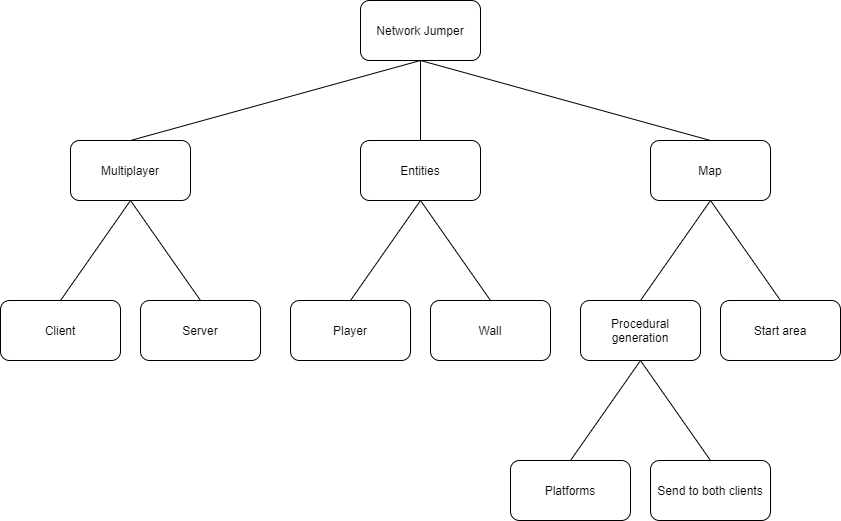
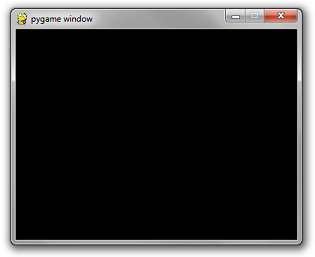
**Project Design Document – Joseph Henry**

**The whole project (basic decomposition)**

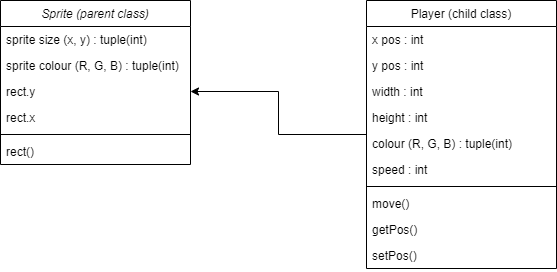
I have broken my project down into three main stages so that I can better understand and manage the problem. It will also allow me to analyse, design and solve the smaller parts of the problem before putting them all together to form a final product.



**Prototype 1 – (The multiplayer)**

**Problem decomposition**

1. Create a client program that will act as the main game loop. This client should define and render a basic display for the game to be displayed within.
2. Create a new file that will run a local server. This file should open a new socket that will listen on a given port for connections. This will then create a connection on a child socket with the client. This connection will be used to send data between the local server and the client.
3. Create a new function that will take in an ip address and port of the server wanting to be joined. It should then take this information and attempt to send data to the requested server address. The game server should show the connection to the host so that they not only know that someone has connected but also who it is (via their ip).
4. Create a parent class of sprite and a child class called player. This will be used to define the character each client will be playing as, initialise them onto the screen and control all the movements. The parent class will also be useful later in the generation of platforms.



In pygame there is a default class for a sprite, I have used this to create my player class.

1. The local server should be able to hold data from clients in the form of an object.
2. The client should ask the network function to send its player’s object data to the local server. The local server will then send back the other player’s position to the original client. This will allow both clients to have both players positions at all times which means both clients can keep track of the other client’s movements respectively and show them on screen by updating after each communication with the server.

**Main Algorithms**

1 - The first main algorithm used for this prototype is the transferring of object data between the clients. This will use a local network to transfer data between the machines.

Client 2



Client 1

Switch



Local server host

On the software side, once the game begins it should render its own clients player and then ask the server for the object data of any other players. The server will have to then return any other players data so that the client can draw them to the screen.

During gameplay, clients will constantly send their own players object data to the server in exchange for the new data sent by the other client. The clients will then update the position of the other player they have stored locally to the newely received position and then update them on the screen.

2 – Player’s movement algorithm. This algorithm will function by checking the state of keys on the keyboard. If one of the movement keys is pressed, its state will update to true. Once the state of a movement control key is updated to true the algorithm will change the players coordinate positions (dependant on which key: ‘a’ and ‘d’ change x position while ‘w’ and ‘s’ change y position) by the players speed set in the player class. It will then update the screen to show the players new position and check to see if the key is still pressed before either repeating or stopping.

def move(self):

keys = keys.get\_pressed()

if keys == d

self.x += self.speed

if keys == a

self.x -= self.speed

if keys == w:

self.y -= self.speed

if keys == s:

self.y += self.speed

**Prototype 2 – (Spawn zone and physics)**

**Problem decomposition**

1. Create better-suited movement controls. I will do this by implementing jumping and removing the up and down motions previously used (get rid of w to move up and s to move down and replace with w to jump).

def move(self):

keys = keys.get\_pressed()

if keys == d

self.x += self.speed

if keys == a

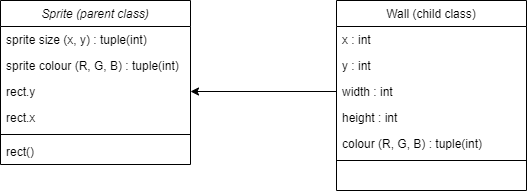
self.x -= self.speed

if keys == w and self.jumping == False:

self.jumping = True

self.jump ()

1. I will then need to make a new class called a wall. This will act as the platforms and the floor/roof/walls of the game.



Walls will be the only obstacle for players to encounter during play, as they have to traverse through a never-ending map without falling into gaps in the floor.

1. Set the walls spawn locations and size and then make the client render the spawn area on the game window by updating the screen.

Black bars are the wall sprites.

The player sprites

1. Add all wall sprites to a spritelist; this will make updating or drawing all walls at once easier than by updating the entire sprite list at once rather than each one individually. This can be easily done using pygame’s sprite.group() function e.g()

platformSpriteList = pygame.sprite.Group ()

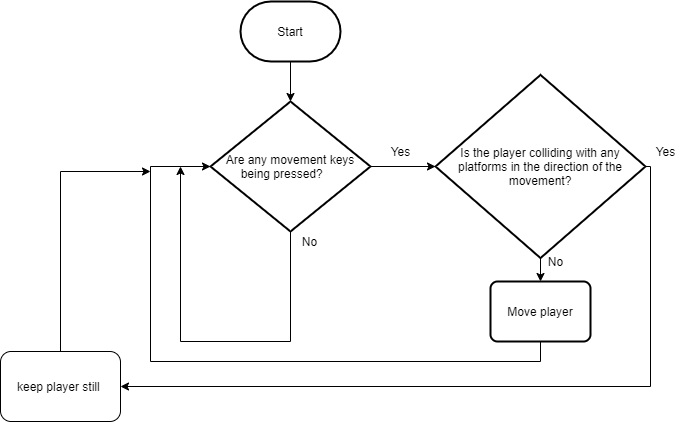
platformSpriteList.add (Entity)

platformSpriteList.draw (gameDisplay)

1. Use the spritelist to check if any players are colliding with any walls in the spritelist. If yes then stop their movement in that direction. If not then no change.
2. Add gravity to the player so that they do not float but rather fall until they have a platform/floor below them.

**Main Algorithms**

1 – Collision between players and the map:

**-**

If the player’s character is colliding with a wall in the direction it is trying to move, the movement should not be completed as the wall should block it. So I will use the sprite list to detect any collisions and this will tell the player whether they can or cannot move in the direction inputted.

Players should not be pushed through the floor or appear half in the floor.

2 – Gravity to the players:

Add a new attribute to the player class, gravity : int

Inside of the move method I will then add a constant down force of this gravity

This will constantly push the players character down until it collides with a floor below it, then the collision algorithm will step in and stop the gravity from moving the player down.

Players should not be pushed through the floor or appear half in the floor.

def move(self):

keys = keys.get\_pressed()

if keys == d

self.x += self.speed

if keys == a

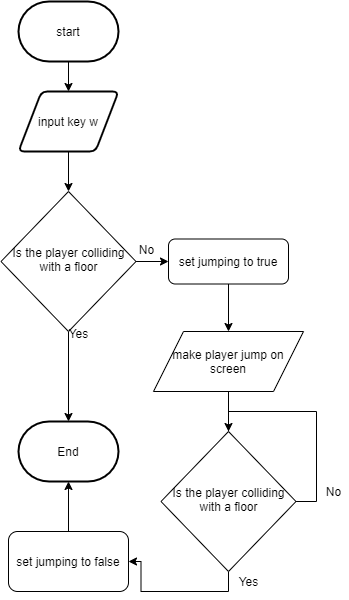
self.x -= self.speed

if keys == w and self.jumping == False:

self.jumping = True

self.jump ()

self.y += gravity

3 – Player jump movements:

If the player is already jumping or is falling (not on a platform) then jump should not be able to be completed. However once the player is touching a floor it will allow the player to jump again.

**Prototype 3 – (Map generation)**

**Problem decomposition**

1. Create a title screen so that players are not put straight into playing as soon as the game is launched. This should include a join server button, a start server button and a quit button as well as the title of the game.

Network Jumper

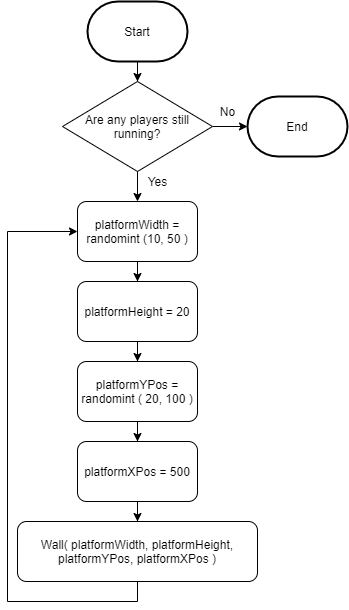
Start server

Join Server

Quit

1. Make the screen scroll to the right so that more of the map can be shown to the player to navigate through.
2. Generate new platforms procedurally so that there is variety to each play through and so that the map is infinite preventing players from running out of map to run through.
3. Once a platform goes off the viewable area, it should be deleted. This would reduce the chance of the game slowing down due to the large number of entities being stored in local memory.
4. Make holes in the floor at random intervals so that there is some way for the player to actually lose. Also, once the game does end return the player back to the title screen so they can chose whether to start a new game or to quit.

**Main Algorithms**

1 - Procedurally generated platforms:

Once a new platform is generated, it will be sent to the clients from the server and the clients will add the new wall sprite (platform) to the platformSpriteList. They will then be drawn to the screen for the users to interact with.

I will also add some checks in the generation to make sure platforms can always be reached. This will prevent platforms from being unusable.

This will also be used for the ground level of the map as there needs to be holes for players to fall through for there to be a way for the player to actually lose.

2 – Delete old platforms:

For platform in platformSpriteList:

If platform.x < 0:

platform.remove ()

Pygame has many built in methods for sprites, with remove () being used as a way to delete an instance of a sprite, so this will be used to remove any unwanted sprites.

3 – Screen scrolling:

Within the player class, I will add a new method called scrollPlayer () where it will reduce the x coordinate of the player by 1 moving it 1 pixel closer to the left of the screen.

def scrollPlayer ():

self.x -= 1

Within the wall class, I will add a new method called scroll () where it will also reduce the x coordinate of the wall by 1 moving it 1 pixel to the left. Both the walls and the players will move to the left at the same speed to give the effect of the screen scrolling to the right.

def scroll ():

self.x -= 1

The client will constantly call the players and walls to scroll by calling the scroll () to the whole platformSpriteList and scrollPlayer () to the clients own player. It only needs to be applied to their own client’s player as the players position coordinates will be transferred to to the other player, so it will still be shown on both players as they exchange positions.