_characters()
        self.character_picker.clear_widgets()
        num_characters = 0
        for character in characters:  # Create an ImageButton for each character
            if not isfile(characters[character]):
                continue
            char_button = ImageButton(source=characters[character])
            char_button.bind(on_press=self.character_pressed)
            char_button.reload()
            self.character_picker.add_widget(char_button)
            num_characters += 1
        self.character_picker.width = num_characters * Window.height * 0.5

    # Called when the user taps on a character.
    def character_pressed(self, button):
        if self.character_choices > 0:  # Don't do anything if the user has selected all their characters.
            self.character_choices -= 1
            color = [0, 0, 0]
            color[2-self.character_choices] = 1  # Create a colour (either red, green or blue).
            rect_color = Color(*color)
            self.character_picker.canvas.add(rect_color)
            # Make a rounded rectangle outline with size depending on which choice it is.
            self.character_picker.canvas.add(Line(rounded_rectangle=[button.x + Window.width * .05 - Window.width *
                                                                     (2-self.character_choices) * .005,
                                                                     button.y + Window.height * .01 - Window.height *
                                                                     (2-self.character_choices) * .01,
```

```
                                                    button.width - Window.width * .1 + Window.width *
                                                    (2-self.character_choices) * .01,
                                                    button.height + Window.height *
                                                    (2-self.character_choices) * .02,
                                                    Window.width*.05],
                                            width=Window.width*.005))
            self.characters.append(button.source.split("/")[1])
            if self.character_choices == 0:  # If all the choices have been made.
                self.start_button.background_color = (.5, 1, .5, .7)
                self.start_button.background_down = "atlas://data/images/defaulttheme/button_pressed"

    # Called when the cancel button is pressed.
    def cancel(self, button=False):
        if button:
            for instruction in self.character_picker.canvas.children:
                if hasattr(instruction, "points"):  # Remove all the rounded rectangles.
                    self.character_picker.canvas.remove(instruction)
            self.characters = []
            self.character_choices = self.init_character_choices
            self.start_button.background_color = (.3, .4, .3, .7)
        else:
            self.manager.remove_widget(self)
            del self

    def leave(self, popup):
        if self.multiplayer:
            self.transport.write("leave\n".encode("utf-8"))
        switch("Play", self.manager, "up")
        popup.dismiss()
        self.cancel()

    def start(self):
        if self.start_button.background_color == [.5, 1, .5, .7]:
            if self.multiplayer:
                self.transport.write(("characters;" + ";".join(self.characters) + ";" + self.team + "\n").
                                     encode("utf-8"))  # Send a message containing the characters chosen and the team.
```

```
            self.add_widget(Popup(pos_hint={"x": .1, "y": .35}, size_hint=(.8, .3), auto_dismiss=False,
                                  content=Label(text="Waiting for players", font_name="pixel_font.otf",
                                                font_size=Window.height * 0.05), title=""))
        else:
            self.manager.add_widget(Game("Game", self.characters))
            switch("Game", self.manager, "down")
            self.cancel()

    def enter_game(self, message):
        message = message.replace("characters_chosen;", "")  # Message is a 3 dimensional array in string form.
        background = message.split(";;;")[-1]
        message = message.split(";;;")[0]
        message = message.split(";;")
        output = {}
        for user in message:
            if not user == "":
                user = user.split(";")
                output[user[0]] = user[1:]
        self.manager.add_widget(Game("Game", output, multiplayer=True, username=self.username, protocol=self.protocol,
                                     transport=self.transport, background=background))
        switch("Game", self.manager, "down")
        self.cancel()
```

game.py

This is a program for the game itself.

```
from kivy.uix.screenmanager import Screen
from kivy.uix.floatlayout import FloatLayout
from kivy.uix.gridlayout import GridLayout
from kivy.clock import Clock
from kivy.uix.popup import Popup
from kivy.uix.button import Button
from kivy.core.window import Window
from kivy.graphics import Rectangle, Color, Ellipse
from kivy.garden.joystick import Joystick
from image_editing import make_dict, get_info
from misc import ImageButton, switch, get_characters
from character import Character
from ai import AI
from ball import Ball
from os.path import isdir
from random import choice
import math


# This is the screen where the actual game is played. Inherits the Screen class from kivy.
class Game(Screen):
    def __init__(self, name, characters, multiplayer=False, username="", protocol=None, transport=None,
                 background=None):
        super().__init__()
        self.name = name
        self.characters = characters
        self.multiplayer = multiplayer
        self.username = username
        self.protocol = protocol
        if protocol:
            self.protocol.game = self
        self.transport = transport
```

```
self.messages = []
self.selected_character = None
if not background:
    # Choose a random background if one was not already given (in multiplayer).
    background = choice(["Images/[0.7]background_mud.png", "Images/[0.6]background_grass.png",
                         "Images/[0.7]background_night.png", "Images/[0.3]background_snow.png"])
self.config = get_info("config.ini", character=False)
self.fps = 60
with self.canvas:
    Rectangle(source=background, size=Window.size)
pause_button = ImageButton(pos_hint={"x": .475, "y": .9}, size_hint=(.05, .1),
                           source="Images/pause.png", color=(1, 1, 1, .2))
pause_button.bind(on_press=lambda button: pause_popup.open())
pause_layout = GridLayout(cols=1)
pause_layout.add_widget(Button(size_hint=(.3, .1), pos_hint={"x": .1, "y": .7}, background_normal='',
                               background_color=(.5, .5, .5, .5), text="Quit", font_name="pixel_font.otf",
                               color=(0, 0, 0, 1), font_size=Window.height * 0.05,
                               on_press=lambda button: self.quit(popup=pause_popup)))
pause_popup = Popup(title="Pause", title_font="pixel_font.otf", title_size=Window.height * 0.07,
                    title_align="center", size_hint=(.9, .9), content=pause_layout)
joystick = CustomJoystick(pos_hint={"x": float(self.config["joystick_pos"][0]),
                                    "y": float(self.config["joystick_pos"][1])},
                          size_hint=self.config["joystick_size"],
                          pad_color=self.config["joystick_color"])
# Calls the selected character's run function every tick of the game. But only when the schedule is active.
# This will be only when the joystick is touched.
self.run_schedule = Clock.schedule_interval(lambda instance: self.selected_character.run(joystick.radians),
                                            1/self.fps)
self.run_schedule.cancel()  # Make the schedule inactive.
joystick.bind(on_touch_down=lambda stick, pad: self.check_touch_location(stick, pad, self.run_schedule))
joystick.bind(on_touch_move=lambda stick, pad: self.check_touch_location(stick, pad, self.run_schedule))
joystick.bind(on_touch_up=lambda stick, pad: self.run_schedule.cancel())
jump_button = ImageButton(pos_hint={"x": float(self.config["button_1_pos"][0]),
                                    "y": float(self.config["button_1_pos"][1])},
                          size_hint=self.config["button_1_size"],
                          source="Images/circle.png", color=self.config["button_1_color"])
```

```
jump_button.bind(on_press=lambda button: self.selected_character.jump())
catch_button = ImageButton(pos_hint={"x": float(self.config["button_2_pos"][0]),
                                     "y": float(self.config["button_2_pos"][1])},
                           size_hint=self.config["button_2_size"],
                           source="Images/circle.png", color=self.config["button_2_color"])
catch_button.bind(on_press=lambda button: self.selected_character.catch())
switch_button = ImageButton(pos_hint={"x": float(self.config["button_3_pos"][0]),
                                      "y": float(self.config["button_3_pos"][1])},
                            size_hint=self.config["button_3_size"],
                            source="Images/circle.png", color=self.config["button_3_color"])
switch_button.bind(on_press=lambda button: self.switch_characters())
light_throw_button = ImageButton(pos_hint={"x": float(self.config["button_1_pos"][0]),
                                           "y": float(self.config["button_1_pos"][1])},
                                 size_hint=self.config["button_1_size"],
                                 source="Images/circle.png", color=self.config["button_1_color"])
light_throw_button.bind(on_press=lambda button: self.selected_character.light_throw())
heavy_throw_button = ImageButton(pos_hint={"x": float(self.config["button_2_pos"][0]),
                                           "y": float(self.config["button_2_pos"][1])},
                                 size_hint=self.config["button_2_size"],
                                 source="Images/circle.png", color=self.config["button_2_color"])
heavy_throw_button.bind(on_press=lambda button: self.selected_character.heavy_throw())
pass_button = ImageButton(pos_hint={"x": float(self.config["button_3_pos"][0]),
                                    "y": float(self.config["button_3_pos"][1])},
                          size_hint=self.config["button_3_size"],
                          source="Images/circle.png", color=self.config["button_3_color"])
pass_button.bind(on_press=lambda button: self.selected_character.pass_to_teammate())
ball = Ball(background)
self.ball = ball
self.sprites = [ball]
self.controllable_sprites = []
if multiplayer:
    self.calculating_ball = False
    if list(self.characters.keys()).index(self.username) == 0:
        self.calculating_ball = True
    self.online_characters = []
    for character in self.characters[self.username][:-1]:
```

```
            controllable_char = AI(make_dict(character), ball, int(self.characters[self.username][-1][-1]))
            self.sprites.append(controllable_char)
            self.controllable_sprites.append(controllable_char)
        self.selected_character = self.controllable_sprites[0]
        self.selected_character.selected = True

        # Create a new class for online characters which has reduced functionality so the online characters
        # aren't using any processing power.
        class OnlineCharacter(Character):
            prev_health = 100

            # The only method usually called in update that needs to be called is update_healthbar so override
            def update(self):
                super().update_healthbar()

            def animate_next_frame(self):  # Have to override this function to prevent animation occurring locally.
                pass

        for user in self.characters:
            if user != self.username:
                for i, character in enumerate(self.characters[user][:-1]):
                    if not isdir("Images/" + character):  # If the user doesn't have the character.
                        character = choice(list(get_characters().keys()))
                    online_char = OnlineCharacter(make_dict(character), ball, int(self.characters[user][-1][-1]))
                    online_char.user = user
                    online_char.num = str(i)
                    self.sprites.append(online_char)
                    self.online_characters.append(online_char)

    else:
        for char in self.characters:
            self.controllable_sprites.append(AI(make_dict(char), ball, 1))
        self.selected_character = self.controllable_sprites[0]
        self.controllable_sprites[0].selected = True
        self.sprites.extend(self.controllable_sprites)
        for i in range(3):
```

```
            self.sprites.append(AI(make_dict(choice(list(get_characters().keys()))), ball, 2))
        self.sprite_layout = FloatLayout()
        self.float_layout = FloatLayout()
        self.add_widget(self.sprite_layout)
        self.add_widget(self.float_layout)
        self.float_layout.add_widget(pause_button)
        self.float_layout.add_widget(joystick)
        self.float_layout.add_widget(jump_button)
        self.has_ball_buttons = [light_throw_button, heavy_throw_button, pass_button]
        self.enemy_has_ball_buttons = [jump_button, catch_button, switch_button]
        self.buttons = [light_throw_button, heavy_throw_button, pass_button, jump_button, catch_button, switch_button]
        for sprite in self.sprites:
            self.sprite_layout.add_widget(sprite)
            sprite.game = self
        # Call these functions every tick.
        self.main_schedule = Clock.schedule_interval(lambda time: (self.update_sprites(), self.update_buttons(),
                                                                   self.arrange_sprites(), self.create_shadows(),
                                                                   self.send_data(), self.online_update(),
                                                                   self.variable_frame_rate(time=time)), 1/self.fps)
        # Need a separate speed for character animation because the speed of the animations has to stay constant.
        Clock.schedule_interval(lambda time: self.animate_sprites(), .05)

    # Called when the screen is entered.
    def on_pre_enter(self):
        # This is just to save resources. Removes the character creation screen when in a game.
        if self.manager.has_screen("Creation Station"):
            self.manager.remove_widget(self.manager.get_screen("Creation Station"))

    @staticmethod
    def check_touch_location(widget, touch, func):  # Checks if the user touched the widget.
        if widget.collide_point(*touch.opos):
            func()

    # Function for changing the frame rate of the game.
    def variable_frame_rate(self, time=None):
        # If the device was able to complete the previous tick in the correct amount of time, increase the fps.
```

```
        if math.isclose(time, 1 / self.fps, rel_tol=0.1):
            self.fps += 1
        elif self.fps > 1:  # Otherwise decrease the fps as long as it is above 1.
            self.fps -= 1
        self.main_schedule.timeout = 1 / self.fps
        self.run_schedule.timeout = 1 / self.fps

    def quit(self, popup=None):
        if self.manager:
            self.main_schedule.cancel()
            switch("Main Menu", self.manager, "up")
            self.manager.remove_widget(self)
            if popup:
                popup.dismiss()
            if self.multiplayer:
                self.transport.write("leave\n".encode("utf-8"))
            del self

    def animate_sprites(self):
        for sprite in self.sprites:
            try:
                sprite.animate_next_frame()
            except AttributeError:
                continue

    def update_sprites(self):
        if self.ball.character:
            self.ball.character.has_ball = True
        for sprite in self.sprites:
            sprite.update()
            try:
                if sprite.selected:
                    self.selected_character = sprite
                    if sprite.dead:
                        self.switch_characters()
            except AttributeError:
```

```
            pass

    def send_data(self):
        if self.multiplayer:
            output = "update;"
            for i, character in enumerate(self.controllable_sprites):
                output += ";".join([self.username, str(i), str(character.x/Window.width),
                                    str(character.y/Window.height), str(character.z/Window.height),
                                    character.prev_state, str(character.state_frames[character.prev_state]),
                                    str(character.health), str(character.color)])
                output += ";;"
                if character.has_ball:
                    self.calculating_ball = True
            output = output.strip(";;")
            if self.calculating_ball:
                output += "\nupdate;"
                ball_char_user = "None"
                ball_char_num = "None"
                if self.ball.character:
                    try:
                        ball_char_num = str(self.controllable_sprites.index(self.ball.character))
                        ball_char_user = self.username
                    except ValueError:  # This means that a different user now has the ball
                        self.calculating_ball = False
                        self.transport.write((output + "\n").encode("utf-8"))
                        return
                for character in self.online_characters:
                    if character.prev_health != character.health:
                        self.ball.hit_player()
                output += ";".join(["ball", str(self.ball.x/Window.width), str(self.ball.y/Window.height),
                                    str(self.ball.z/Window.height), str(self.ball.color), str(self.ball.midair),
                                    str(self.ball.speed), str(self.ball.throwing_team), ball_char_user, ball_char_num,
                                    str(self.ball.passed)])
            self.transport.write((output + "\n").encode("utf-8"))

    def update_buttons(self):
```

```
        for widget in self.buttons:
            try:
                self.float_layout.remove_widget(widget)
            except ValueError:
                pass
        if self.selected_character.has_ball:
            for widget in self.has_ball_buttons:
                self.float_layout.add_widget(widget)
        else:
            for widget in self.enemy_has_ball_buttons:
                self.float_layout.add_widget(widget)

    def switch_characters(self):
        self.selected_character.selected = False
        self.selected_character = self.controllable_sprites[self.controllable_sprites.index(self.selected_character)-1]
        self.selected_character.selected = True

    # Method for moving sprites in front of and behind each other.
    def arrange_sprites(self):
        # Sort the sprites in order of their z and remove then add them back in reverse order.
        for sprite in sorted(self.sprites, key=lambda widget: widget.z, reverse=True):
            self.sprite_layout.remove_widget(sprite)
            self.sprite_layout.add_widget(sprite)

    def create_shadows(self):
        for sprite in self.sprites:
            if not hasattr(sprite, "shadow"):
                sprite.canvas.before.add(Color(0, 0, 0, .3))
                sprite.shadow = Ellipse()
                sprite.canvas.before.add(sprite.shadow)
            # The shadow's size is dependent on the sprites size and height above ground.
            sprite.shadow.size = (sprite.width/2+(sprite.y-sprite.z)/2, sprite.height/3+(sprite.y-sprite.z)/2)
            sprite.shadow.pos = (sprite.center_x-sprite.width/5-(sprite.y-sprite.z)/4,
                                 sprite.z-sprite.height/6-(sprite.y-sprite.z)/4)
            try:
                if sprite.character:  # If the sprite is the ball.
```

```
                sprite.canvas.before.clear()
                del sprite.shadow
        except AttributeError:
            if sprite.health <= 0:
                sprite.canvas.before.clear()
                del sprite.shadow

# Method for keeping the most recent message for each user.
def receive_message(self, message):
    if message.split(";")[0] == "ball":
        try:
            if message.split(";")[8] != self.username:  # If the ball is not calculated by the user.
                self.calculating_ball = False
        except IndexError:
            pass
    for i, user in enumerate(self.messages):
        if message.split(";")[0] == user.split(";")[0]:
            self.messages[i] = message
            return
    # This will only be reached if there are no messages kept from this user.
    self.messages.append(message)

# Method for applying the attributes received in the messages to the online characters and ball.
def online_update(self):
    for message in self.messages:
        message = message.split(";;")
        for i in range(len(message)):
            try:
                message[i] = message[i].split(";")
                if message[i][0] != "ball":
                    for character in self.online_characters:
                        if character.user == message[i][0] and character.num == message[i][1]:
                            character.x = float(message[i][2]) * Window.width
                            character.y = float(message[i][3]) * Window.height
                            character.z = float(message[i][4]) * Window.height
                            try:
```

```
                            from image_editing import make_dict
                            if character.texture != character.image_dict[message[i][5]][int(
                                    float(message[i][6]))].texture:
                                character.texture = character.image_dict[message[i][5]][int(
                                    float(message[i][6]))].texture
                        except IndexError:
                            pass
                        character.prev_health = character.health
                        try:
                            character.health = float(message[i][7])
                            character.color = message[i][8].replace("[", "").replace("]", "").split(", ")
                        except ValueError:
                            character.prev_health = 0
                            character.health = 0
                elif not self.calculating_ball:
                    self.ball.x = float(message[i][1]) * Window.width
                    self.ball.y = float(message[i][2]) * Window.height
                    self.ball.z = float(message[i][3]) * Window.height
                    self.ball.color = list(map(int, message[i][4].replace("[", "").replace("]", "").split(", ")))
                    self.ball.midair = message[i][5] == "True"
                    self.ball.speed = float(message[i][6])
                    self.ball.throwing_team = None if message[i][7] == "None" else int(message[i][7])
                    self.ball.passed = message[i][10] == "True"
                    if message[i][8] == "None":
                        self.ball.character = None
                    else:
                        for character in self.online_characters:
                            if character.user == message[i][8] and character.num == message[i][9]:
                                self.ball.character = character
            except IndexError:
                pass


# Class for a joystick with certain properties.
class CustomJoystick(Joystick):
    def __init__(self, pos_hint=None, size_hint=None, pad_color=None):
```

```
        super().__init__(pos_hint=pos_hint, size_hint=size_hint)
        self.outer_background_color = [0, 0, 0, .3]
        self.outer_line_color = [0, 0, 0, 0]
        self.inner_background_color = [0, 0, 0, 0]
        self.inner_line_color = [0, 0, 0, 0]
        self.pad_background_color = pad_color
        self.pad_line_color = [0, 0, 0, 0]
        self.pad_size = .4
        self.inner_size = .3
        self.outer_size = .6
```

character.py

This is the program for the character class.

```
from kivy.uix.image import Image
from kivy.core.window import Window
from kivy.core.image import Image as CoreImage
from kivy.clock import Clock
from kivy.graphics import Rectangle, Color, Triangle
import math
from copy import deepcopy
from misc import set_color
from random import randint


class Character(Image):
    def __init__(self, image_dict, ball, team):
        super().__init__()
        self.image_dict = image_dict
        self.allow_stretch = True
        self.ball = ball
        self.unit_x = Window.width / 200  # Create a distance unit that is scaled to window size
        self.unit_y = Window.height / 100
        self.size_hint = (None, None)
        self.size = (self.unit_x*30, self.unit_y*30)
        # Boundaries of the pitch
        self.init_bounds = [[Window.width / 12, Window.width / 2.6], [Window.height / 7.8, Window.height / 1.8]]
        if team == 2:
            # Change the horizontal boundaries to the other side and reverse them.
            self.init_bounds[0] = list(map(lambda x: Window.width - self.width - x, self.init_bounds[0]))[::-1]
        # deepcopy is used because otherwise a change to self.bounds will change self.init_bounds.
        self.bounds = deepcopy(self.init_bounds)
        # Start at a random position.
        self.y = randint(int(self.bounds[1][0]), int(self.bounds[1][1]))
        self.x = randint(int(self.bounds[0][0]), int(self.bounds[0][1]))
        self.z = self.y
```

```
        self.team = team
        self.cache_images()
        self.angle = 0
        self.texture = self.image_dict["idle"][0].texture
        self.state = "idle"
        self.prev_state = "idle"
        self.state_is_continuous = True  # If true, the state is repeating e.g. idle, otherwise it is a one-time action.
        self.has_ball = False
        self.had_ball = False  # This is used to determine which player sends the ball information in multiplayer.
        self.running_jump = False
        self.teammates = []
        self.enemies = []
        self.state_frames = {}  # This is a dictionary of all the current frames of each state.
        self.animation_speeds = {}
        for state in self.image_dict:
            self.state_frames[state] = 0
            self.animation_speeds[state] = 1
        self.animation_speeds["idle"] = 0.5
        self.animation_speeds["idle_ball"] = 0.5
        self.animation_speeds["pick_up"] = 0.5
        self.throw = None
        self.health = 100
        self.dead = False
        self.selected = False
        with self.canvas:
            Color(0, 0, 0, 1)
            self.base_bar = Rectangle()  # Create a black bar beneath the healthbar to look like an outline.
            Color(0, 1, 0, 1)
            self.healthbar = Rectangle()
            self.selected_cursor_colour = Color(1, 0, 0, 0)
            self.selected_cursor = Triangle(points=[self.center_x, self.top + self.unit_y * 2,
                                                    self.center_x - 2 * self.unit_x, self.top + 5 * self.unit_y,
                                                    self.center_x + 2 * self.unit_x, self.top + 5 * self.unit_y])

    # Store the image textures in a dictionary.
    def cache_images(self):
```

```
        # Kivy automatically caches images but flipping the images wouldn't work.
        for state in self.image_dict:
            for i in range(len(self.image_dict[state])):
                # nocache is set as tru to prevent kivy's automatic caching.
                self.image_dict[state][i] = CoreImage(self.image_dict[state][i], nocache=True)
                if self.team == 2:  # Make team 2 face the other way.
                    self.image_dict[state][i].texture.flip_horizontal()

    # Called every tick of the game.
    def update(self):
        self.update_enemies_and_teammates()
        if len(self.enemies) == 0 or len(self.teammates) == 0:  # Game over.
            self.parent.parent.quit()
        self.update_healthbar()
        if self.dead:
            return
        self.change_bounds()
        if self.ball.throwing_team and self.collision():  # If the ball has been thrown and has hit the character.
            if self.ball.passed:
                # If the ball was passed by a teammate.
                if self.team == self.ball.throwing_team and not self.had_ball:
                    self.ball.throwing_team = None
                    self.state = "pick_up"
                    self.state_frames["pick_up"] = 2
            elif self.team != self.ball.throwing_team:  # If the ball was thrown by an enemy.
                self.hit()
        if self.ball.character != self:
            if self.ball.character:  # If someone else now has the ball.
                self.had_ball = False
            self.has_ball = False
        if self.state == "jump":
            self.jump_movement()
        else:
            self.running_jump = False
            self.z = self.y
            # If the ball is on the floor, is touching the character, is slow, no-one else has the ball and the
```

```
            # character wasn't just hit by the ball.
            if self.floor_collision() and not self.ball.throwing_team and self.ball.speed < 5 / self.game.fps and \
                    not self.ball.character and not self.color == [1, .3, .3, 1]:
                teammate_picking_up = False
                for teammate in self.teammates:
                    if teammate.state == "pick_up":
                        teammate_picking_up = True
                if not teammate_picking_up:  # If a teammate is already picking the ball up, don't pick it up.
                    self.pick_up()
        if self.state == "pick_up" and self.state_frames[self.state] == 2:  # If the character has finished picking up.
            self.has_ball = True
            self.ball.color = (0, 0, 0, 0)
            self.ball.character = self
        if self.has_ball:
            self.had_ball = True
            if self in self.parent.parent.controllable_sprites:
                for teammate in self.parent.parent.controllable_sprites:
                    teammate.selected = False  # Deselect the characters that don't have the ball.
                self.selected = True  # Select the character with the ball.
            self.ball.midair = False
            self.ball.x_speed = 0
            self.ball.y_speed = 0
            self.ball.z_speed = 0
            self.ball.x = self.center_x
            self.ball.y = self.center_y
            self.ball.z = self.y
        if self.selected:
            self.reposition_selected_cursor()
            self.selected_cursor_colour.a = 1  # Make the cursor visible.
        else:
            self.selected_cursor_colour.a = 0  # Make the cursor invisible.
        if self.state_frames[self.state] == len(self.image_dict[self.state]) - 1 and \
                (self.state == "light_throw" or self.state == "heavy_throw"):  # If character is at the end of the throw
            self.do_throw()

    # Method for changing the image to the next one.
```

```
    def animate_next_frame(self):
        self.prev_state = self.state
        try:
            self.texture = self.image_dict[self.state][int(self.state_frames[self.state])].texture
            # Increment the frame by the animation speed.
            self.state_frames[self.state] += self.animation_speeds[self.state]
        except IndexError:  # The animation is complete.
            self.state_frames[self.state] = 0
            if not self.state_is_continuous:
                self.idle()
                self.state_is_continuous = True
        if self.state_is_continuous:
            self.idle()  # Automatically go back to idle.

    # Changes the bounds so that the character can retrieve the ball if it is outside of the pitch on their side.
    def change_bounds(self):
        if not self.ball.character:
            if self.team == 1:
                if self.ball.x - self.width / 2 < self.init_bounds[0][0]:  # If the ball is off on the left.
                    self.bounds[0][0] = -self.width / 2
                else:
                    self.bounds[0][0] = self.init_bounds[0][0]
                if self.ball.x - self.width < self.init_bounds[0][1]:  # If the ball is on the character's side.
                    if self.ball.z < self.init_bounds[1][0]:  # If the ball is off on the bottom.
                        self.bounds[1][0] = 0
                    else:
                        self.bounds[1][0] = self.init_bounds[1][0]
                    if self.ball.z > self.init_bounds[1][1]:  # If the ball is off on the top.
                        self.bounds[1][1] = Window.height * 0.65
                    else:
                        self.bounds[1][1] = self.init_bounds[1][1]
            else:
                if self.ball.x - self.width / 2 > self.init_bounds[0][1]:  # If the ball is off on the right.
                    self.bounds[0][1] = Window.width - self.width / 2
                else:
                    self.bounds[0][1] = self.init_bounds[0][1]
```

```
            if self.ball.x > self.init_bounds[0][0]:  # If the ball is on the character's side.
                if self.ball.z < self.init_bounds[1][0]:  # If the ball is off on the bottom.
                    self.bounds[1][0] = 0
                else:
                    self.bounds[1][0] = self.init_bounds[1][0]
                if self.ball.z > self.init_bounds[1][1]:  # If the ball is off on the top.
                    self.bounds[1][1] = Window.height * 0.65
                else:
                    self.bounds[1][1] = self.init_bounds[1][1]
        else:
            self.bounds = deepcopy(self.init_bounds)  # Reset the boundaries.

    # Called when the joystick is moved or when the AI is controlling the character.
    def run(self, angle):
        if not self.running_jump:  # If the character is in a running jump they can't change direction.
            self.angle = angle
            if not self.state_is_continuous:  # Don't interrupt a one-time action e.g. catch.
                return
            self.state_is_continuous = True
            if self.has_ball:
                self.state = "run_ball"
            else:
                self.state = "run"
        # Ensure the character stays within the bounds.
        # If direction is right or x is greater than the left-most boundary.
        if math.cos(self.angle) > 0 or self.x > self.bounds[0][0]:
            # If direction is left or x is less than the right-most boundary.
            if math.cos(self.angle) < 0 or self.x < self.bounds[0][1]:
                # Use trig to get the amount x should be increased by.
                self.x += self.unit_x * math.cos(self.angle) * 30 / self.game.fps
        # If direction is up or x is greater than the bottom-most boundary.
        if math.sin(self.angle) > 0 or self.z > self.bounds[1][0]:
            # If direction is down or x is less than the top-most boundary.
            if math.sin(self.angle) < 0 or self.z < self.bounds[1][1]:
                # y and z are increase by the same amount because the player is staying on the ground.
                self.y += self.unit_y * math.sin(self.angle) * 30 / self.game.fps
```

```
            self.z += self.unit_y * math.sin(self.angle) * 30 / self.game.fps

    def jump(self):
        if self.state == "run":  # If the player is currently running, do a running jump.
            self.running_jump = True
        self.state_is_continuous = False
        self.state = "jump"

    def jump_movement(self):
        if self.running_jump:
            self.run(self.angle)
        if self.state_frames["jump"] > 1:  # Jump movement starts on the 3rd frame.
            # Increase height by a constantly decreasing amount.
            self.y += self.unit_y * (len(self.image_dict["jump"]) / 2 + 1 - self.state_frames["jump"]) * \
                      60 / self.game.fps

    def idle(self):
        if self.has_ball:
            self.state = "idle_ball"
        else:
            self.state = "idle"

    def pick_up(self):
        self.ball.midair = False
        self.state_is_continuous = False
        self.state = "pick_up"

    def catch(self):
        if not self.state_is_continuous:  # Don't interrupt anything.
            return
        self.state_is_continuous = False
        self.state = "catch"

    def light_throw(self):
        if not self.ball.midair and self.has_ball:
            self.state_is_continuous = False
```

```
            self.state = "light_throw"
            self.state_frames[self.state] = 0  # Resetting the frame allows feinting.
            self.has_ball = False

    def heavy_throw(self):
        if not self.ball.midair and self.has_ball:
            self.state_is_continuous = False
            self.state = "heavy_throw"
            self.state_frames[self.state] = 0
            self.has_ball = False

    def do_throw(self, throw_to_enemies=True, explosion=False):
        try:
            target_x, target_z = self.find_target(throw_to_enemies=throw_to_enemies)
        except TypeError:  # If no target is found.
            return
        if explosion:
            target_x *= -1  # Make the ball go into the player holding it.
        self.ball.color = (1, 1, 1, 1)
        self.ball.character = None
        throw_type = self.state
        self.ball.throwing_team = self.team
        self.ball.midair = True
        if not throw_to_enemies:
            self.ball.passed = True
        else:
            self.ball.passed = False
        self.ball.y = self.center_y
        self.ball.throw(target_x, target_z, throw_type)

    def pass_to_teammate(self):
        if len(self.teammates) > 1:
            self.has_ball = False
            self.ball.character = None
            self.do_throw(throw_to_enemies=False)
```

```
    # Update the lists of enemies and teammates.
    def update_enemies_and_teammates(self):
        for sprite in self.parent.parent.sprites:
            try:
                if sprite not in self.enemies and sprite not in self.teammates and sprite.health > 0:
                    if sprite.team == self.team:
                        self.teammates.append(sprite)
                    else:
                        self.enemies.append(sprite)
            except AttributeError:
                pass
        for group in self.enemies, self.teammates:
            for sprite in group:
                if sprite.health < 0:
                    group.remove(sprite)

    # Function finds the character (either enemy or teammate depending on the type of throw) whose angle from the
    # thrower is closest to the current/most recent angle made on the joystick.
    def find_target(self, throw_to_enemies=True):
        angle = math.pi * 2  # Start with just above the maximum angle.
        chosen_target = None
        if throw_to_enemies:
            target_list = self.enemies
        else:
            target_list = self.teammates
        for target in target_list:
            if target == self:
                continue
            dif_x = target.center_x - self.center_x
            dif_z = target.z - self.z
            target_angle = math.atan(dif_z / dif_x)
            # Have to sin and then asin self.angle to get the lowest version of the angle e.g. pi rad becomes 0 rad.
            angle_dif = abs(target_angle-math.asin(math.sin(self.angle)))
            if angle_dif < angle:
                angle = angle_dif
                chosen_target = target
```

```
        if chosen_target:
            return [chosen_target.center_x, chosen_target.z]

    # Return true if the ball is touching or is very close to the centre of the player.
    def collision(self):
        if math.isclose(self.ball.z, self.z, abs_tol=self.unit_x*5) and \
                math.isclose(self.ball.center_x, self.center_x, abs_tol=self.unit_x*10) and \
                self.y - self.z <= self.ball.y - self.ball.z < self.height:
            return True

    # Used for picking up rather than getting hit. Due to this, the ball must also be close to the ground.
    def floor_collision(self):
        if math.isclose(self.ball.y, self.ball.z, abs_tol=self.unit_x) and self.collision():
            return True

    # Method for changing the value and location of the healthbar.
    def update_healthbar(self):
        if self.health <= 0:
            self.base_bar.size = (0, 0)
            self.healthbar.size = (0, 0)
            self.selected_cursor_colour.a = 0
            self.color = [*self.color[:-1], float(self.color[-1]) - 1.5 / self.game.fps]  # Slowly fade away.
            if self.color[-1] <= 0 and not self.dead:
                for enemy in self.enemies:
                    try:
                        del enemy.enemies[enemy.enemies.index(self)]  # Remove self from enemy's enemy list.
                    except ValueError:
                        pass
                for teammate in self.teammates:
                    try:
                        del teammate.teammates[teammate.teammates.index(self)]
                    except ValueError:
                        pass
                self.dead = True
            if self.has_ball:
                self.ball.midair = True
```

```
            self.has_ball = False
            self.ball.color = (1, 1, 1, 1)
            self.ball.character = None
    else:
        health_width = self.width/200*self.health
        self.healthbar.size = (health_width, self.unit_y)
        self.healthbar.pos = (self.center_x-self.width/4, self.top+self.unit_y)
        self.base_bar.size = (self.width/2+self.unit_y/2, self.unit_y*3/2)
        self.base_bar.pos = (self.center_x-self.width/4-self.unit_y/4, self.top+self.unit_y*3/4)

# Called when the ball hits the player.
def hit(self):
    if self.state == "catch" and self.state_frames[self.state] < 6:  # If the player caught in time.
        self.ball.midair = False
        self.ball.throwing_team = None
        self.ball.passed = False
        self.has_ball = True
        self.ball.color = (0, 0, 0, 0)
        self.ball.character = self
        self.idle()
        return
    if self.color == [1, .3, .3, 1]:  # Prevent double hits
        return
    if self.ball.throw_type == "light_throw":
        self.health -= 20
    else:
        self.health -= 25
    self.ball.hit_player()
    self.color = (1, .3, .3, 1)  # Go red.
    Clock.schedule_once(lambda clock: set_color(self, (1, 1, 1, 1)), 0.25)  # Return to normal colour in 0.25 secs.
    if self.state != "jump":
        self.state = "hit"  # Don't interrupt the jump animation.
        self.state_is_continuous = False

# Moves the selected cursor to above the player's head.
def reposition_selected_cursor(self):
```

```
        self.selected_cursor.points = [self.center_x, self.top + self.unit_y * 2,
                                       self.center_x - 2 * self.unit_x, self.top + 5 * self.unit_y,
                                       self.center_x + 2 * self.unit_x, self.top + 5 * self.unit_y]
```

AI.py

This is the program for the AI class.

```
from character import Character
import math
import random
from numpy.random import normal


# Inherits the Character class. Adds computer-controlled functionality to the Character.
class AI(Character):
    def __init__(self, image_dict, ball, team):
        super().__init__(image_dict, ball, team)  # Initialise the parent class (Character) to get the same attributes.
        self.current_angle = math.radians(random.randint(0, 360))  # Start off with a random angle
        self.walk_tick = 0
        self.acted = False
        self.thrown = False
        self.prev_angle = 0

    # Called every tick of the game.
    def update(self):
        if not self.selected:  # If the character is selected, ignore all the additional AI methods.
            self.run()
            self.defend()
            self.random_throw()
        super().update()

    # Overrides the run method of Character. Will generate an angle based on the circumstances and then call the
    # overridden run method with the generated angle.
    def run(self, angle=None):
        if not self.selected:  # Ignore the overridden section if the character is selected.
            # First find the x and z distances of the character to the ball.
            dif_x = self.center_x - self.ball.x
            dif_z = self.z - self.ball.z
            angle = self.get_angle(dif_z, dif_x)  # Get the angle from the character to the ball.
```

```
        if not self.ball.throwing_team or self.ball.throwing_team == self.team:  # If the ball hasn't been thrown.
            self.thrown = False
        # If the ball has been thrown by the opponent
        if self.ball.throwing_team and self.ball.throwing_team != self.team:
            if self.thrown:  # If the angle has already been generated, don't create a new angle.
                angle = self.prev_angle
            else:
                # Find the direction the ball is travelling.
                ball_angle = self.get_angle(self.ball.z_speed, self.ball.x_speed)
                # We want the AI to try to move away from the ball. The best way to do this is to move perpendicular
                # to the ball's direction of travel. However, this is not realistic and the AIs will all move in the
                # same/opposite directions. Instead, the AI should move perpendicular to the thrower. However,
                # this still has the possibility that the ball is thrown to a different player and the AI runs into
                # the ball. To prevent this, the AI will run perpendicular to the thrower in the way that moves away
                # from the ball.
                if angle < ball_angle:
                    angle -= math.pi / 2
                else:
                    angle += math.pi / 2
                self.prev_angle = angle
                self.thrown = True
        elif self.ball.character or not self.bounds[0][0] + self.width / 10 < self.ball.x < \
                self.bounds[0][1] + self.width:  # If someone has the ball or the ball is in the other side.
            angle = self.random_angle()
        else:  # This means that the ball is on the AI's side and no-one has picked it up
            distance = math.sqrt(dif_x ** 2 + dif_z ** 2)  # Get the distance to the ball
            counter = 0
            for sprite in self.teammates:
                if math.sqrt((sprite.center_x - sprite.ball.x) ** 2 + (sprite.z - sprite.ball.z) ** 2) < \
                        distance:  # If the teammate is closer to the ball than the AI
                    if counter == 1:  # If two teammates are closer then move randomly
                        angle = self.random_angle()
                    else:
                        counter += 1
            if dif_x > 0:
                angle += math.pi  # The angle was away from the ball before now it is towards the ball.
```

```
        super().run(angle)  # Call the original run function with the generated angle.

    # Generate a random angle but not every tick since this would just make the AI vibrate in place
    def random_angle(self):
        # walk_tick is incremented every tick so it has to be divided by the fps to get a constant time.
        if self.walk_tick * 30 / self.game.fps < random.randint(10, 15):
            self.walk_tick += 1
        else:  # If the AI has run in the same direction for long enough.
            # We want the AIs to run randomly but weighted towards the back of their half.
            mean = 0
            # For team 1, pi radians as the angle is towards the back of their half whereas 0 is for team 2
            if self.team == 1:
                mean = math.pi
            self.current_angle = normal(mean, math.pi/2)  # Use the normal distribution with standard deviation as pi/2
            self.walk_tick = 0  # Reset the walk timer.
        return self.current_angle

    # Jump, catch or do nothing. Only do the action once per throw
    def defend(self):
        if self.ball.throwing_team and self.ball.throwing_team != self.team:
            if not self.acted:
                dif_x = self.center_x - self.ball.x
                dif_z = self.z - self.ball.z
                angle = self.get_angle(dif_z, dif_x)
                ball_angle = self.get_angle(self.ball.z_speed, self.ball.x_speed)
                if math.sqrt(dif_x ** 2 + dif_z ** 2) < self.width * 2:  # If the ball is close to the AI
                    if abs(angle - ball_angle) < 0.1:  # If the ball is going towards the AI
                        chance = random.randint(0, 2)  # 1/3 chance for each action
                        if chance == 0:
                            super().jump()
                        elif chance == 1:
                            super().catch()
                        self.acted = True
        else:
            self.acted = False  # Reset once the ball is no longer in midair.
```

```
    @staticmethod
    def get_angle(a, b):
        try:
            angle = math.atan(a / b)
        except ZeroDivisionError:  # If b is zero set angle as a right angle (pi/2 radians)
            angle = math.pi / 2
        return angle

    def random_throw(self):
        if self.has_ball and random.randint(0, self.game.fps) < 3:  # Only do sometimes when the character has the ball.
            chance = random.randint(0, 2)
            if chance < 1:
                super().light_throw()
            elif chance < 2:
                super().heavy_throw()
            else:
                super().pass_to_teammate()
```

ball.py

This is the program for the ball class.

```
from kivy.uix.image import Image
from kivy.core.window import Window
from kivy.clock import Clock
from kivy.graphics import Rectangle
import math
from random import randint
import time
from os.path import isfile


# Class for the ball. Inherits the Image class from kivy.
class Ball(Image):
    def __init__(self, surface):
        super().__init__()
        self.size_hint = (None, None)
        self.pos = (Window.width / 2, Window.height / 2)  # Start in the middle
        self.z = Window.height / 3
        self.x_speed = randint(int(self.width * -0.5), int(self.width * .5))  # Have a random x_speed
        self.y_speed = 0
        self.z_speed = 0
        self.speed = 0
        # Get the friction constant from the surface
        self.surface_friction = float(surface[surface.index("[") + 1:surface.index("]")])
        self.completion_ticks = 10
        self.throwing_team = None  # This is the team that the ball was thrown by
        self.character = None  # The character holding the ball
        self.midair = True
        self.has_bounced = False
        self.rolling = False
        self.passed = False
        self.throw_type = None
        self.source = "Images/ball.png"
```

```
        self.size = (Window.width / 35, Window.height / 20)
        self.timer = time.time()

    # Called every tick of the game
    def update(self):
        self.x += self.x_speed
        self.z += self.z_speed
        self.y += self.y_speed
        self.speed = math.sqrt(
            (self.x_speed / self.width) ** 2 + (self.z_speed / self.height) ** 2) * 30 / self.game.fps  # Pythagoras
        self.bounce()
        if self.character:
            self.rolling = False
            self.passed = False
        if self.rolling:
            # x_speed and z_speed are multiplied by surface_friction every tick. With a high fps, this would make
            # them decrease more rapidly since there are more ticks. Therefore, surface friction must be
            # exponentiated by a constant over the fps.
            self.x_speed *= self.surface_friction ** (30 / self.game.fps)
            self.z_speed *= self.surface_friction ** (30 / self.game.fps)
        if self.midair:
            # Since y_speed is subtracted from rather than multiplied by, the fps only has to multiply the constant.
            self.y_speed -= (self.height * 30 / self.game.fps) / self.completion_ticks  # Gravity
        else:
            self.y_speed = self.z_speed  # Make the ball's vertical speed 0
            if not self.character:
                self.y = self.z  # Make the ball on the ground.
        self.hot_potato()

    # Called when a character throws the ball
    def throw(self, target_x, target_z, throw_type):
        distance = math.sqrt((target_x - self.x) ** 2 + (target_z - self.z) ** 2)  # Pythagoras theorem.
        total_speed = self.width * 60 / self.game.fps  # Set the speed as a constant.
        self.throw_type = throw_type
        if throw_type == "heavy_throw":  # Increase the speed if it's a heavy throw, decrease it if it's a pass.
            total_speed *= 1.5
```

```
        if self.passed:
            total_speed *= 0.5
        self.completion_ticks = distance / total_speed  # Work out how many ticks it will take to reach the target.
        # Work out the separate x and z components of the speed based on the x and z distances and the number of ticks.
        self.x_speed = (target_x - self.x) / self.completion_ticks
        self.z_speed = (target_z - self.z) / self.completion_ticks
        # self.height / 4 makes the trajectory look more realistic.
        self.y_speed = self.z_speed + (self.height / 4) * 30 / self.game.fps
        if throw_type == "light_throw":  # Make the trajectory more looping for light_throw.
            self.y_speed += (self.height / 4) * 30 / self.game.fps

    def bounce(self):
        if self.y < self.z < Window.height * .65:  # If the ball is touching the ground.
            # If the ball bounced last frame and is still going downwards i.e. the ball is very slow.
            if self.has_bounced and self.y_speed < self.z_speed:
                self.passed = False
                self.midair = False
                self.y_speed = self.z_speed
                self.y = self.z
                self.rolling = True
                return
            self.has_bounced = True
            self.throwing_team = None  # Once the ball has bounced, it doesn't do damage.
            self.x_speed *= self.surface_friction  # Doesn't have to be exponentiated by fps since it only happens once.
            self.z_speed *= self.surface_friction
            # Multiply the vertical speed by friction and then take it away from the z_speed.
            self.y_speed = self.z_speed - self.surface_friction * (self.y_speed - self.z_speed)
            self.y += self.y_speed
        else:
            self.has_bounced = False
        if self.z < 0 or self.z > Window.height * .64:  # If the ball has hit the upper or lower boundaries.
            self.z_speed *= -.8
            self.y_speed = self.z_speed + .8 * (self.y_speed + self.z_speed)
            self.z += self.z_speed
        if self.x < 0 or self.x > Window.width - self.width:  # If the ball has hit the left or right boundaries.
            self.x_speed *= -.8
```

```
        self.x += self.x_speed

    # Called when the ball hits a player.
    def hit_player(self):
        self.x_speed *= -.7
        self.y_speed *= .7
        self.z_speed *= .7
        self.throwing_team = None
        self.update()

    # If a team is time-wasting with the ball, the ball will explode, damaging them.
    def hot_potato(self):
        if self.midair and not self.passed:  # Reset timer whenever the ball is thrown.
            self.timer = time.time()
        if time.time() - self.timer > 7:  # After 7 seconds, explode.
            self.explode()
            self.timer = time.time()  # Reset timer.

    def explode(self):
        if self.character:
            self.character.do_throw(explosion=True)
        else:
            self.midair = True
            self.rolling = False
            self.throw(Window.width - self.center_x, randint(0, int(Window.height/3)), "light_throw")
            self.y_speed += self.height * 10 / self.game.fps  # Give a slight boost to the ball's vertical speed.
        if self.center_x < Window.width / 2:  # Set the throwing team as the team on the other side.
            self.throwing_team = 2
        else:
            self.throwing_team = 1
        # Animate the explosion
        self.explosion_animation()

    # Recursive function for animating the explosion.
    def explosion_animation(self, counter=1, explosion=None):
        if not explosion:
```

```
        with self.canvas:
            explosion = Rectangle(source="Explosion/1.png", width=self.width*2, height=self.height*2, pos=self.pos)
    if isfile("Explosion/" + str(counter) + ".png"):
        explosion.source = "Explosion/" + str(counter) + ".png"
        # Animate the next frame in 0.025 seconds.
        Clock.schedule_once(lambda clock: self.explosion_animation(counter=counter+1, explosion=explosion), .025)
    else:  # Once all the images have been animated, remove the explosion.
        self.canvas.remove(explosion)
```

controls_changer.py

This is the program for the change controls screen.

```
from kivy.uix.screenmanager import Screen
from kivy.uix.floatlayout import FloatLayout
from kivy.uix.popup import Popup
from kivy.uix.colorpicker import ColorPicker
from kivy.uix.scatterlayout import ScatterLayout
from kivy.uix.gridlayout import GridLayout
from kivy.uix.slider import Slider
from kivy.uix.image import Image
from kivy.graphics import Rectangle, Color, Line
from kivy.core.window import Window
from misc import ImageButton, switch
from image_editing import get_info


# This is the screen for changing the controls of the game. The user can change the size, opacity and colours of the
# buttons and joysticks. Inherits the Screen class from kivy.
class ControlsChanger(Screen):
    def __init__(self, name):
        super().__init__()
        self.name = name
        self.config = get_info("config.ini", character=False)
        with self.canvas:
            self.background = Rectangle(source="Images/[0.7]background_mud.png", size=Window.size)
        restart_confirmation = ImageButton(source="Images/cancel_text.png")
        restart_confirmation.bind(on_press=self.cancel)
        set_to_default = ImageButton(source="Images/set_to_default.png")
        set_to_default.bind(on_press=self.set_to_default)
        restart_grid_layout = GridLayout(cols=1)
        restart_grid_layout.add_widget(restart_confirmation)
        restart_grid_layout.add_widget(set_to_default)
        restart_popup = Popup(pos_hint={"x": .3, "y": .35}, size_hint=(.4, .3),
                              content=restart_grid_layout, title="")
```

```
restart_button = ImageButton(pos_hint={"x": .9375, "y": .875}, size_hint=(0.05, 0.1),
                             source="Images/cancel.png")
restart_button.bind(on_press=lambda button: restart_popup.open())
save = ImageButton(pos_hint={"x": .0125, "y": .875}, size_hint=(0.05, 0.1), source="Images/save.png")
save.bind(on_press=lambda button: self.save())
color_picker = ColorPicker(pos_hint={"x": .5, "y": .6}, size_hint=(.2, .4))
color_picker.children[0].remove_widget(color_picker.children[0].children[1])
color_picker.bind(color=lambda picker, color: self.set_color(color))
self.scale_slider = Slider(min=.5, max=2, value=1, cursor_size=(Window.width*.025, Window.width*.025),
                           background_width=Window.width*.05)
self.scale_slider.bind(on_touch_move=lambda slider, touch: self.slider_touched(slider, "scale"))
self.scale_slider.bind(on_touch_down=lambda slider, touch: self.slider_touched(slider, "scale"))
self.alpha_slider = Slider(min=0, max=1, value=.7, cursor_size=(Window.width*.025, Window.width*.025),
                           background_width=Window.width*.05)
self.alpha_slider.bind(on_touch_down=lambda slider, touch: self.slider_touched(slider, "alpha"))
self.alpha_slider.bind(on_touch_move=lambda slider, touch: self.slider_touched(slider, "alpha"))
slider_layout = GridLayout(rows=2, cols=1, pos_hint={"x": .2, "y": .775}, size_hint=(.2, .2))
slider_layout.add_widget(self.scale_slider)
slider_layout.add_widget(self.alpha_slider)
# The kivy ScatterLayout widgets are used for the buttons and joystick because it allows enlargement.
button_1_scatter = ScatterLayout(do_rotation=False, do_scale=False, do_translation=False, id="button_1",
                                 pos=(float(self.config["button_1_pos"][0])*Window.width,
                                      float(self.config["button_1_pos"][1])*Window.height),
                                 size_hint=self.config["button_1_size"])
button_1_scatter.add_widget(Image(source="Images/circle.png", color=self.config["button_1_color"],
                                  size_hint=(1, 1)))
button_2_scatter = ScatterLayout(do_rotation=False, do_scale=False, do_translation=False, id="button_2",
                                 pos=(float(self.config["button_2_pos"][0]) * Window.width,
                                      float(self.config["button_2_pos"][1]) * Window.height),
                                 size_hint=self.config["button_2_size"])
button_2_scatter.add_widget(Image(source="Images/circle.png", color=self.config["button_2_color"],
                                  size_hint=(1, 1)))
button_3_scatter = ScatterLayout(do_rotation=False, do_scale=False, do_translation=False, id="button_3",
                                 pos=(float(self.config["button_3_pos"][0]) * Window.width,
                                      float(self.config["button_3_pos"][1]) * Window.height),
                                 size_hint=self.config["button_3_size"])
```

```
        button_3_scatter.add_widget(Image(source="Images/circle.png", color=self.config["button_3_color"],
                                          size_hint=(1, 1)))
        joystick_scatter = ScatterLayout(do_rotation=False, do_scale=False, do_translation=False, id="joystick",
                                         pos=(float(self.config["joystick_pos"][0]) * Window.width,
                                              float(self.config["joystick_pos"][1]) * Window.height),
                                         size_hint=self.config["joystick_size"], auto_bring_to_front=False)
        joystick_scatter.add_widget(Image(source="Images/circle.png", color=[0, 0, 0, .3],
                                          size_hint=(.6, .6), pos_hint={"center_x": .5, "center_y": .5}))
        joystick_scatter.add_widget(Image(source="Images/circle.png", color=self.config["joystick_color"],
                                          size_hint=(.4, .4), pos_hint={"center_x": .5, "center_y": .5}))
        float_layout = FloatLayout()
        self.add_widget(float_layout)
        float_layout.add_widget(restart_button)
        float_layout.add_widget(save)
        float_layout.add_widget(color_picker)
        float_layout.add_widget(slider_layout)
        self.scatters = [joystick_scatter, button_1_scatter, button_2_scatter, button_3_scatter]
        for scatter in self.scatters:
            float_layout.add_widget(scatter)
            scatter.outline_color = Color(1, 1, 1, 0)
            scatter.outline = Line(close=True)
            scatter.canvas.add(scatter.outline_color)
            scatter.canvas.add(scatter.outline)
        self.selected_scatter = None

    def cancel(self, button):
        if button:
            button.parent.parent.parent.parent.dismiss()
            button.parent.clear_widgets()
        switch("Main Menu", self.manager, "right")
        self.manager.remove_widget(self)

    # Called when the user touches the screen.
    def on_touch_down(self, touch):
        scatter_touched = False
        for scatter in self.scatters:
```

```
        if scatter.collide_point(*touch.pos):  # If the user touched the scatter.
            self.selected_scatter = scatter
            scatter.outline.points = (0, 0, 0, scatter.height, scatter.width, scatter.height, scatter.width, 0)
            scatter.outline_color.a = 1
            scatter_touched = True
            self.scale_slider.value = self.selected_scatter.scale
            self.alpha_slider.value = self.selected_scatter.children[0].children[0].color[-1]
    for scatter in self.scatters:
        if scatter != self.selected_scatter and scatter_touched:  # Make the outline invisible if not selected.
            scatter.outline_color.a = 0
    if not scatter_touched:  # This is required so that touch still works for buttons and the colour wheel.
        super().on_touch_down(touch)

# Called when the user drags their finger after touching the screen.
def on_touch_move(self, touch):
    # Make sure the scatter doesn't go above a certain point unless it is the joystick.
    if self.selected_scatter and (touch.y + self.selected_scatter.height *
                                  self.selected_scatter.scale/2 < Window.height*.6 or
                                  len(self.selected_scatter.children[0].children) == 2) \
            and (self.selected_scatter.collide_point(*touch.opos) or touch.id == 1):
        touch.id = 1
        self.selected_scatter.pos = (touch.x-self.selected_scatter.width*self.selected_scatter.scale/2,
                                     touch.y-self.selected_scatter.height*self.selected_scatter.scale/2)

def slider_touched(self, slider, slider_type):
    if self.selected_scatter:
        if slider_type == "scale":
            self.selected_scatter.scale = slider.value
        elif slider_type == "alpha":
            self.selected_scatter.children[0].children[0].color[-1] = slider.value

def set_color(self, color):
    if self.selected_scatter:
        # Change only the rgb values.
        self.selected_scatter.children[0].children[0].color[0:3] = color[0:3]
```

```
    def save(self):
        for scatter in self.scatters:
            self.config[scatter.id + "_pos"] = (scatter.x / Window.width, scatter.y / Window.height)
            self.config[scatter.id + "_size"] = list(map(lambda x: x*scatter.scale, scatter.size_hint))
            self.config[scatter.id + "_color"] = scatter.children[0].children[0].color
        with open("config.ini", "w") as config:
            config_list = []
            for key in self.config.keys():
                config_list.append(key + "\n" + ";".join(map(str, self.config[key])))
            config.write("\n".join(config_list))
        Popup(pos_hint={"x": .3, "y": .35}, size_hint=(.4, .3),
              content=Image(source="Images/saved.png"), title="").open()
        switch("Main Menu", self.manager, "right")

    def set_to_default(self, button):
        self.cancel(button)
        with open("config.ini", "w") as config:
            with open("default_config.ini", "r") as default_config:
                config.write(default_config.read())
```

Networking

multiplayer.py

This is the program for the multiplayer screen.

```
from kivy.uix.screenmanager import Screen
from kivy.uix.floatlayout import FloatLayout
from kivy.uix.gridlayout import GridLayout
from kivy.uix.button import Button
from kivy.core.window import Window
from kivy.uix.label import Label
from kivy.uix.textinput import TextInput
from kivy.uix.tabbedpanel import TabbedPanel, TabbedPanelItem
from kivy.clock import Clock
from kivy.uix.scrollview import ScrollView
from kivy.uix.popup import Popup
from misc import switch, ImageButton
from lobby import Lobby
from join import Join
from image_editing import get_info
from misc import get_characters
from kivy.support import install_twisted_reactor
import sys
from os.path import isfile

if 'twisted.internet.reactor' in sys.modules:
    del sys.modules['twisted.internet.reactor']
install_twisted_reactor()  # This is required to allow the kivy and twisted event loops to work together.
from twisted.internet import reactor
from client import ClientFactory
```

```
# This is the screen for multiplayer. It inherits the Screen class from kivy.
class Multiplayer(Screen):
    def __init__(self, name):
        super().__init__()
        self.name = name
        info = get_info("online_info.ini", character=False)
        self.username = "".join(map(str, info["username"]))
        self.friends = info["friends"]
        self.client_factory = ClientFactory(self, self.username)
        self.transport = None
        reactor.connectTCP(info["server_ip"][0], 50000, self.client_factory)  # Start the client.
        connecting = Label(text="[color=ff0000]A[/color]ttempting to connect...", font_name="pixel_font.otf",
                           size_hint=(0.7, 0.3), pos_hint={"x": .15, "y": .35}, color=(0, 0, 0, 1),
                           font_size=Window.height * 0.1, markup=True)
        self.tracker = 0
        self.add_widget(connecting)
        Clock.schedule_interval(lambda instance: self.animate_text(connecting), .05)
        back = ImageButton(pos_hint={"x": .01, "top": .99}, size_hint=(0.1, 0.1), source="Images/back.png")
        back.bind(on_press=lambda button: switch("Play", self.manager, "right"))
        self.add_widget(back)
        self.requested_username = ""
        self.requested_friend = ""

    # This method makes the colour red flow through the text.
    def animate_text(self, label):
        if label not in self.children:
            return
        s = "Attempting to connect..."
        self.tracker += 1
        if self.tracker == len(s):
            self.tracker = 0
        # Have darker reds towards the edges of the section of text being coloured.
        label.text = s[:self.tracker] + "[color=550000]" + s[self.tracker:self.tracker + 1] + "[/color]" + \
            "[color=990000]" + s[self.tracker + 1:self.tracker + 2] + "[/color]" + \
            "[color=ff0000]" + s[self.tracker + 2:self.tracker + 3] + "[/color]" + \
            "[color=990000]" + s[self.tracker + 3:self.tracker + 4] + "[/color]" + \
```

```
            "[color=550000]" + s[self.tracker + 4:self.tracker + 5] + "[/color]" + \
            s[self.tracker + 5:]

    # This method sends information to the server about username, friends and characters.
    def initialise_connection(self):
        self.transport.write(("username;" + self.username + "\n").encode("utf-8"))
        characters = get_characters()
        characters_list = []
        friend_characters_list = []
        for character in characters:  # Create an ImageButton for each character
            if isfile("Images/" + character + "/section_locations.pickle"):  # If character is built-in
                continue
            elif isfile(characters[character]):  # If the character was made by the user.
                characters_list.append(character)
            else:
                friend_characters_list.append(character)
        self.transport.write(("custom_characters;" + ";".join(characters_list) + "\n").encode("utf-8"))
        self.transport.write(("friends;;" + ";".join(self.friends) + ";;" +
                              ";".join(friend_characters_list) + "\n").encode("utf-8"))

    # Called when the screen is entered.
    def on_pre_enter(self):
        if self.transport:
            self.initialise_connection()

    # This method creates a popup allowing the user to enter a username.
    def set_username(self):
        popup_layout = GridLayout(cols=1, spacing=Window.width * .01)
        popup_layout.add_widget(Label(text="Create a username", font_name="pixel_font.otf",
                                      font_size=Window.height * 0.07))
        username_input = TextInput(text="Username", multiline=False, font_name="pixel_font.otf",
                                   font_size=Window.height * 0.07)
        # Filter out certain punctuation.
        username_input.input_filter = lambda text, from_undo: "".join(char for char in text if char not in
                                                                      "|\\?*<\":>+[]/'")[:20 - len(username_input.text)]
        popup_layout.add_widget(username_input)
```

```
        confirm_button = Button(text="Confirm", background_normal='', background_color=(.5, 1, .5, .5),
                                font_name="pixel_font.otf", font_size=Window.height * 0.07)
        confirm_button.bind(on_press=lambda button: self.username_confirmed(username_input.text))
        cancel_button = Button(text="Cancel", background_normal='', background_color=(1, .5, .5, .5),
                               font_name="pixel_font.otf", font_size=Window.height * 0.07)
        cancel_button.bind(on_press=lambda button: (switch("Play", self.manager, "right"),
                                                    self.manager.remove_widget(self), self.username_popup.dismiss()))
        popup_layout.add_widget(confirm_button)
        popup_layout.add_widget(cancel_button)
        self.username_popup = Popup(pos_hint={"x": .1, "y": .2}, size_hint=(.8, .6), title="", auto_dismiss=False,
                                    content=popup_layout)
        self.username_popup.open()

    def username_confirmed(self, text):
        self.transport.write(("wants_username;" + text + "\n").encode("utf-8"))
        self.requested_username = text

    def friend_username_confirmed(self, text):
        if text in self.friends:
            Popup(pos_hint={"x": .2, "y": .35}, size_hint=(.6, .3), title="",
                  content=Label(text="Already your friend", font_name="pixel_font.otf",
                                font_size=Window.height * 0.05)).open()
        elif text == self.username:
            Popup(pos_hint={"x": .15, "y": .35}, size_hint=(.7, .3), title="",
                  content=Label(text="You can't be your own friend", font_name="pixel_font.otf",
                                font_size=Window.height * 0.05)).open()
        else:
            self.transport.write(("add_friend;" + text + "\n").encode("utf-8"))
            self.requested_friend = text

    @staticmethod
    def username_taken():
        Popup(pos_hint={"x": .2, "y": .35}, size_hint=(.6, .3), title="",
              content=Label(text="Username taken", font_name="pixel_font.otf", font_size=Window.height * 0.05)).open()

    # This method locally saves the username.
```

```
    def username_accepted(self):
        self.username = self.requested_username
        with open("online_info.ini", "r") as online_info:
            info_list = online_info.read().split("\n")
            info_list[info_list.index("username") + 1] = self.username
        with open("online_info.ini", "w") as online_info:
            online_info.write("\n".join(info_list))
        self.username_popup.dismiss()
        self.initialise_connection()

    @staticmethod
    def no_friend():
        Popup(pos_hint={"x": .2, "y": .35}, size_hint=(.6, .3), title="",
              content=Label(text="Player doesn't exist", font_name="pixel_font.otf",
                            font_size=Window.height * 0.05)).open()

    # This method saves the friend locally.
    def added_friend(self):
        self.friends.append(self.requested_friend)
        with open("online_info.ini", "r") as online_info:
            info_list = online_info.read().split("\n")
            if info_list[info_list.index("friends") + 1] != "":
                info_list[info_list.index("friends") + 1] += ";"
            info_list[info_list.index("friends") + 1] += self.requested_friend
        with open("online_info.ini", "w") as online_info:
            online_info.write("\n".join(info_list))
        Popup(pos_hint={"x": .2, "y": .35}, size_hint=(.6, .3), title="",
              content=Label(text="Added friend", font_name="pixel_font.otf",
                            font_size=Window.height * 0.05)).open()
        self.initialise_connection()  # Send the new changed data to the server.

    # Only do this once connected to the server.
    def create_buttons(self):
        self.clear_widgets()
        back = ImageButton(pos_hint={"x": .01, "top": .99}, size_hint=(0.1, 0.1), source="Images/back.png")
        back.bind(on_press=lambda button: switch("Play", self.manager, "right"))
```

```
        host_button = Button(size_hint=(.6, .3), pos_hint={"x": .2, "y": .55}, background_normal='',
                             background_color=(.5, .5, .5, .5), text="Host Game", font_name="pixel_font.otf",
                             color=(0, 0, 0, 1), font_size=Window.height * 0.15)
        if self.manager.has_screen("Lobby"):
            self.manager.remove_widget(self.manager.get_screen("Lobby"))
        self.manager.add_widget(Lobby("Lobby", self.username, self.client_factory.protocol, self.transport))
        host_button.bind(on_press=lambda button: switch("Lobby", self.manager, "left"))
        join_button = Button(size_hint=(.6, .3), pos_hint={"x": .2, "y": .15}, background_normal='',
                             background_color=(.5, .5, .5, .5), text="Join Game", font_name="pixel_font.otf",
                             color=(0, 0, 0, 1), font_size=Window.height * 0.15)
        if self.manager.has_screen("Join"):
            self.manager.remove_widget(self.manager.get_screen("Join"))
        self.manager.add_widget(Join("Join", self.username, self.friends, self.client_factory.protocol, self.transport))
        join_button.bind(on_press=lambda button: switch("Join", self.manager, "left"))
        self.create_friends_popup()
        friends_button = Button(size_hint=(.2, .1), pos_hint={"x": .8, "y": .9}, background_normal='',
                                background_color=(.5, .7, .5, .5), text="Friends", font_name="pixel_font.otf",
                                color=(0, 0, 0, 1), font_size=Window.height * 0.05)
        friends_button.bind(on_press=lambda button: self.friends_popup.open())
        float_layout = FloatLayout()
        self.add_widget(float_layout)
        float_layout.add_widget(back)
        float_layout.add_widget(host_button)
        float_layout.add_widget(join_button)
        float_layout.add_widget(friends_button)

    # Create a popup for friends which will have two tabs.
    def create_friends_popup(self):
        friends_tabs = TabbedPanel(do_default_tab=False, tab_width=Window.width * .44, tab_height=Window.height * .1)
        friends_tab = TabbedPanelItem(text="Friends", font_name="pixel_font.otf", font_size=Window.height * .05)
        scroll = ScrollView(size_hint=(1, None), pos_hint={"x": .1, "y": .1},
                            size=(Window.width * .8, Window.height * .7))
        self.friends_layout = GridLayout(cols=3, size_hint_y=None, row_default_height=Window.height * .1,
                                         spacing=[Window.width * .001, Window.height * .01],
                                         padding=Window.height * .01)
        self.friends_layout.bind(minimum_height=self.friends_layout.setter('height'))
```

```
        friends_tab.add_widget(scroll)
        add_friends_tab = TabbedPanelItem(text="Add a friend", font_name="pixel_font.otf",
                                          font_size=Window.height * .05)
        add_friends_layout = GridLayout(cols=1, spacing=Window.width * .01, padding=Window.width * .05)
        add_friends_layout.add_widget(Label(text="Enter friend's username", font_name="pixel_font.otf",
                                            font_size=Window.height * 0.07))
        username_input = TextInput(text="Username", multiline=False, font_name="pixel_font.otf",
                                   font_size=Window.height * 0.07)
        username_input.input_filter = lambda text, from_undo: "".join(char for char in text if char not in
                                                                      "|\\?*<\":>+[]/'")[:20 - len(username_input.text)]
        add_friends_layout.add_widget(username_input)
        add_friends_tab.add_widget(add_friends_layout)
        confirm_button = Button(text="Confirm", background_normal='', background_color=(.5, 1, .5, .5),
                                font_name="pixel_font.otf", font_size=Window.height * 0.07)
        confirm_button.bind(on_press=lambda button: self.friend_username_confirmed(username_input.text))
        add_friends_layout.add_widget(confirm_button)
        friends_tabs.add_widget(friends_tab)
        friends_tabs.add_widget(add_friends_tab)
        scroll.add_widget(self.friends_layout)
        self.friends_popup = Popup(pos_hint={"x": .05, "y": .05}, size_hint=(.9, .9), content=friends_tabs, title="",
                                   separator_height=0)
        self.friends_popup.bind(on_open=lambda popup: self.transport.write(
            ("check_status;" + ";".join(self.friends) + "\n").encode("utf-8")))
        friends_tab.bind(on_press=lambda tab: self.transport.write(
            ("check_status;" + ";".join(self.friends) + "\n").encode("utf-8")))

    # Update the friends popup with the current statuses of the user's friends.
    def friends_statuses(self, message):
        self.friends_layout.clear_widgets()
        friends = message.split(";;")[1:-1]
        for friend in friends:
            friend = friend.split(";")
            if friend[0] == "":
                if len(friends) == 1:
                    self.friends_layout.add_widget(Button(background_normal='', background_down='',
                                                          background_color=(1, 1, 1, .8), font_name="pixel_font.otf",
```

```
                                                color=(0, 0, 0, 1), font_size=Window.height * 0.025,
                                                text="You have no friends"))
                    return
                continue
            self.friends_layout.add_widget(Button(background_normal='', background_down='',
                                                  background_color=(1, 1, 1, .8), font_name="pixel_font.otf",
                                                  color=(0, 0, 0, 1), font_size=Window.height * 0.025, text=friend[0]))
            if friend[1].startswith("in_lobby"):
                friend[1] = friend[1].split("-")
                self.friends_layout.add_widget(Button(background_normal='', background_down='',
                                                      background_color=(1, 1, 1, .8), font_name="pixel_font.otf",
                                                      color=(0, 0, 0, 1), font_size=Window.height * 0.025,
                                                      text="In a lobby: " + friend[1][2] + "/6"))
                self.friends_layout.add_widget(Button(font_name="pixel_font.otf", color=(1, 1, 1, 1),
                                                      font_size=Window.height * 0.025, text="Join",
                                                      on_press=lambda button:
                                                      self.transport.write(
                                                          ("request;" + friend[1][1] + "\n").encode("utf-8"))))
            else:
                self.friends_layout.add_widget(Button(background_normal='', background_down='',
                                                      background_color=(1, 1, 1, .8), font_name="pixel_font.otf",
                                                      color=(0, 0, 0, 1), font_size=Window.height * 0.025,
                                                      text=friend[1]))
                self.friends_layout.add_widget(Button(background_normal='', background_down='',
                                                      background_color=(0, 0, 0, 0)))

    # Called when the user is disconnected from the server.
    def disconnected(self, error_message):
        # If the user is no longer in multiplayer, don't interrupt them.
        if self.manager.current_screen != self and error_message.startswith("TCP connection timed out:"):
            return
        Popup(pos_hint={"x": .3, "y": .35}, size_hint=(.4, .3),
              content=Label(text="Connection Failed", font_name="pixel_font.otf",
                            color=(1, 1, 1, 1), font_size=Window.height * 0.05), title="").open()
        switch("Play", self.manager, "right")
        if self.manager.has_screen("Lobby"):
```

```
            self.manager.remove_widget(self.manager.get_screen("Lobby"))
        if self.manager.has_screen("Join"):
            self.manager.remove_widget(self.manager.get_screen("Join"))
        self.manager.remove_widget(self)
        del self
```

client.py

This is the program for the client class.

```
from misc import switch
from twisted.internet import protocol
from online_images import send_images
from os import makedirs
from twisted.internet.reactor import callLater


# Class that handles online communication with the server. Inherits the Protocol class from twisted.
class Client(protocol.Protocol):
    characters_received = False
    images = b''
    scheduled_timout = None

    # Called when the client connects to the server.
    def connectionMade(self):
        if self.scheduled_timout:
            self.scheduled_timout.cancel()
        self.transport.setTcpNoDelay(True)  # This prevents the Nagle algorithm which delays and coalesces messages
        self.factory.parent.transport = self.transport
        if self.username != "":  # If the user already has a username.
            self.factory.parent.initialise_connection()
        else:
            self.factory.parent.set_username()

    # Called when the client receives a message from the server.
    def dataReceived(self, data):
        try:
            # Messages always end in "\n" in case they get combined so they can be split up again.
            messages = data.decode('utf-8').split("\n")
        except UnicodeDecodeError:  # This means that the message is an image instead.
            messages = self.receive_images(data)
        for message in messages:
```

```
if message.startswith("unknown_characters"):
    message = message.replace("unknown_characters;", "")
    if message == "":  # If there are no unknown characters, start.
        self.factory.parent.create_buttons()
    else:
        self.send_images(message.split(";"))
elif message == "username_taken":
    self.factory.parent.username_taken()
elif message == "username_accepted":
    self.factory.parent.username_accepted()
elif message == "added_friend":
    self.factory.parent.added_friend()
elif message == "no_friend":
    self.factory.parent.no_friend()
elif message == "download_success":
    self.factory.parent.create_buttons()
elif message.startswith("friends_statuses"):
    self.factory.parent.friends_statuses(message)
elif message.startswith("teams"):
    message = message.replace("teams;", "")
    self.lobby_screen.update_slots(message)
elif message.startswith("lobbies"):
    message = message.replace("lobbies;", "")
    self.join_screen.show_lobbies(message)
elif message.startswith("request"):
    message = message.replace("request;", "")
    self.lobby_screen.request_received(message)
elif message == "accepted":
    switch("Lobby", self.join_screen.manager, "down")
elif message.startswith("start"):
    self.lobby_screen.start_game()
elif message.startswith("characters_chosen"):
    if not self.characters_received:  # Only do this once.
        self.characters_received = True
        self.character_selection.enter_game(message)
elif message.startswith("update"):
```

```
                self.game.receive_message(message.replace("update;", ""))

    def send_images(self, characters):
        data = send_images(characters)
        self.transport.write(data)

    def receive_images(self, data):
        self.images += data
        messages = []
        if b'start' in data:
            messages = self.images.split(b'[delimiter]')
            messages = ("".join(list(map(lambda x: x.decode("utf-8"), messages[:messages.index(b"start")])))).split(
                "\n")
        if b'end' in data:
            self.images = self.images.split(b'[delimiter]')
            # Capture any messages that were sent before the images
            messages = ("".join(list(map(lambda x: x.decode("utf-8"), self.images[:self.images.index(b"start")]))) +
                        "".join(list(map(lambda x: x.decode("utf-8"),
                                         self.images[self.images.index(b"end") + 1:])))).split("\n")
            for image in self.images[self.images.index(b"start") + 1:self.images.index(b"end")]:
                try:
                    pathname = image.decode("utf-8").replace("Character Storage", "Images")
                except UnicodeDecodeError:
                    try:
                        with open(pathname, "w+b") as file:
                            file.write(image)
                    # w+ mode creates the file if it doesn't exist but doesn't create the directory if it doesn't exist.
                    except FileNotFoundError:
                        makedirs("/".join(pathname.split("/")[:-1]))  # Create the directory.
                        with open(pathname, "w+b") as file:
                            file.write(image)
        return messages


# Factory for the client protocol. Inherits the ClientFactory class from twisted.
class ClientFactory(protocol.ClientFactory):
```

```
    protocol = Client

    def __init__(self, parent, username):
        self.protocol.username = username
        self.parent = parent

    def clientConnectionLost(self, connector, reason):
        connector.connect()  # Try to reconnect
        self.protocol.scheduled_timeout = callLater(5, connector.stopConnecting)  # Timeout after 5 seconds

    def clientConnectionFailed(self, connector, reason):
        self.parent.disconnected(reason.getErrorMessage())
```

join.py

This is the program for the join screen.

```
from kivy.uix.screenmanager import Screen
from kivy.uix.floatlayout import FloatLayout
from kivy.uix.gridlayout import GridLayout
from kivy.uix.scrollview import ScrollView
from kivy.uix.button import Button
from kivy.core.window import Window
from kivy.graphics import Rectangle
from misc import switch, ImageButton


# Screen for showing available lobbies.
class Join(Screen):
    def __init__(self, name, username, friends, protocol, transport):
        super().__init__()
        self.name = name
        self.username = username
        self.friends = friends
        self.protocol = protocol
        self.protocol.join_screen = self
        self.transport = transport
        back = ImageButton(pos_hint={"x": .01, "top": .99}, size_hint=(0.1, 0.1), source="Images/back.png")
        back.bind(on_press=lambda button: switch("Multiplayer", self.manager, "right"))
        self.refresh_button = ImageButton(pos_hint={"x": .8, "y": .9}, size_hint=(0.1, 0.1),
                                          source="Images/refresh.png")
        self.refresh_button.bind(on_press=lambda button: self.refresh())
        scroll = ScrollView(size_hint=(.8, None), pos_hint={"x": .1, "y": .1}, size=(Window.width*.8, Window.height*.8))
        self.canvas.add(Rectangle(pos=(Window.width*.1, Window.height*.1), size=(Window.width*.8, Window.height*.8)))
        self.grid_layout = GridLayout(cols=3, size_hint_y=None, row_default_height=Window.height*.1,
                                      spacing=[Window.width*.001, Window.height*.01], padding=Window.height*.01)
        self.grid_layout.bind(minimum_height=self.grid_layout.setter('height'))
        scroll.add_widget(self.grid_layout)
        float_layout = FloatLayout()
```

```
        self.add_widget(float_layout)
        float_layout.add_widget(back)
        float_layout.add_widget(self.refresh_button)
        float_layout.add_widget(scroll)

    # Called when the screen is entered.
    def on_pre_enter(self):
        self.transport.write("leave\n".encode("utf-8"))  # In case the user is still considered to be in a lobby.
        self.transport.write("list_lobbies\n".encode("utf-8"))

    def show_lobbies(self, message):
        self.refresh_button.color = (1, 1, 1, 1)
        self.grid_layout.clear_widgets()
        if message == "":
            self.grid_layout.add_widget(Button(background_normal='', background_down='', background_color=(0, 0, 0, .2),
                                               font_name="pixel_font.otf", color=(0, 0, 0, 1),
                                               font_size=Window.height * 0.025, text="There are no available lobbies"))
            return
        lobbies = message.split(";;")
        hosts = []
        for i in range(len(lobbies)):
            lobbies[i] = lobbies[i].split(";")
            if lobbies[i][0] in hosts or lobbies[i][0] == self.username:  # If the user is host of the lobby, ignore it.
                del lobbies[i]
            elif lobbies[i][0] in self.friends:  # If a friend is host, move it to the start of the list.
                lobbies.insert(0, lobbies.pop(i))
            hosts.append(lobbies[i][0])
        for lobby in lobbies[:100]:  # Get the first 100 lobbies.
            if int(lobby[1]) < 6:  # If the lobby is not full.
                self.grid_layout.add_widget(Button(background_normal='', background_down='', color=(0, 0, 0, 1),
                                                   background_color=(0, 0, 0, .2), font_name="pixel_font.otf",
                                                   font_size=Window.height * 0.025, text=lobby[0] + "'s game"))
                self.grid_layout.add_widget(Button(background_normal='', background_down='', color=(0, 0, 0, 1),
                                                   background_color=(0, 0, 0, .2), font_name="pixel_font.otf",
                                                   font_size=Window.height * 0.025, text=lobby[1] + "/6 players"))
                self.grid_layout.add_widget(Button(font_name="pixel_font.otf", color=(1, 1, 1, 1),
```

```
                                        font_size=Window.height * 0.025, text="Join",
                                        on_press=lambda button: self.transport.write(
                                            ("request;" + lobby[0] + "\n").encode("utf-8"))))

    def refresh(self):
        self.refresh_button.color = (.5, .5, .5, 1)
        self.transport.write("list_lobbies\n".encode("utf-8"))
```

lobby.py

This is the program for the lobby screen.

```
from kivy.uix.screenmanager import Screen
from kivy.uix.floatlayout import FloatLayout
from kivy.uix.gridlayout import GridLayout
from kivy.uix.button import Button
from kivy.uix.label import Label
from kivy.uix.popup import Popup
from kivy.uix.widget import WidgetException
from kivy.core.window import Window
from misc import switch, ImageButton
from character_selection import CharacterSelection


# Screen for showing the lobby the user is in.
class Lobby(Screen):
    def __init__(self, name, username, protocol, transport):
        super().__init__()
        self.name = name
        self.username = username
        self.protocol = protocol
        self.protocol.lobby_screen = self
        self.transport = transport
        leave_button = Button(background_normal='', background_color=(1, 1, 1, 1), text="Leave?",
                              font_name="pixel_font.otf", color=(0, 0, 0, 1), font_size=Window.height * 0.1)
        leave_button.bind(on_press=lambda button: self.leave(leave_popup))
        leave_popup = Popup(pos_hint={"x": .3, "y": .35}, size_hint=(.4, .3),
                            content=leave_button, title="")
        start_button = Button(background_normal='', background_color=(1, 1, 1, 1), text="Start?",
                              font_name="pixel_font.otf", color=(0, 0, 0, 1), font_size=Window.height * 0.1)
        start_button.bind(on_press=lambda button: self.start(button, start_popup))
        start_popup = Popup(pos_hint={"x": .3, "y": .35}, size_hint=(.4, .3),
                            content=start_button, title="")
        back = ImageButton(pos_hint={"x": .01, "top": .99}, size_hint=(0.1, 0.1), source="Images/back.png")
```

```
back.bind(on_press=lambda button: leave_popup.open())
self.start_button = Button(size_hint=(.3, .1), pos_hint={"x": .6, "y": .1}, background_normal='',
                           background_color=(.3, .4, .3, .7), text="Start", font_name="pixel_font.otf",
                           color=(0, 0, 0, 1), font_size=Window.height * 0.05, background_down="")
self.start_button.bind(on_press=lambda button: start_popup.open() if self.start_button.background_color ==
                       [.5, 1, .5, .7] else None)
team_1_btn = Button(size_hint=(.3, .1), pos_hint={"x": .1, "y": .7}, background_normal='',
                    background_color=(.5, .5, .5, .5), text="Team 1", font_name="pixel_font.otf",
                    color=(0, 0, 0, 1), font_size=Window.height * 0.05)
team_1_btn.bind(on_press=lambda button: self.change_team("team_1"))
team_2_btn = Button(size_hint=(.3, .1), pos_hint={"x": .6, "y": .7}, background_normal='',
                    background_color=(.5, .5, .5, .5), text="Team 2", font_name="pixel_font.otf",
                    color=(0, 0, 0, 1), font_size=Window.height * 0.05)
team_2_btn.bind(on_press=lambda button: self.change_team("team_2"))
self.team_slots = {"team_1": [], "team_2": [], "centre": []}
for i in range(3):
    self.team_slots["team_1"].append(Button(size_hint=(.3, .1), pos_hint={"x": .1, "y": (.55-i*.15)},
                                            background_normal='', background_down='',
                                            background_color=(1, 1, 1, 1), font_name="pixel_font.otf",
                                            color=(0, 0, 0, 1), font_size=Window.height * 0.05))
    self.team_slots["team_2"].append(Button(size_hint=(.3, .1), pos_hint={"x": .6, "y": (.55-i*.15)},
                                            background_normal='', background_down='',
                                            background_color=(1, 1, 1, 1), font_name="pixel_font.otf",
                                            color=(0, 0, 0, 1), font_size=Window.height * 0.05))
    for j in range(2):
        self.team_slots["centre"].append(Button(size_hint=(.15, .075), pos_hint={"x": .425,
                                                                                 "y": (.725-(2*i+j)*.1)},
                                                background_normal='', background_down='',
                                                background_color=(1, 1, 1, 1), font_name="pixel_font.otf",
                                                color=(0, 0, 0, 1), font_size=Window.height * 0.025))
float_layout = FloatLayout()
self.add_widget(float_layout)
self.add_widget(self.start_button)
float_layout.add_widget(back)
float_layout.add_widget(team_1_btn)
float_layout.add_widget(team_2_btn)
```

```
        for i in range(3):
            float_layout.add_widget(self.team_slots["team_1"][i])
            float_layout.add_widget(self.team_slots["team_2"][i])
            float_layout.add_widget(self.team_slots["centre"][i])
            float_layout.add_widget(self.team_slots["centre"][i+3])
        self.team_slots["centre"][0].text = self.username
        self.requests = []

    # Called when the screen is entered,
    def on_pre_enter(self):
        self.requests = []
        self.transport.write("create_lobby\n".encode("utf-8"))

    # Takes the message containing the teams and outs the players in their team's slots.
    def update_slots(self, message):
        message = message.split(";;")
        for i in range(3):
            message[i] = message[i].split(";")
            self.team_slots["team_1"][i].text = ""
            self.team_slots["team_2"][i].text = ""
            self.team_slots["centre"][i].text = ""
            self.team_slots["centre"][i+3].text = ""
        for i, user in enumerate(message[0]):
            self.team_slots["team_1"][i].text = user
        for i, user in enumerate(message[1]):
            self.team_slots["centre"][i].text = user
        for i, user in enumerate(message[2]):
            self.team_slots["team_2"][i].text = user
        if message[3] == self.username:
            try:
                self.add_widget(self.start_button)
            except WidgetException:
                pass
        else:
            try:
                self.remove_widget(self.start_button)
```

```
            except WidgetException:
                pass
        if message[1][0] == "":  # If there are no players in the centre
            if message[0][0] != "" and message[2][0] != "":
                # If both teams have at least 1 player
                self.start_button.background_color = (.5, 1, .5, .7)
                self.start_button.background_down = "atlas://data/images/defaulttheme/button_pressed"
                return
        self.start_button.background_color = (.3, .4, .3, .7)
        self.start_button.background_down = ""

    def change_team(self, new_team):
        self.transport.write(("team;" + new_team).encode("utf-8"))

    def request_received(self, username):
        if username in self.requests:  # If the user has already asked to join.
            return
        else:
            self.requests.append(username)
        popup_layout = GridLayout(cols=1)
        popup_layout.add_widget(Label(text=username + " wants to join", font_name="pixel_font.otf", color=(1, 1, 1, 1),
                                      font_size=Window.height * 0.025))
        popup_layout.add_widget(Button(text="Accept?", font_name="pixel_font.otf", color=(1, 1, 1, 1),
                                       font_size=Window.height * 0.025, on_press=lambda button:
                                       self.transport.write(("accept;" + username).encode("utf-8")),
                                       on_release=lambda btn: (popup_layout.parent.parent.parent.dismiss(),
                                                               self.requests.remove(username) if username in
                                                               self.requests else None)))
        Popup(pos_hint={"x": .15, "y": .35}, size_hint=(.7, .3), title="", content=popup_layout,
              on_dismiss=lambda btn: self.requests.remove(username) if username in self.requests else None).open()

    def leave(self, popup):
        self.transport.write("leave\n".encode("utf-8"))
        switch("Multiplayer", self.manager, "right")
        popup.dismiss()
```

```
    def start(self, button, popup):
        if self.start_button.background_color == [.5, 1, .5, .7]:
            popup.remove_widget(button)
            popup.dismiss()
            self.transport.write("start\n".encode("utf-8"))

    def start_game(self):
        character_choices = 1
        team = "team_2"
        for i in range(3):
            if self.username == self.team_slots["team_1"][i].text:
                team = "team_1"
        if self.username == self.team_slots[team][0].text:
            for slot in self.team_slots[team]:
                if slot.text == "":
                    character_choices += 1
        self.manager.add_widget(CharacterSelection("Character Selection", character_choices=character_choices,
                                                   multiplayer=True, protocol=self.protocol, transport=self.transport,
                                                   username=self.username, team=team))
        switch("Character Selection", self.manager, "down")
```

online_images.py

This is a function used by both client and server to send images.

```
from image_editing import make_dict


# Function for formatting the images to be sent to and from the client and server.
def send_images(characters, path="Images"):
    data = b"[delimiter]start[delimiter]"
    for character in characters:
        if character:
            image_dict = make_dict(character, path=path)
            for value in image_dict.values():
                for pathname in value:
                    pathname = pathname.replace("Images", path)
                    with open(pathname, "rb") as image:
                        data += pathname.encode("utf-8") + b'[delimiter]' + image.read() + b'[delimiter]'
    data += b'[delimiter]end[delimiter]'
    return data
```

server.py

This is the program for the server class.

```
from twisted.internet import reactor, protocol
from os import makedirs, path
from online_images import send_images
from random import choice


# Class that handles online communication with clients. Inherits the Protocol class from twisted.
class Server(protocol.Protocol):
    username = None
    lobby = None
    team = None
    ingame = False
    images = b''
    previous_status = ""

    # Called when a message is received from a client.
    def dataReceived(self, data):
        try:
            messages = data.decode('utf-8').split("\n")
        except UnicodeDecodeError:
            messages = self.receive_images(data)
        for message in messages:
            if message.startswith("username") and not self.username:
                self.transport.setTcpNoDelay(True)
                self.username = message.replace("username;", "")
                self.factory.clients[self.username] = self
            elif message.startswith("wants_username"):
                if self.check_username(message):
                    self.transport.write("username_taken\n".encode("utf-8"))
                else:
                    self.transport.write("username_accepted\n".encode("utf-8"))
            elif message.startswith("add_friend"):
```

```
        if self.check_username(message):
            self.transport.write("added_friend\n".encode("utf-8"))
        else:
            self.transport.write("no_friend\n".encode("utf-8"))
    elif message.startswith("custom_characters"):
        self.custom_characters(message)
    elif message.startswith("friends"):
        self.get_friends_images(message)
    elif message.startswith("check_status"):
        self.friends_statuses(message)
    elif message == "list_lobbies":
        self.list_lobbies()
    elif message == "create_lobby" and not self.lobby:
        self.create_lobby()
    elif message.startswith("request"):
        self.pass_on_request(message)
    elif message.startswith("accept"):
        self.accept(message)
    elif message == "start":
        self.start()
    elif message == "leave":
        self.connectionLost(leaving=True)
    elif message.startswith("team"):
        team = message.partition(";")[-1]
        if len(self.factory.pending_lobbies[self.lobby][team]) == 3 or team == self.team:
            team = "centre"
        self.factory.pending_lobbies[self.lobby][self.team].remove(self.username)
        self.factory.pending_lobbies[self.lobby][team].append(self.username)
        self.team = team
        self.send_teams()
    elif message.startswith("characters"):
        self.character_selection(message)
    if message.startswith("update"):
        self.ingame = True
        self.update(message)
    else:
```

```
            self.ingame = False

    # Called when a client disconnects. This method handles them being removed from lobbies they were in.
    def connectionLost(self, reason=None, leaving=False):
        if self.lobby:
            if self.lobby in self.factory.lobbies:
                if self.username in self.factory.lobbies[self.lobby]:
                    message = self.previous_status.replace("update;", "").split(";;")
                    for i in range(len(message)):
                        message[i] = message[i].split(";")
                        if message[i][0] != "ball":
                            message[i][7] = "disconnected"
                        message[i] = ";".join(message[i])
                    message = "update;" + ";;".join(message)
                    self.update(message)
                    del self.factory.lobbies[self.lobby][self.username]
            # If they were the host of a pending lobby, either make the next player host or delete the lobby if there
            # are no other players.
            if self.factory.pending_lobbies[self.lobby]["host"] == self.username:
                if len(self.factory.pending_lobbies[self.lobby]["all_players"]) > 1:
                    self.factory.pending_lobbies[self.lobby]["host"] = \
                        self.factory.pending_lobbies[self.lobby]["all_players"][1]
                else:
                    del self.factory.pending_lobbies[self.lobby]
            self.factory.pending_lobbies[self.lobby][self.team].remove(self.username)
            self.factory.pending_lobbies[self.lobby]["all_players"].remove(self.username)
            self.send_teams()  # Update the teams for the other users.
            self.lobby = None
        if self.username and not leaving:
            del self.factory.clients[self.username]

    # Checks if the username has already been used.
    @staticmethod
    def check_username(message):
        username = message.partition(";")[-1]
        if path.isfile("Character Storage/" + username + ".txt"):
```

```
            return True
        else:
            return False

    # Checks if the server has the characters saved already and update the user's info file.
    def custom_characters(self, message):
        output = "unknown_characters;"
        characters = message.replace("custom_characters;", "").split(";")
        for character in characters:
            if not path.isdir("Character Storage/" + character):
                output += character + ";"
        with open("Character Storage/" + self.username + ".txt", "w+") as user_file:
            user_file.write("\n".join(characters))
        output += "\n"
        self.transport.write(output.encode("utf-8"))

    # Get the data of all the images of the user's friends that they don't have and send them.
    def get_friends_images(self, message):
        message = message.split(";;")[1:]
        friends = message[0].split(";")
        friend_images = message[1].split(";")
        characters = []
        for friend in friends:
            if friend == "":
                continue
            with open("Character Storage/" + friend + ".txt", "r") as images_file:
                characters.extend(images_file.read().split("\n"))
        characters = list(set(characters))  # Converting images to a set removes any duplicates
        for character in characters:
            if character in friend_images:
                characters.remove(character)
        data = send_images(characters, path="Character Storage")
        self.transport.write(data)

    # Format the message into images and save them.
    def receive_images(self, data):
```

```
        self.images += data
        messages = []
        if b'start' in data:
            messages = self.images.split(b'[delimiter]')
            messages = "".join(list(map(lambda x: x.decode("utf-8"), messages[:messages.index(b"start")]))).split("\n")
        if b'end' in data:
            self.images = self.images.split(b'[delimiter]')
            # Capture any messages that were sent before the images
            messages = ("".join(list(map(lambda x: x.decode("utf-8"), self.images[:self.images.index(b"start")]))) +
                        "".join(list(map(lambda x: x.decode("utf-8"),
                                         self.images[self.images.index(b"end") + 1:])))).split("\n")
            for image in self.images[self.images.index(b"start") + 1:self.images.index(b"end")]:
                try:
                    pathname = image.decode("utf-8").replace("Images", "Character Storage")
                except UnicodeDecodeError:
                    try:
                        with open(pathname, "w+b") as file:
                            file.write(image)
                    except FileNotFoundError:
                        makedirs("/".join(pathname.split("/")[:-1]))
                        with open(pathname, "w+b") as file:
                            file.write(image)
        self.transport.write("download_success\n".encode("utf-8"))
        return messages

    # Get the current statuses of the user's friends and send them.
    def friends_statuses(self, message):
        friends = message.split(";")[1:]
        output = "friends_statuses;;"
        for friend in friends:
            output += friend + ";"
            if friend in self.factory.clients:
                if self.factory.clients[friend].lobby:
                    if self.factory.clients[friend].ingame:
                        output += "In a game;;"
                    else:
```

```
                    output += "in_lobby-" + \
                              self.factory.pending_lobbies[self.factory.clients[friend].lobby]["host"] \
                              + "-" + str(len(self.factory.pending_lobbies[self.factory.clients[friend].lobby]
                                              ["all_players"])) + ";;"
            else:
                output += "Online;;"
        else:
            output += "Offline;;"
    output += "\n"
    self.transport.write(output.encode("utf-8"))

# Send all the pending lobbies.
def list_lobbies(self):
    hosts = []
    for lobby in self.factory.pending_lobbies:
        if lobby in self.factory.lobbies:  # If the players are in a game.
            continue
        if len(self.factory.pending_lobbies[lobby]["all_players"]) > 0:
            if not self.factory.clients[self.factory.pending_lobbies[lobby]["host"]].ingame:
                hosts.append(self.factory.pending_lobbies[lobby]["host"] + ";" +
                             str(len(self.factory.pending_lobbies[lobby]["all_players"])))
        else:
            del self.factory.pending_lobbies[lobby]
    self.transport.write(("lobbies;" + ";;".join(hosts) + "\n").encode("utf-8"))

# Create a new pending lobby.
def create_lobby(self):
    self.lobby = len(self.factory.pending_lobbies) + 1
    self.factory.pending_lobbies[self.lobby] = {}
    self.factory.pending_lobbies[self.lobby]["host"] = self.username
    self.factory.pending_lobbies[self.lobby]["team_1"] = []
    self.factory.pending_lobbies[self.lobby]["team_2"] = []
    self.factory.pending_lobbies[self.lobby]["centre"] = [self.username]
    self.factory.pending_lobbies[self.lobby]["all_players"] = [self.username]
    self.team = "centre"
    self.send_teams()
```

```
    # Pass a request to join from a user to the host of the lobby they want to join.
    def pass_on_request(self, message):
        host = message.partition(";")[-1]
        if host in self.factory.clients:
            if self.factory.clients[host].lobby:
                self.factory.clients[host].transport.write(("request;" + self.username + "\n").encode("utf-8"))
            else:
                self.transport.write("fail\n".encode("utf-8"))
        else:
            self.transport.write("fail\n".encode("utf-8"))

    # Accept a request to join a lobby.
    def accept(self, message):
        player_name = message.partition(";")[-1]
        if player_name in self.factory.clients:
            if not self.factory.clients[player_name].lobby and \
                    len(self.factory.pending_lobbies[self.lobby]["all_players"]) < 6:
                self.factory.pending_lobbies[self.lobby]["centre"].append(player_name)
                self.factory.pending_lobbies[self.lobby]["all_players"].append(player_name)
                self.factory.clients[player_name].team = "centre"
                self.factory.clients[player_name].lobby = self.lobby
                self.factory.clients[player_name].transport.write("accepted\n".encode("utf-8"))
                self.send_teams()
        else:
            self.transport.write("fail\n".encode("utf-8"))

    # Start a game from the pending lobby.
    def start(self):
        try:
            for name in self.factory.pending_lobbies[self.lobby]["all_players"]:
                try:
                    self.factory.clients[name].transport.write("start\n".encode("utf-8"))
                except KeyError:
                    self.factory.pending_lobbies[self.lobby][self.team].remove(name)
        except KeyError:
```

```
            self.transport.write("fail\n".encode("utf-8"))
        self.factory.lobbies[self.lobby] = {}
        self.factory.lobbies[self.lobby]["characters"] = 0

    # Send the updated teams to all the users in the pending lobby.
    def send_teams(self):
        try:
            for name in self.factory.pending_lobbies[self.lobby]["all_players"]:
                try:
                    self.factory.clients[name]. \
                        transport.write(("teams;" +
                                         ";".join(self.factory.pending_lobbies[self.lobby]["team_1"]) + ";;" +
                                         ";".join(self.factory.pending_lobbies[self.lobby]["centre"]) + ";;" +
                                         ";".join(self.factory.pending_lobbies[self.lobby]["team_2"]) + ";;" +
                                         self.factory.pending_lobbies[self.lobby]["host"] + "\n")
                                        .encode("utf-8"))
                except KeyError:
                    self.factory.pending_lobbies[self.lobby]["all_players"].remove(name)
        except KeyError:
            pass

    # Send all the characters selected to the users in the lobby.
    def character_selection(self, message):
        message = message.split(";")
        if self.username not in self.factory.lobbies[self.lobby]:
            self.factory.lobbies[self.lobby][self.username] = []
            for character in message[1:]:
                self.factory.lobbies[self.lobby][self.username].append(character)
                if character != "team_1" and character != "team_2":
                    self.factory.lobbies[self.lobby]["characters"] += 1
        if self.factory.lobbies[self.lobby]["characters"] == 6:  # Once all the characters have been chosen.
            background = choice(["Images/[0.7]background_mud.png", "Images/[0.6]background_grass.png",
                                 "Images/[0.7]background_night.png", "Images/[0.3]background_snow.png"])
            try:
                for name in self.factory.lobbies[self.lobby]:
                    if name == "characters":
```

```
                        continue
                    try:
                        output = ""
                        for user in self.factory.lobbies[self.lobby]:
                            if user == "characters":
                                continue
                            output += user + ";"
                            for character in self.factory.lobbies[self.lobby][user]:
                                output += character + ";"
                            output += ";"
                        self.factory.clients[name].transport.write(
                            ("characters_chosen;" + output + ";" + background + "\n").encode("utf-8"))
                    except KeyError:
                        del self.factory.lobbies[self.lobby][name]
            except KeyError:
                self.transport.write("fail\n".encode("utf-8"))

    # Pass on the updates to all the users in the lobby.
    def update(self, message):
        self.previous_status = message
        try:
            for name in self.factory.lobbies[self.lobby]:
                if name == "characters" or name == self.username:
                    continue
                self.factory.clients[name].transport.write((message+"\n").encode("utf-8"))
        except KeyError:
            self.transport.write("fail\n".encode("utf-8"))


class ServerFactory(protocol.Factory):
    def __init__(self):
        self.protocol = Server
        self.clients = {}
        # self.pending_lobbies will be a dictionary with keys being lobbies and the values being dictionaries of each
        # lobby's teams and players.
        self.pending_lobbies = {}
```

```
        # self.lobbies will be a dictionary with keys being lobbies and values being dictionaries of each players
        # characters.
        self.lobbies = {}


# Start the server.
reactor.listenTCP(50000, ServerFactory())
reactor.run()
```