# DAVE: A HTML5 GAME AND A STANDALONE LAN MULTIPLAYER GAME

3rd Year Project

Ritchie Shekleton
N00143569

## Abstract

The objective of the project was to explore new skills as programmers and understand some of the processes behind creating a platformer game in Game Maker Studio. There are thousands of platformer games and due to this, a lot of functionality could be emulated in order to create a working game prototype.
Dave is both a Single player and multiplayer game that allows two users to play the game on a local network or single player in a browser. The game progression is linear and the game levels differ depending on which version of the game is running. The game originally had a story which it followed but as development progressed it started to look too ambitious. The story was built around the character Dave, a young man who suffered from social anxiety, due to this his goal was to achieve day to day tasks such as shopping with as little interaction with others as possible.

# Acknowledgements

First, I would like to thank my project partner Claudiu, he has helped me get though three years of this course and will probably help me in the fourth. He has been a good friend and for that I thank him.
Second, I would like to thank our project supervisor John Montayne, without him I doubt our project would be anywhere near as good. He told us when we were not working hard enough and when we were spending too much time on unnecessary aspects of the project. Thanks for all the advice and support.
Third, I would like to thank all my mates in college for their help in testing, their opinions on our ideas and their support throughout the last three years.
Lastly, I would like to thank my family for their love and support for the last twenty-one years.

# Contents

# 1. Introduction

## 1.1 Overview of project Technologies

Dave is a game split into two different variations, one is multiplayer and the other single player. The multiplayer variation is played on a LAN server while the single player game is player on a webpage. The players can register and log in to play the game together while connected to the same network. The game is built in GML (Game Maker Language) which is a language derived from C++ and JS. All the game resources were created by the project team using various programs such as Photoshop and Spriter.

## 1.2 Project Approach

The work was divided fairly between the project partners, one was the main coder for the project and was responsible for the implementation of the game's main functionality. The other in charge of all the resource creation: Animations, Tilesets, sound. The work was done in iterations with the plan to add or improve functionality for the game each week. The two developers implemented the project in parts, each part contained game functionality and the implementation decision was decided between the two and the project supervisor every seven days.

## 1.3 Initial Challenges

One of the initial challenges was that this application was that the first group project the two programmers worked on. The work was divided evenly and the output prototypes had to be combined which led to small compatibility issues between code and asset. This was the first time the two ever worked on the development of a game. The college computers crashed while working in Game Maker, this was due to the college firewall blocking the access of Game Maker to a restricted part of the system, due to this most of the work had to be completed on the developer's personal computers.

## 1.4 Areas of Learning

The aim of the project was to create a fully functional game and learn new programming languages as well as learn new production skills such as animation and asset creation in their respective software. The main area of learning was the software. Spriter is an animation application that uses pre-drawn characters to create animations that can be outputted as sprite sheets. Learning this software was one of the more difficult parts. The character had multiple moving parts therefore the animation timeline was densely populated. This made it easy to make mistakes and even lose track of which tab in the timeline did what. Each individual action of the character was animated in Spriter and it took a couple of tires before something that looked good in game maker was produced. The output from Spriter was brought into Game Maker and was assigned to object that require the assets.

# 2. Feasibility Study and Requirements

## 2.1 User Roles

During the requirement analysis, the systems functionality is decided, in this task all the limitations that might impede the development process will need to be known based on tasks that the users want to be able to do in the game.
A User role defines permissions and capabilities of a user that interacts with a system and what they wish to be able to do on said platform.
The capabilities of each role give the developers the information required to determine what the users aim on the software is, the proficiency level of that user and how frequent the use of the platform is.

### 2.1.1 General Gamer:

The type of player that has a great level of proficiency on similar games, very familiar with the genre. Plays games regularly. Likes to be able to save progress as high scores and compete with others on the scoreboard.

### 2.1.2 Casual player:

The type of user with a fair level of proficiency on similar games, familiar with the genre. Plays games from time to time.

### 2.1.3 New player:

The type of player that has a poorer proficiency in similar games, unfamiliar with the genre. Played very few games.

### 2.1.4 Friends or Siblings:

A type of person with a decent proficiency in games, familiar enough with the genre. Plays games from time to time and prefers to play multiplayer with friends and or family.

## 2.2 Redefined User Roles

### 2.2.1 First time player:

The type of user that has very little proficiency in platformer games and is rather new to the gaming scene, prefers a more casual gaming experience and prefer simple to adapt to games since they are new users.

### 2.2.2 Frequent Player:

The type of user with a fair level of proficiency in platformer games. They would also have a good understanding of platformer mechanics and what to expect through their progression in a level. These players may be either pc games or console but due to experience with similar applications they adapt quickly to any game they touch.

### 2.2.3 Experienced Player:

The type of player with the highest level of proficiency for games in general. They have a great understanding of the game and play it to achieve highest possible score. They play games on a regular basis and are most likely to pay for any necessary edge to achieve peak performance.

## 2.3 User Requirements

### 2.3.1 Competitive player:

As a competitive player, I want to be able to load my high scores to a table that anyone can see so that I can see and compare my performance with other players in the world.

### 2.3.2 Casual player:

As a casual player, I want to be able to save and load my previous progress form game saves so that I can finish the game at my own pace.

### 2.3.3 Casual player:

As a casual player, I want to be able to revisit previous levels in the game so that I may replay levels that I really enjoyed.

### 2.3.4 Casual player:

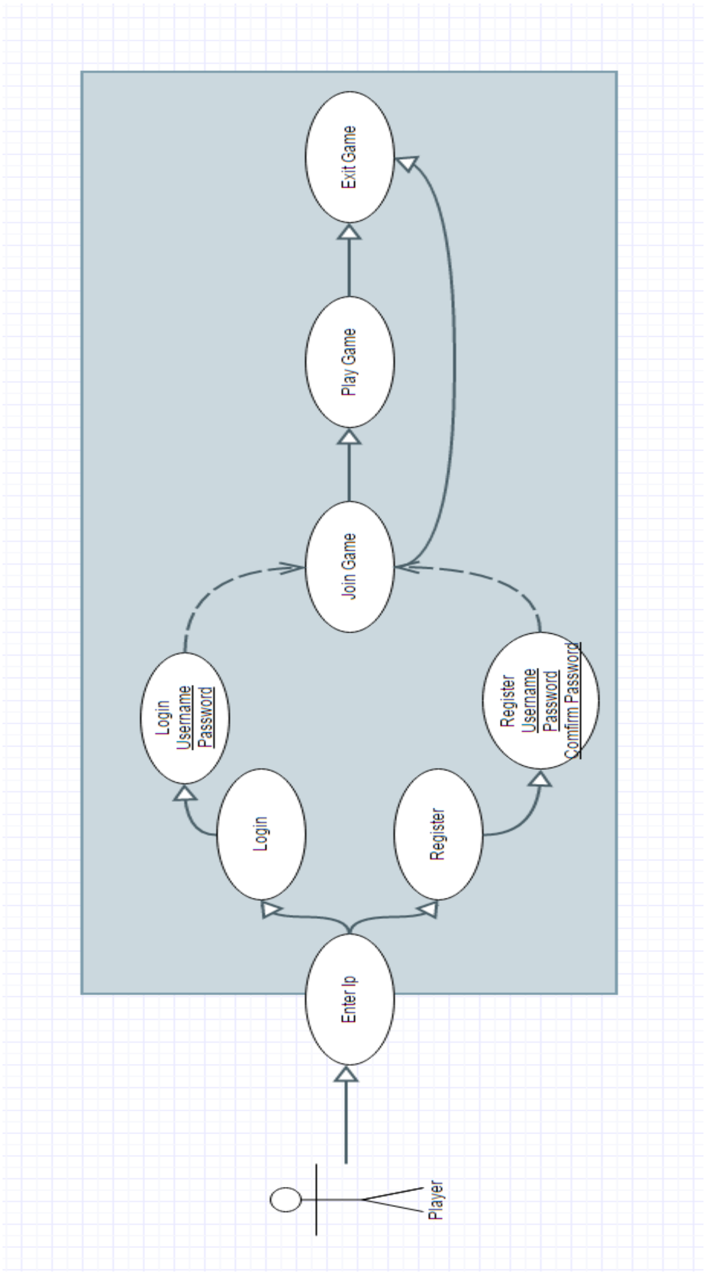
As a person who is more experienced in gaming I like to play games at a higher difficulty than the average player so that I can get the most enjoyment and challenge out of a game as possible.

### 2.3.5 Casual player:

As a casual player, I would like to be able to change difficulty mid game if the game seems too difficult so that I may complete the game.

## 2.3 Use Case Diagram

### 2.3.1 Multiplayer:

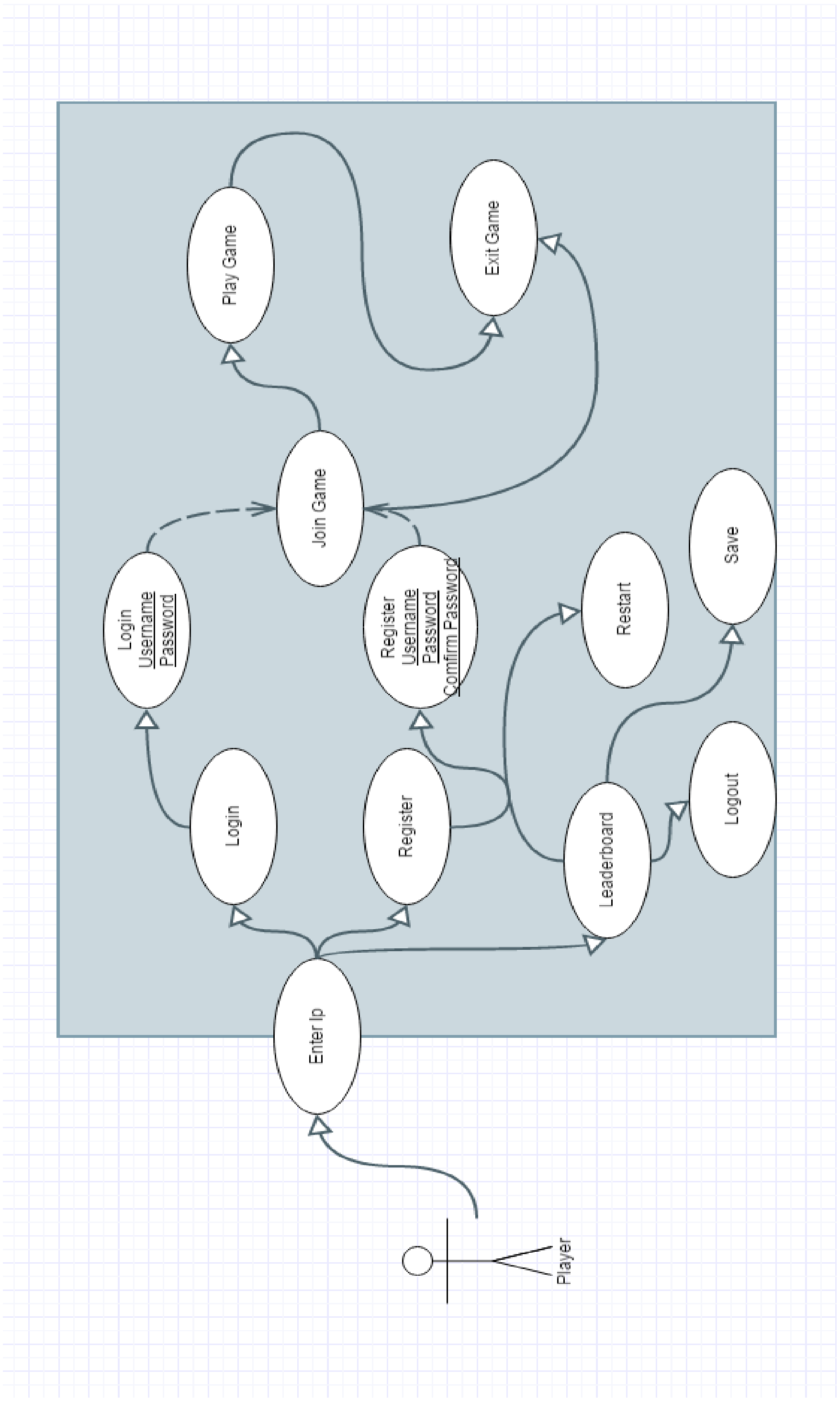


(figure 2.3.1)

The application developed is a game and will have one user which will be the player, each player will have the same abilities and rights as the other and is completely up to the player what he does.

Players of the multiplayer version of the game are able to connect to the server by entering the IP of the machine the server client is running on. They are able to login or register if they don't have an account. After creating an account, they may join a game and play with another person that is connected to the server.

### 2.3.2 Single Player:

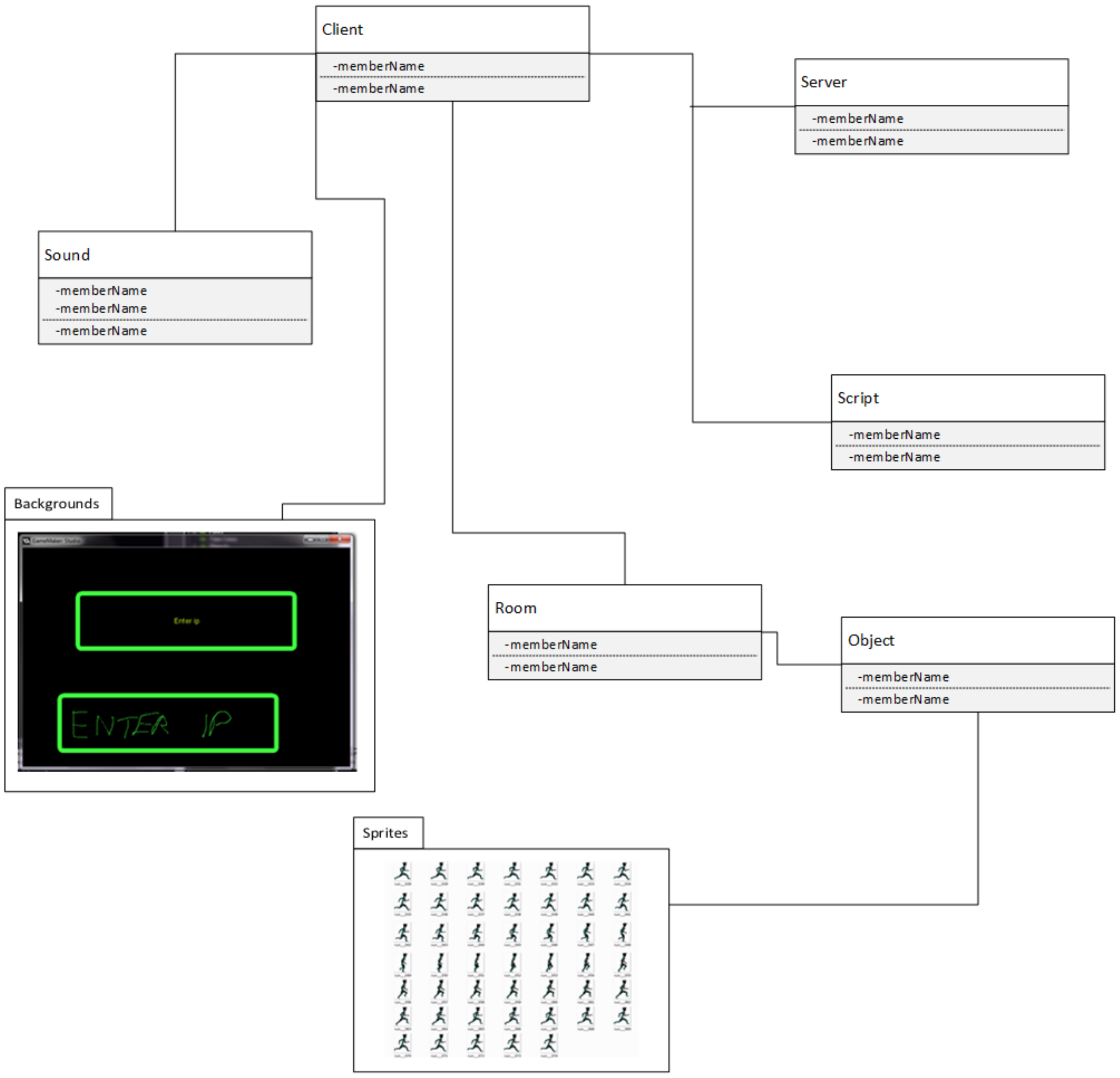


(figure 2.3.2)

The single player version of the game is exported to work in html5 and give some extra functionality such as check Leader-boards and save their scores to a database.

The game is virtually identical with few changes in gameplay and more functionality.

## 2.4 System Model



(figure 2.4.1)

There are multiple options available when exporting the game, the four main are an executable file for windows, html5, IOS and android. Each of these formats have different advantages and disadvantages.

### 2.4.1 Android

| ADVANTAGES | DISSADVANTAGES |
|---|---|
| Uses Java Coding | Touch controls in game |
| Largest user base | Not all phones will be able to run the app |
| Multiple phones from different manufacturers | |
| Developers have more options in terms of capabilities and access to the android market | |

The advantages of exporting as an android app are: uses java coding, largest user base and multiple phones from different manufacturers use android so there is no standardisation which gives a developer more options in terms of hardware capabilities and access to the android app market.

The disadvantages are having to use touch controls in the game, no standardisation is also a disadvantage since certain phones will not be able to run the app.

### 2.4.2 IOS

| ADVANTAGES | DISSADVANTAGES |
|---|---|
| Makes more money through micro transaction | Smaller market |
| Access to the iPhone market | Uses Swift |
| Smaller selection of phones | App has to be approved |

The advantages of exporting as a IOS app, makes more money through micro-transactions, access to iPhone market and it doesn't have as big of a selection of phones.

The disadvantages of iPhone are a smaller market, its own coding language and the app has to be approved.

### 2.4.3 Windows

| ADVANTAGES | DISSADVANTAGES |
|---|---|
| Basic Setting | None really |
| Export is all that is needed | |

The advantages to exporting for windows is that it's the basic setting, no extra things need to be done in order to export. There are no noticeable disadvantages to exporting for windows.

The advantages of using exporting to html5 are that it easy to update, easy to remove, advertising is easy to add and since it is a website it is easy to just share just give someone the link.

The main disadvantages are that the file must be hosted on a web server. This is because of browsers being unable to run the game or load assets because of cross-site scripting. The web server also stops an issue where sprites, backgrounds and text do not colour properly. The other issue is less than, greater than and equals give different results. For example, one third plus one third plus one third does not equal one because of floating point maths, it could be that it actually equals 1.00000001, so if you were trying to have an if statement equal to one it will never be true.
Due to these advantages and disadvantages the plan is making a single player version of the game work on a PHP enabled browser and a standalone multiplayer game for windows.

## 2.5 Feasibility Study

Gaming in general has been on the rise for the past couple of years and is still growing, with this new-found interest small game development companies have been emerging along with it. Due to the multiple ways to publish a game, practically anyone can develop a game and market it.
Problems that could arise during the development process are time management and unrealistic expectations of what can be done with Game Maker Studio. Lack of communication and unforeseen circumstances such as illness could also be an issue but both are highly unlikely.

## 2.6 Functional Requirements

Functional requirements will show the different things the system will do.

- The application will have a login screen which allows users to connect to the LAN server.
- The application will allow the player to play and record their progress on a database.
- The application will consist of one well-structured level that the player will attempt to complete in the shortest amount of time possible.
- The application will have a single player version that can be played on a web browser.
- The application will have somewhat intelligent enemy AI.
- The application resources will all be designed and animated by the project partners.

## 2.7 Non-Functional Requirements

- The proposed application will be easily accessible and will be split into two working apps. One will be multiplayer while the other Single-player. The application will be playable on pc only.
- The application’s gameplay will be based on already existing applications such as Mario and Limbo. The aesthetic design will be designed from the ground up and will be unique to the game.

# 3.Research and Analysis

## 3.1 New Technologies

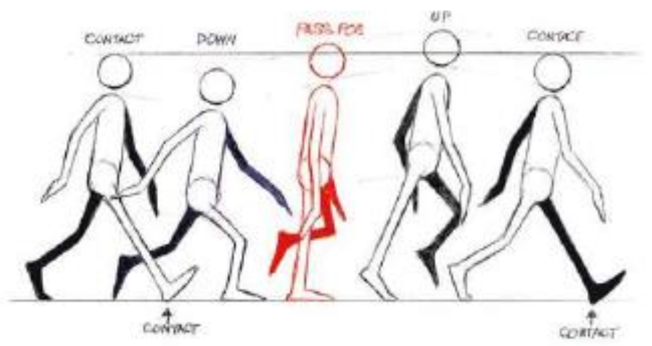
The plan for this project was to create a platformer game, create all the game resources and animate
Each from scratch, this means that some new technology was necessary to assist with the asset creation process. The game resources were first created using paper prototype, scanned and then imported into Photoshop, after the models were complete they were animated in Spriter.

## 3.2 Photoshop

Photoshop is a graphics editor developed and published by Adobe Systems for Window and macOS.
Photoshop was used to create most of the visual resources in the game. The game design is flat, simple and clean. Once the paper prototypes were finished, they were scanned and imported into Photoshop.
Once the main image was imported, the layering process began. Movable parts of a resource that was intended to be a sprite were placed on different layers, the layers were exported as separate images for the next step of the process in Spriter.

## 3.3 Spriter

Spriter is an animation tool from brashmonkey.com created to develop bone based sprite animations for making games. Photoshop exported the sprite into all of its separate moving pieces, these pieces were imported in Spriter and were put back together to make the sprite whole again. The layering had to be perfect so the body parts had to be placed in a specific order so that when movement was to occur it would not appear where it was not supposed to. When the sprite was assembled properly the bones were attached to their respective body part.
Bones work in a parent > child way, therefore when a parent is moved the child moves also but when a child is moved only the child and its sub-bones are moved.
Once all the bones were added to the animation process began and the gestures for each sprite were created, this includes walking animations, jumping etc.



(figure 3.3.1)

## 3.4 Game Maker

Game Maker is a game creation program which allows users to create cross-platform and multi genre video games using a scripting language known as Game Maker Language (GML). GML is primarily a scripting language, it is used to enhance and control the design of a game through more conventional programming, as opposed to drag and drop. GML is incredibly similar to C++ and JavaScript. Game Maker uses a physics engine called Box2D which is written in C++ and is used by multiple famous apps including Angry Birds.

## 3.5 Game Maker Studio Networking

### 3.5.1 TCP / IP / Server-to-Client Model / Client-to-Client Model

TCP/IP works by exchanging data packets (byte form) between IP addresses. These data packets can be sent, received and used to perform any sort of application one needs. UDP was another option that could be used because it is similar. Although UDP is faster than TCP it has the tendency to lose packets (which is why it could not be used). TCP works by setting up a host and having a client connect to a host in order to properly exchange data. This can be used in multiple ways, one being Server-to-Client and the other being Client-to-Client models.

A server is a host application that stores data and passes all data through itself and sends it to a single client or multiple clients, whichever is needed. A client is an application that sends/receives requests from the server it is connected to in order to exchange data with the server.
The server-to-client method works by having clients connect to a host and have the clients send data and make requests for data from the host. This method is more

secure than client-to-client methods. The reason for this being that the clients are not directly connected to each other, but connected to a server instead to act as a medium for data exchange.

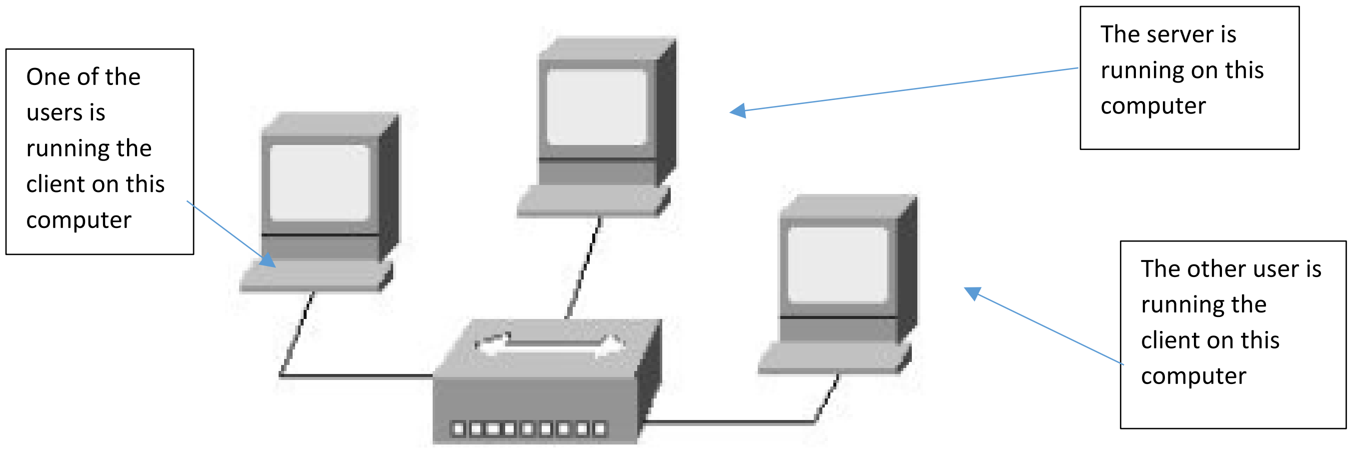
The client-to-client method works by having a client host a server on top of a client and have one or more clients connect to this host client. This is not as secure as the server-to-client method since clients are directly connected to each other and not a host server. The reason this is unsafe is that harmful clients can take information from the other clients and use it without their permission.
Game Maker Studio processes everything in steps which is the cap on FPS called room speed.
By default, room speed is 30, the game will then run at average 30 FPS. This limits the number of data packets that can be processed. An increase of room speed will also increase the number of packets that can be processed.

Sockets are IDs for clients, each socket is registered to a specific client and each client acquires a socket when it connects to the server. Sockets send data to the client they are registered to. Buffers are a form of data storage. Data is written to buffers and read from buffers. Data is read the same way you wrote it. So, if you wrote 1, 2, 3 in that order, you would read the data back out 1, 2, 3 in the same order.

### 3.5.2 Types of buffer:

*buffer_fixed*: Is a buffer with a fixed size. If the data size exceeds the buffer's size, the extra data will be removed.
*buffer_grow*: Is a buffer that will grow as data gets added to it. If the data size exceeds the buffer's size the buffer will increase its size in order to accommodate the extra data. The buffer will not go back to its original size if data is removed.
*buffer_wrap*: Is a buffer that wraps its data. If any data added to this type of buffer that would exceed its size, the data will insert itself at the start of the buffer and overwrite any data it overlaps. This buffer will not increase in size.
*buffer_fast*: Is a buffer that is like a faster buffer_fixed but can only hold buffer_u8 and buffer_s8 data types.
A node is a device that sends, receives or processes data in a telecommunication

network. A network is a combination of computers or other devices, which are referred to as nodes. The most commonly known network is the Ethernet network shown below it is also the one that the game is going to use.


One of the users is running the client on this computer
The server is running on this computer
The other user is running the client on this computer

(figure 3.5.1)

In this example the server is being run on a separate computer than the clients. This isn't necessary for this game but for more advanced games a dedicated server would generally be used. Each of the client objects will try to connect to the server by sending the necessary information through the router. The server will then send responses back to them. The computers will be constantly sending packages back and forth as the game is played.

## 3.6 Game Maker Studio Database

In order to save high scores/logins in the game, databases were looked into. There are a lot of different extensions available to get to add databases. Some of the extensions that used
MySQL couldn’t be used for this because they have serious security breaches when hooked up to game maker because any player could access the database. There were also DLLs that could be used but the ones relating to databases have been discontinued.

### 3.6.1 GMSDB

GMSDB which is a simple database written entirely in GML. It can be used in any project that requires a basic solution to retrieve and store organised data. It can perform the basic CRUD operations and also has custom filters, limits, sorting, calculations, offsets and joins. The drawback to this database is that it is in the programs memory, it only exists when the program is running. However, it is possible to save the database file and load it when the game starts.

### 3.6.2 gameSyncHS

gameSyncHS is an extension that allows the user to create a leader board and store it in a
MySQL database. In order to use this extension, you require;

- A PHP enabled web host that supports SQL.
- FTP access to the host, or access to a web based file manager.
- Rights to create and update a table on the database.
- A Device using the extension requires an internet connection to get the data from the webhost

## 3.7 Game Maker - Understanding motion

Game Maker comes with its own physics engine which is based on Box2D (written in C++), this physics engine cannot be accessed directly through game maker, it can however be overwritten.
Box2D is however available online and this is the version that was looked at to better understand how the physics used by the Game Maker software works. The measurements are in pixels.
Every level of a game takes place in a room, the coordinates of each room are given as follows, top left corner of a room is represented by (x, y) and the bottom right corner by (*room_width*, *room_height*)

### 3.7.1 GML move-functions

*move_random* (*hsnap, vsnap*)
Moves the instance to a random location in the room.
*move_twords_point(x, y, speed)*
This is a vector function as it sets the speed and direction of an instance. The instance will move towards (x, y) at a given speed.
*move_contact_all(dir, maxdist)*, *move_contact_solid(dir, maxdist)*
This moves the called instance in a direction (*dir*), up to (*maxdist*) pixels or until the

instance comes in contact with another instance. This does not have a set speed and is using the position assignment method to move the instance. The *move_contact_solid* works the same way but will only change direction when it comes in contact with a solid.
If only the direction of an instance is changed, it will move in that direction but the drawn sprites animation will remain in the same position. This can be fixed using the *image_angle = direction*.
Sprites are better to be drawn facing to the right so that the *image_angle* = 0.

### 3.7.2 Speed

In Game Maker speed is represented by pixels per step in a current direction.

Negative = moving ”backwards”

Positive = moving “forwards”

With respect to current direction.

### 3.7.3 HSpeed

Horizontal component of the instances speed pixels/step.

Negative = moving ”left”

Positive = moving “right”

### 3.7.4 VSpeed

Vertical component of the instances speed, pixels/step.

Negative = moving ”up”

Positive = moving “down”

This is not very strange since the coordinates of a room in GML ae (0, 0) top left.

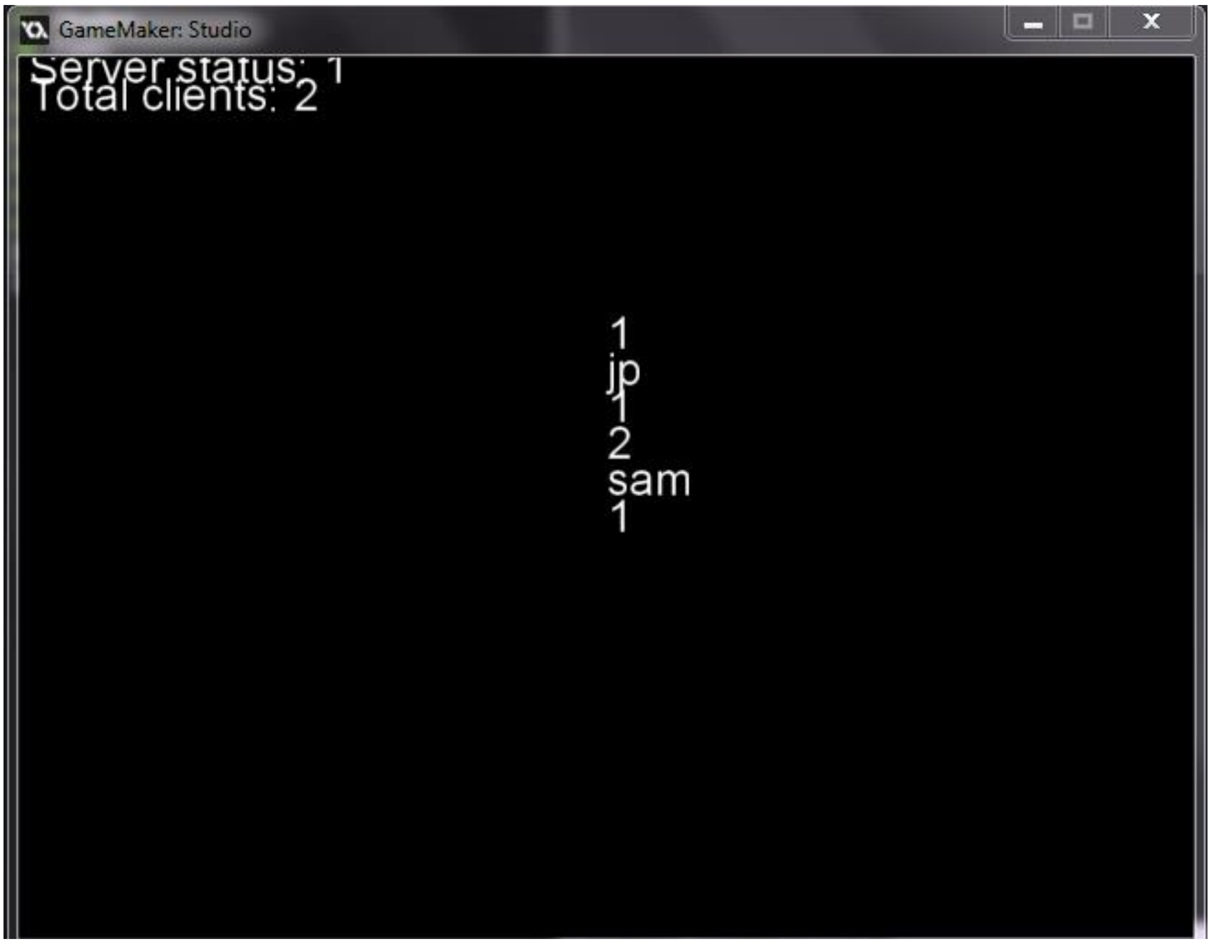
### 3.7.5 Gravity

Acceleration is a given direction (down==270) at a given rate. In Game Maker gravity is a vector.
A force which adds speed in a given direction. The vector is added to the instance each step, it can add up quickly and come to dominate the instances motion. In Game Maker gravity is set on a per-instance basis. This is very helpful since we can make it affect different objects in different ways.

Speed increase over time can give acceleration due to gravity a more pronounced effect but this can also mess up collision detection if the object is moving too fast. Gravity can be used in many different ways, giving the effect of wind and act as an opposing force on an object.
Multiple gravity forces may not affect a single instance, pseudo gravity may be used to do this.

# 4.Design

## 4.1 Server Design

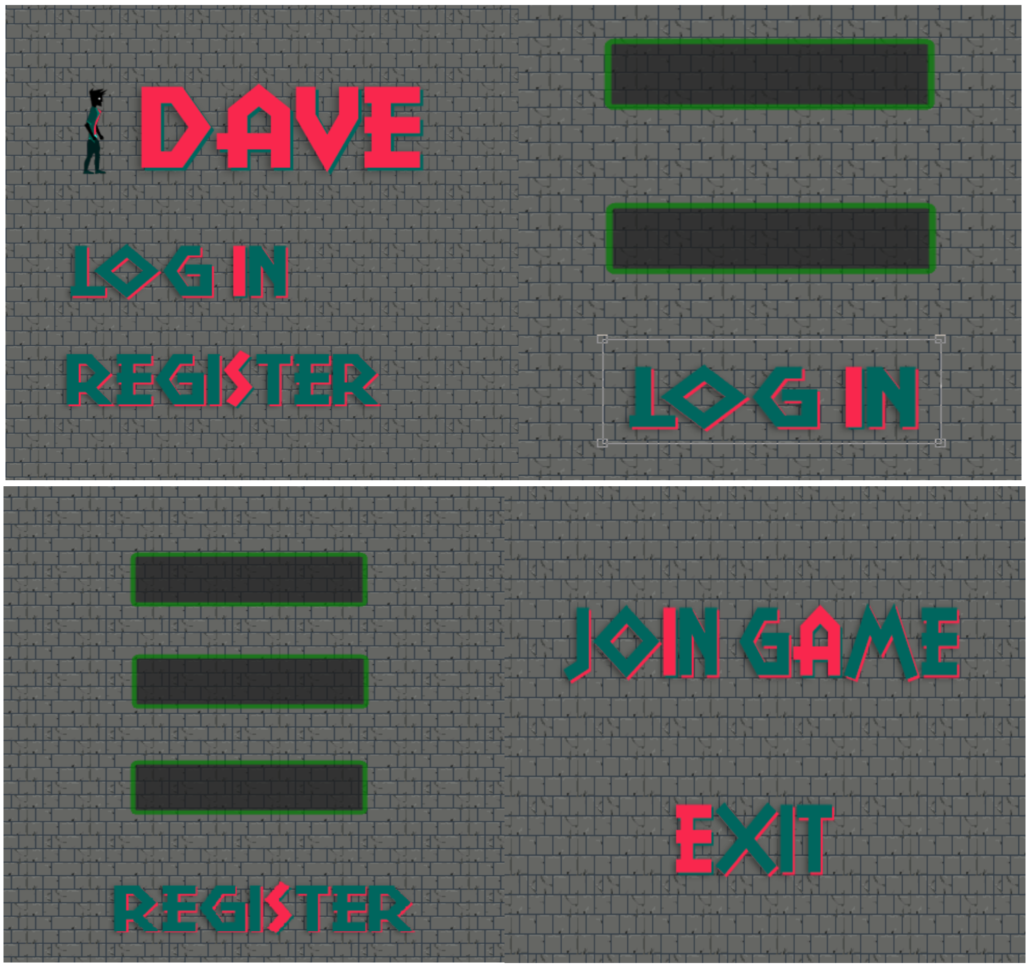


```
Server status: 1
Total clients: 2

1
jp
1
2
sam
1
```

(figure 4.1.1)

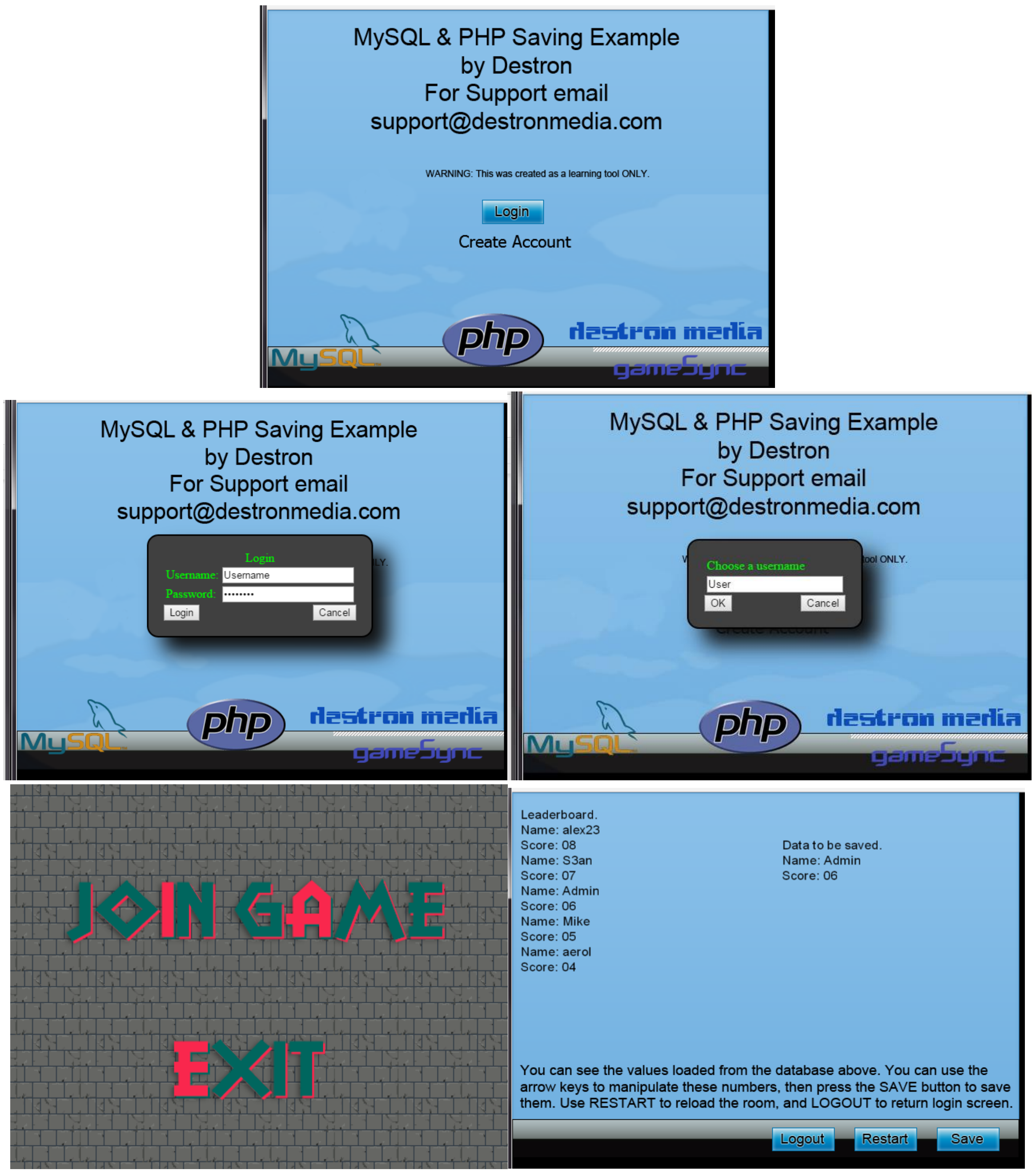
When the user executes the server file this screen pops up, it is a plain black background. The information in the centre of the screen is about the current clients on the server. Pop-up notifications also appear when the server is up, a client joins and disconnects. The server status is either a one or a zero, one if working and zero if it’s not. In the centre it contains the player info, the first number is the player’s id which is assigned when they connect to the server, the second is the player’s username and the third is if they are in the game world or not (one or zero).

## 4.2 Menu Design



(figure 4.2.1)

This is the current menu system for the client side of the multiplayer version, all of the design aspects of these menus are going to be changed but the functionality will remain the same. The first thing the user will see when they execute the game client is a screen which requires them to enter the IP of the computer that is running the server. The user is then greeted with a screen that offers two options, login and register. If the user selects the login button, they will then be asked to enter their username and password. If they had selected register they would be required to enter a username, a password and to confirm that password. If they succeeded they are sent to the main menu. They now have two options join game, which will allow the user to start playing the game, or exit which will close the window.


MySQL & PHP Saving Example
by Destron
For Support email
support@destronmedia.com

WARNING: This was created as a learning tool ONLY.

Leaderboard.
Name: alex23
Score: 08
Name: S3an
Score: 07
Name: Admin
Score: 06
Name: Mike
Score: 05
Name: aerol
Score: 04

Data to be saved.
Name: Admin
Score: 06

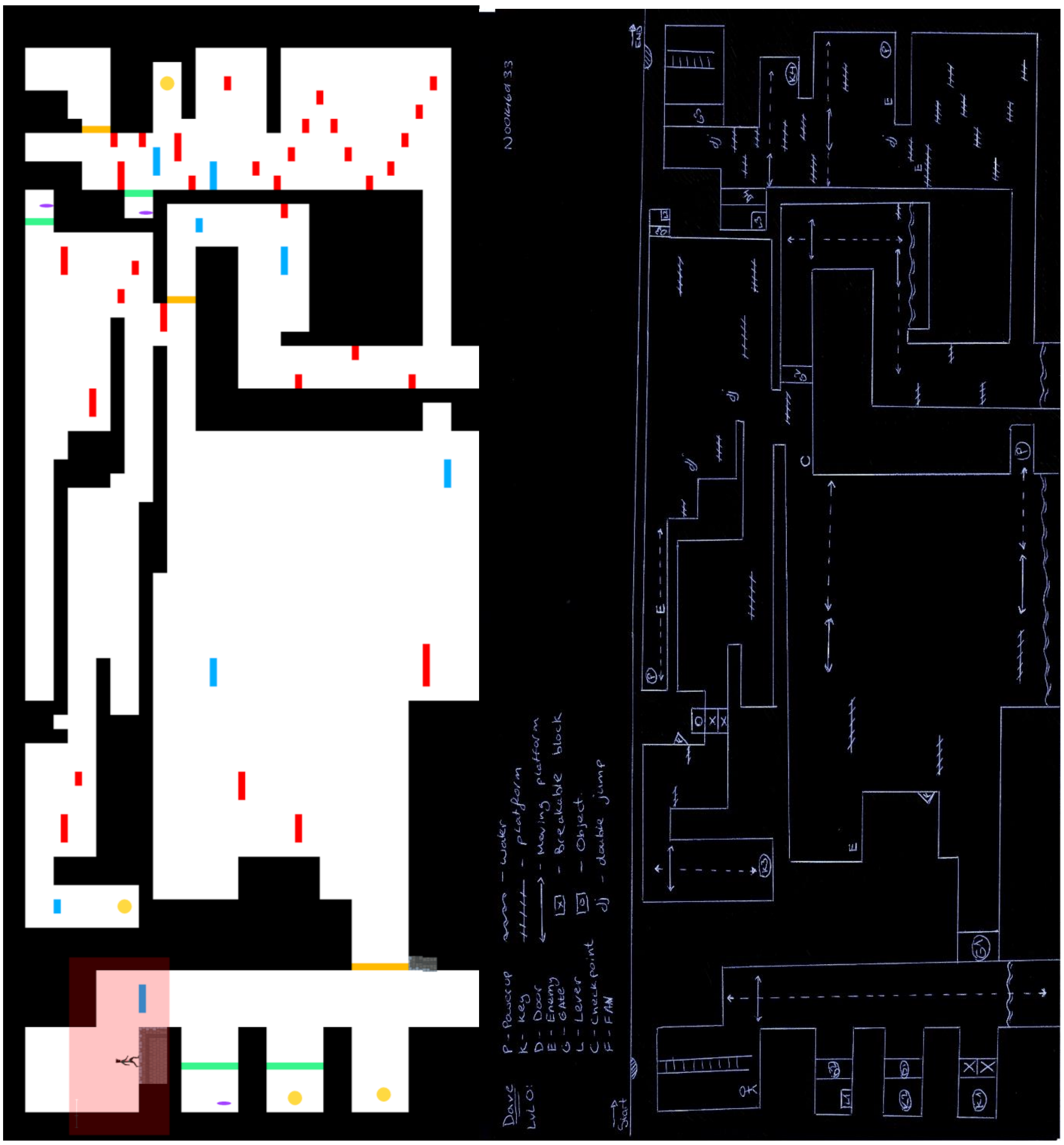
You can see the values loaded from the database above. You can use the arrow keys to manipulate these numbers, then press the SAVE button to save them. Use RESTART to reload the room, and LOGOUT to return login screen.

(figure 4.2.2)

This is the current menu system for the HTML5 version of the game, all of the design aspects of these menus are going to be changed because it is still using the menus from Destron media’s gameSync which was used in to connect to the database, but the functionality will remain the same. It starts off by offering the user two choices, create account and login. If they select login they have to input their username and their password. If they had selected create account they would be required to enter a username, a password and to confirm that password. After these screens the user is at the main menu, they can now enter the game, exit or enter the leader board screen. The leader board screen shows the two scores above and below the user's score. The three buttons in the leader board are to let the user logout which sends them back to the first screen. Restart which will update the leader board in case someone has recently finished the game. Save which will save the user's score.

## 4.3 Level Design

The game will have a multiplayer aspect and the level will be tweaked in a way as to make it impossible to finish if the two players don’t cooperate. The high score element of the game is a time based one, meaning that the faster they complete the level the higher on the board they will belong. The game takes place underground since Dave attempts to complete his task with as little human contact as possible. Because of this he attempts to get from point A to point B through the sewers. The sewers are like a maze with a series of obstacles set up to impede the player and make the game somewhat more challenging. Below is a blueprint of how the level will be designed, accompanied by a key of where each object is placed.



(figure 4.3)

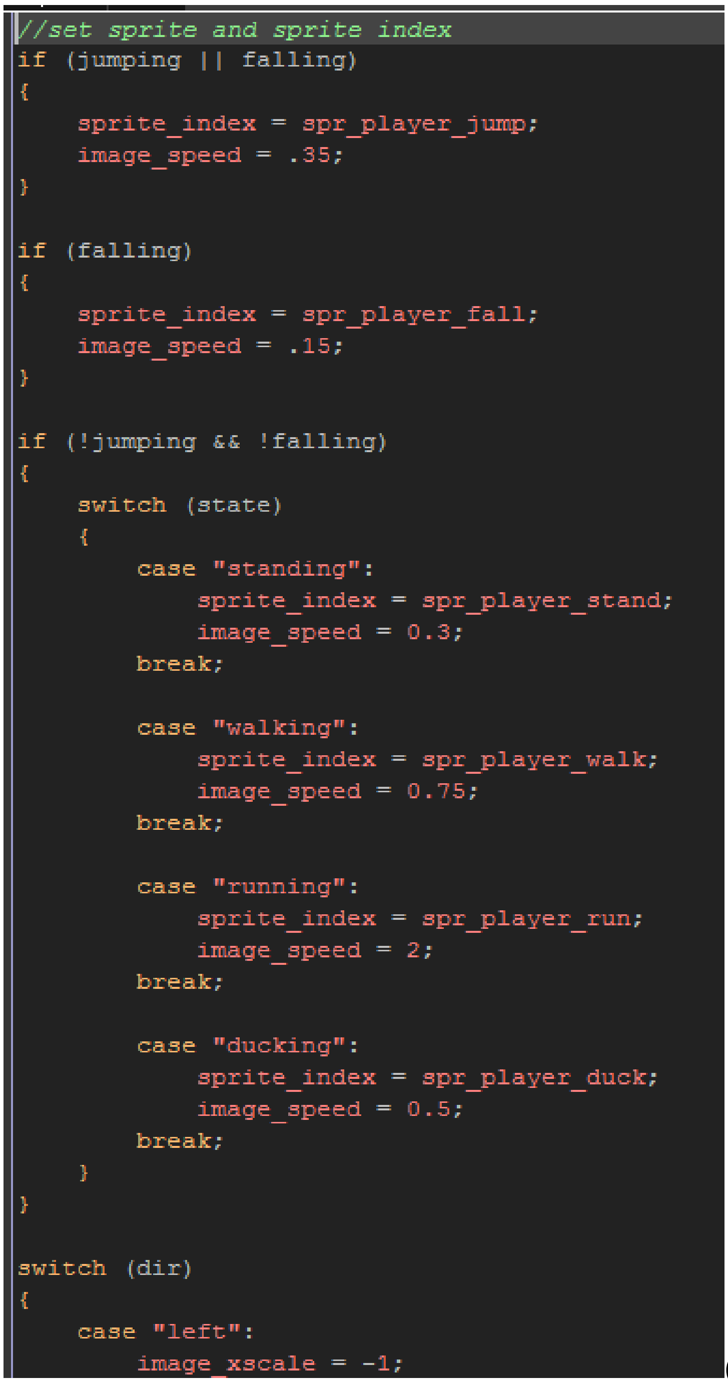
Red platforms are static and will be there to be used by the player to ascend or descend depending on what goal he is trying to achieve, in this case making it to the end. Blue platforms are non-static and move across from left to right or up and down, these platforms are implemented to add a bit more of a challenge. The purple ovals are lever which open gates. Gates are orange and are only opened when the right lever is active. Green diagonal lines are doors and are only opened when the right key is used, the keys are numbered and are represented by the yellow circles on the above illustration. The multiplayer world is slightly different and can only be completed if two players work together in order to get to the end.

# 5. Implementation

## 5.1 Common for both platforms

### 5.1.1 Scripts

#### *scr_setSprite*

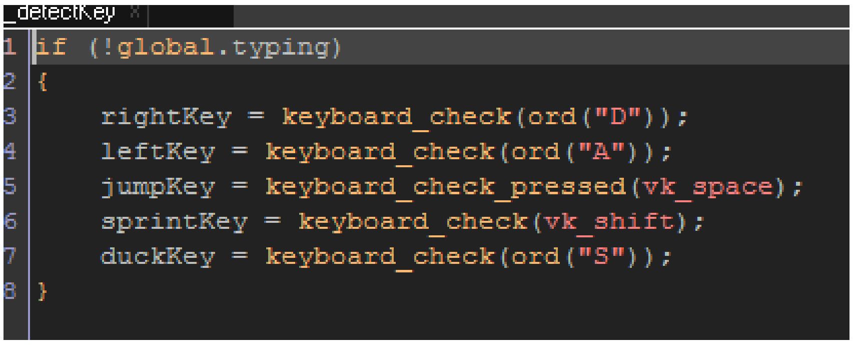


```
//set sprite and sprite index
if (jumping || falling)
{
    sprite_index = spr_player_jump;
    image_speed = .35;
}

if (falling)
{
    sprite_index = spr_player_fall;
    image_speed = .15;
}

if (!jumping && !falling)
{
    switch (state)
    {
        case "standing":
            sprite_index = spr_player_stand;
            image_speed = 0.3;
        break;

        case "walking":
            sprite_index = spr_player_walk;
            image_speed = 0.75;
        break;

        case "running":
            sprite_index = spr_player_run;
            image_speed = 2;
        break;

        case "ducking":
            sprite_index = spr_player_duck;
            image_speed = 0.5;
        break;
    }
}

switch (dir)
{
    case "left":
        image_xscale = -1;
```

(figure 5.1.1)

This script is used to change the player's sprite, image speed, which is how fast the sprites will change and the player's direction.

## scr_detectKey

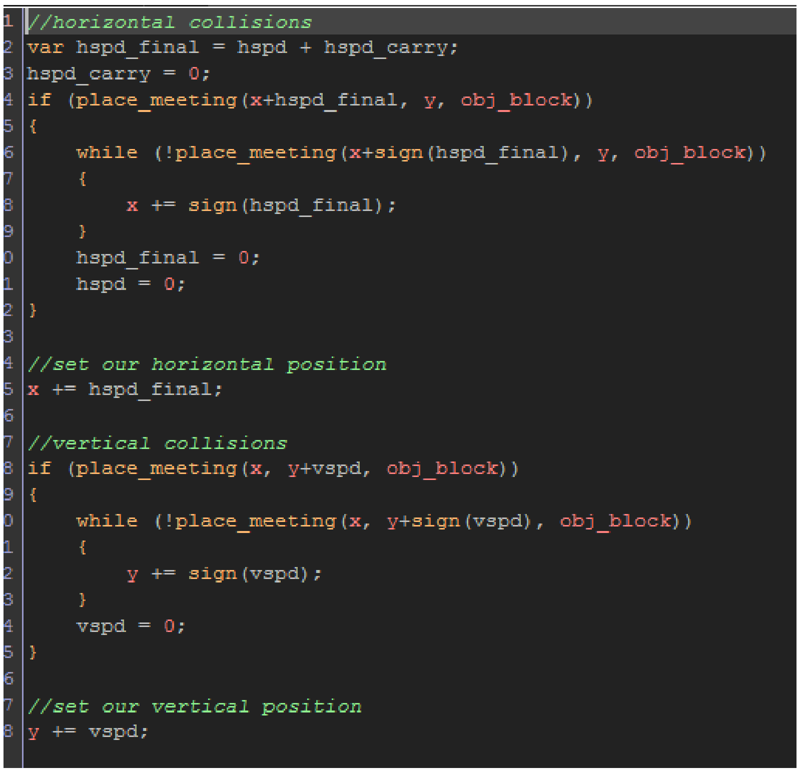


```
_detectKey
if (!global.typing)
{
    rightKey = keyboard_check(ord("D"));
    leftKey = keyboard_check(ord("A"));
    jumpKey = keyboard_check_pressed(vk_space);
    sprintKey = keyboard_check(vk_shift);
    duckKey = keyboard_check(ord("S"));
}
```

(figure 5.1.2)

This script gets the input from the user for the player’s controls.

## scr_collisionCheck

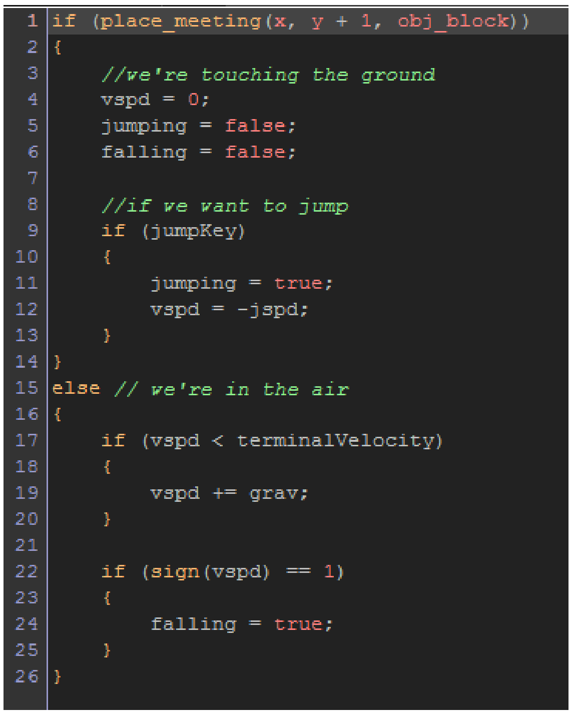


```
//horizontal collisions
var hspd_final = hspd + hspd_carry;
hspd_carry = 0;
if (place_meeting(x+hspd_final, y, obj_block))
{
    while (!place_meeting(x+sign(hspd_final), y, obj_block))
    {
        x += sign(hspd_final);
    }
    hspd_final = 0;
    hspd = 0;
}

//set our horizontal position
x += hspd_final;

//vertical collisions
if (place_meeting(x, y+vspd, obj_block))
{
    while (!place_meeting(x, y+sign(vspd), obj_block))
    {
        y += sign(vspd);
    }
    vspd = 0;
}

//set our vertical position
y += vspd;
```

(figure 5.1.3)

This script is for the player’s collision with the *obj_block* and its child objects. It checks if the player’s x position plus their final horizontal speed, their y position collides with an instance of the block object. It does the same for vertical collisions but the vertical speed is used instead.

## scr_groundCheck

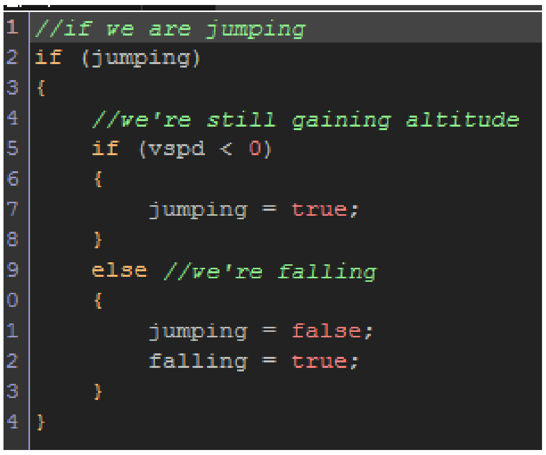


```
if (place_meeting(x, y + 1, obj_block))
{
    //we're touching the ground
    vspd = 0;
    jumping = false;
    falling = false;

    //if we want to jump
    if (jumpKey)
    {
        jumping = true;
        vspd = -jspd;
    }
}
else // we're in the air
{
    if (vspd < terminalVelocity)
    {
        vspd += grav;
    }

    if (sign(vspd) == 1)
    {
        falling = true;
    }
}
```

(figure 5.1.4)

This script is used to check if a player is touching the ground. The player is only able to jump if they are touching the ground. When jumping, and falling are equal to true the player's sprite will change to the jump and fall sprite respectively.

## scr_jumpCheck

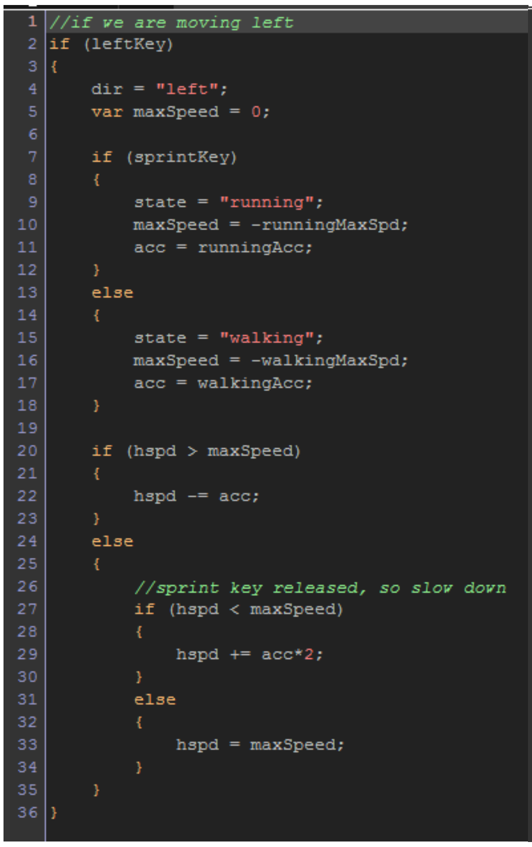


```
//if we are jumping
if (jumping)
{
    //we're still gaining altitude
    if (vspd < 0)
    {
        jumping = true;
    }
    else //we're falling
    {
        jumping = false;
        falling = true;
    }
}
```

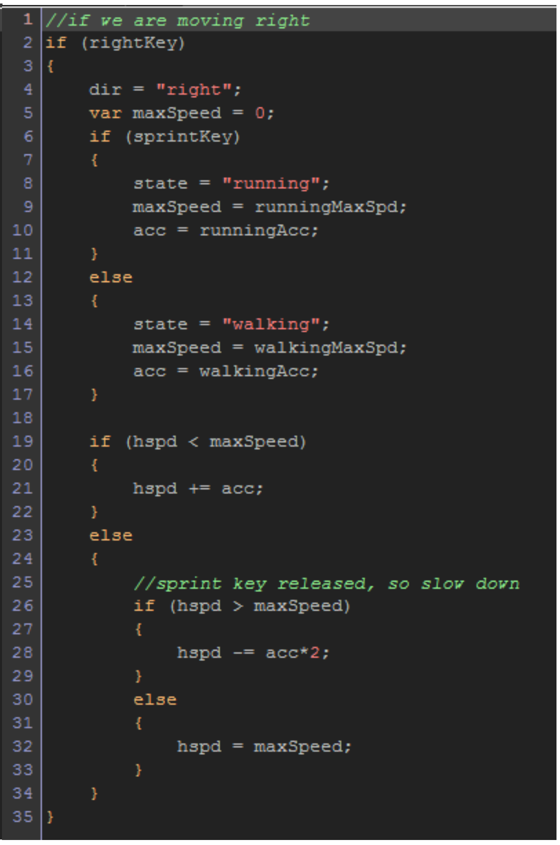
(figure 5.1.5)

This sprite is used to check if the player is jumping, when the player's vertical speed is greater than zero the player is still going up. If it is less than zero, the player is falling.

## scr_leftCheck and scr_rightCheck



```
//if we are moving left
if (leftKey)
{
    dir = "left";
    var maxSpeed = 0;

    if (sprintKey)
    {
        state = "running";
        maxSpeed = -runningMaxSpd;
        acc = runningAcc;
    }
    else
    {
        state = "walking";
        maxSpeed = -walkingMaxSpd;
        acc = walkingAcc;
    }

    if (hspd > maxSpeed)
    {
        hspd -= acc;
    }
    else
    {
        //sprint key released, so slow down
        if (hspd < maxSpeed)
        {
            hspd += acc*2;
        }
        else
        {
            hspd = maxSpeed;
        }
    }
}
```

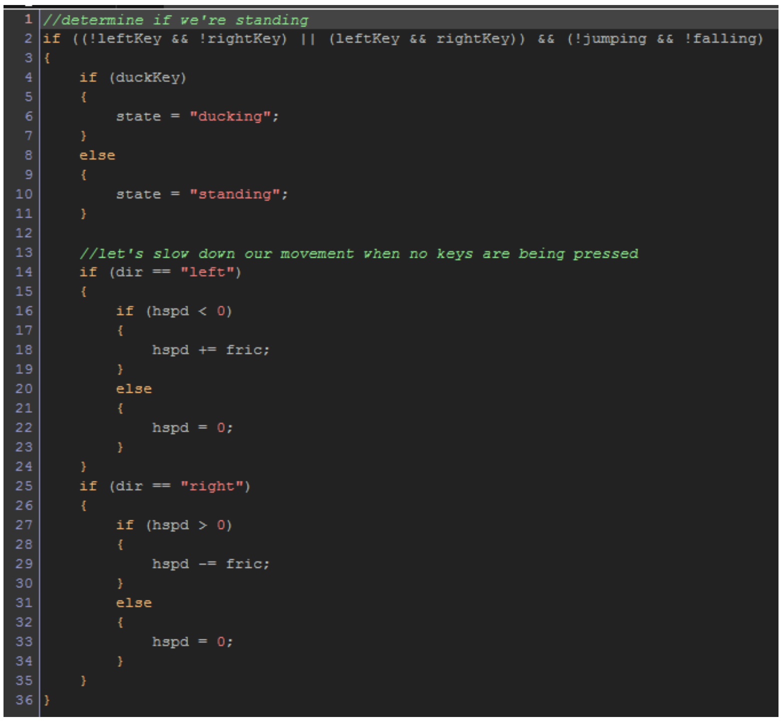


```
//if we are moving right
if (rightKey)
{
    dir = "right";
    var maxSpeed = 0;
    if (sprintKey)
    {
        state = "running";
        maxSpeed = runningMaxSpd;
        acc = runningAcc;
    }
    else
    {
        state = "walking";
        maxSpeed = walkingMaxSpd;
        acc = walkingAcc;
    }

    if (hspd < maxSpeed)
    {
        hspd += acc;
    }
    else
    {
        //sprint key released, so slow down
        if (hspd > maxSpeed)
        {
            hspd -= acc*2;
        }
        else
        {
            hspd = maxSpeed;
        }
    }
}
```

(figure 5.1.6)

These scripts are used when a player is moving right or left. If the user is pushing the sprint key, the state is changed to running and the running max speed and acceleration are used as well. When they are not pushing, the sprint key the state is walking and the walking max speed and acceleration is used. The player accelerates if their horizontal speed is less than their max speed.

### *scr_standCheck*



```
//determine if we're standing
if ((!leftKey && !rightKey) || (leftKey && rightKey)) && (!jumping && !falling)
{
    if (duckKey)
    {
        state = "ducking";
    }
    else
    {
        state = "standing";
    }

    //let's slow down our movement when no keys are being pressed
    if (dir == "left")
    {
        if (hspd < 0)
        {
            hspd += fric;
        }
        else
        {
            hspd = 0;
        }
    }
    if (dir == "right")
    {
        if (hspd > 0)
        {
            hspd -= fric;
        }
        else
        {
            hspd = 0;
        }
    }
}
```

(figure 5.1.7)

This script is used to check if the player is standing, if the player is pressing the duck key change the state to ducking, if not state is standing.

## 5.1.2 Objects

### *obj_localplayer*

Create Event: Initialise variables.

Alarm 0 Event: Resets jump height.

Step Event: Activates platforming scripts and for the multiplayer version will send the player’s coordinates and sprite information to the server.

Collision Event with *obj_platform*: Change player’s x position so they move with the platform.

Collision Event with *obj_platformV*: Change player’s y position so they move with the platform.

Collision Event with *obj_locked*: If the player’s key is the same colour as the door, destroy the instance. The multiplayer version will send the door colour to the server, so the door is gone from every client. If the key colour does not match set the x position to the previous x position.

Outside Room Event: If the player falls outside the room, set their x and y position to the starting x and y positions. If the player has reached a checkpoint set their x and y

positions to the checkpoints x and y positions. If the player has more than one life take away a life. If they have only one life left the user is sent to the main menu, in the multiplayer version the server would be alerted to the change. The life is also reset.
Press <Escape> Event: Send the user to the menu and reset their lives.

### *obj_block*

Solid object map is mainly made of this object.
Children: *obj_platform, obj_platformV, obj_platform_new, obj_platformSmall, obj_locked, obj_gate.*

### *obj_platform*

Create Event: Initialise variables.
Step Event: Increase the angle by radian and the x position by the cos of the angle.
Collision Event with *obj_localplayer*: The player's y position equals the platform's y position.
Draw Event: Draw self.

### *obj_platformV*

Create Event: Initialise variables.
Step Event: Increase the angle by radian and the y position by the sin of the angle.
Collision Event with *obj_localplayer*: The player's y position equals the platform's y position.
Draw Event: Draw self.

### *obj_platform_new and obj_platformSmall*

Create Event: Set the sprite index to minus one.
Step Event: If an instance of the local player object exists, if its y position and half its sprite height is less than the platform's y position or the user press the down key, change the mask index to minus one. This allows the player to fall through the platform. Else the mask index equals the platforms sprite.
Draw Event: Draw the platform.

### *obj_power*

Collision Event with *obj_localplayer*: Changes the player's jump height to their power up jump height and sets the *alarm[0]* to four hundred, about 4 seconds. Creates an instance of *obj_powerRespawn*, then destroys itself.

### *obj_checkpoint*

Create Event: Set image speed and image index to zero.
Step Event: Set the image angle to increase by one for each step. If the local player is colliding with the checkpoint, set the global checkpoint to this checkpoints id. Set the global checkpoint's x and y positions to this checkpoints coordinates and set the global checkpoint room to the room this checkpoint is in. If this checkpoint is active set the image index to one.

### *obj_key*

Step Event: If the local player collides with the key, it changes the player's *key_on* value to the key's colour, the key is then destroyed. In the multiplayer version, the key colour is sent to the server so that it will be destroyed for every player. Image speed is set to 0.35.
Draw Event: For the key object *draw_sprite_ext()* is used to draw the object, it draws the sprite at a position with custom scaling, rotation colour and alpha values. This is needed because the keys colour is set in the room.

### *obj_locked*

Draw Event: The locked object uses the *draw_sprite_ext()* function for the same reason as the key object.

### *obj_lever and obj_leveraftergate*

Create Event: Set image speed and image index to zero.
Step Event: If the local player is colliding with the lever, set the image speed to one and set gate open to true. If the image index equals sixty-one, set the image speed to zero. In the multiplayer version, the remote player can also collide with the lever, when neither are touching the lever, gate open is set to false. The server is also alerted to these changes.
Draw Event: Draw self.
There are three different lever objects and three lever after gate objects for the multiplayer. Each lever is attached to their gate e.g. lever one and gate one, and lever after gate one for the multiplayer.

### *obj_gate*

Create Event: Set the global gate open variable to false.

Step Event: If global gate open is true make the gate invisible and increase its y position by 500. If global gate open is false set visible to true and the y position equals the starting y position. In the HTML5 version when global gate open is true the instance is destroyed.

### *obj_stats*

Create Event: Initialise variables and set the size of the GUI layer.

Step Event: Add or subtract from health and make sure it doesn’t leave the range.

Draw Event: Override the draw event.

Draw GUI Event: If in the game room, set the offset. Using a for loop draw the grey hearts to match the max hp. Then draw the regular hearts over the grey hearts equal to the players current hp.

### *obj_ladder*

Collision Event with *obj_localplayer*: When the player collides with the ladder they are sent to the end room.

### *obj_powerRespawn*

Create Event: Set *alarm[0]* to five times the room speed, about five seconds.

Alarm 0 Event: Creates an instance of *obj_power* then destroys itself.

### *obj_timeController*

Create Event: Initialise global variables.

Step Event: If *global.count_up* is equal to true start adding one divided by the room speed to *global.seconds*. When *global.seconds* is less than sixty and greater than 59.9, set *global.seconds* to zero and add one to *global.minutes*. When *global.minutes* is equal to sixty add one to *global.hours.*

Draw Event: Override the draw event.

Draw GUI Event: Set the font type and colour. Draw the text using the global variables.

### *obj_score*

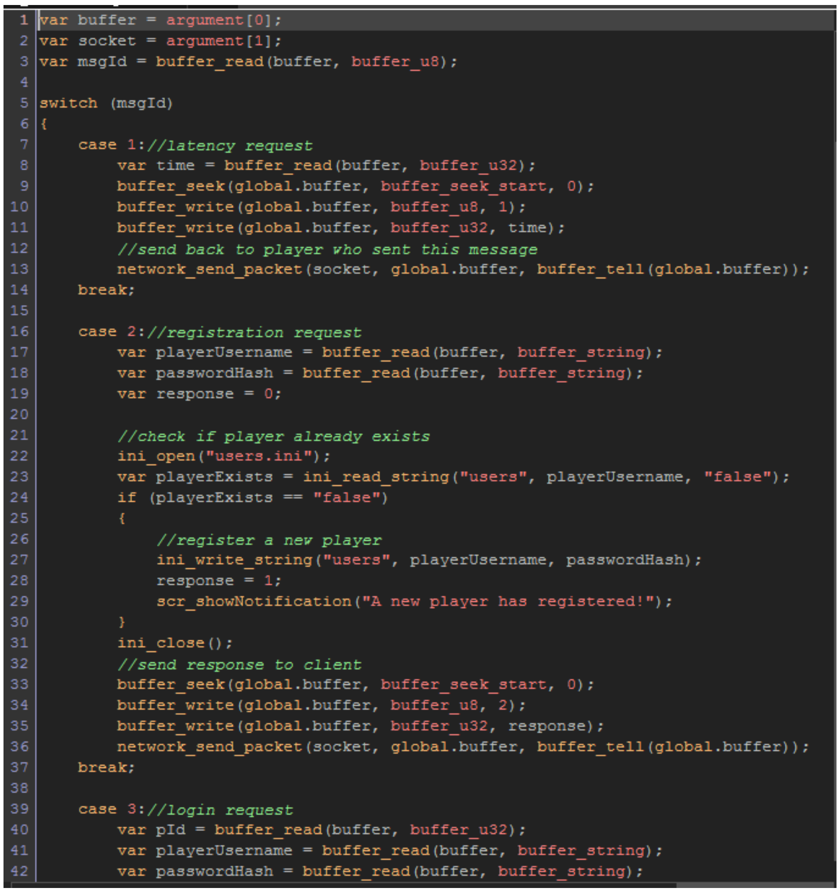
Draw Event: Override the draw event.

Draw GUI Event: Set the font type and colour. Display the score by using the global variables created in the *obj_timeController*.

## 5.2 Multiplayer Version

### 5.2.1 Server Scripts

#### *scr_handleIncomingPackets*



```
var buffer = argument[0];
var socket = argument[1];
var msgId = buffer_read(buffer, buffer_u8);

switch (msgId)
{
    case 1://latency request
        var time = buffer_read(buffer, buffer_u32);
        buffer_seek(global.buffer, buffer_seek_start, 0);
        buffer_write(global.buffer, buffer_u8, 1);
        buffer_write(global.buffer, buffer_u32, time);
        //send back to player who sent this message
        network_send_packet(socket, global.buffer, buffer_tell(global.buffer));
    break;

    case 2://registration request
        var playerUsername = buffer_read(buffer, buffer_string);
        var passwordHash = buffer_read(buffer, buffer_string);
        var response = 0;

        //check if player already exists
        ini_open("users.ini");
        var playerExists = ini_read_string("users", playerUsername, "false");
        if (playerExists == "false")
        {
            //register a new player
            ini_write_string("users", playerUsername, passwordHash);
            response = 1;
            scr_showNotification("A new player has registered!");
        }
        ini_close();
        //send response to client
        buffer_seek(global.buffer, buffer_seek_start, 0);
        buffer_write(global.buffer, buffer_u8, 2);
        buffer_write(global.buffer, buffer_u32, response);
        network_send_packet(socket, global.buffer, buffer_tell(global.buffer));
    break;

    case 3://login request
        var pId = buffer_read(buffer, buffer_u32);
        var playerUsername = buffer_read(buffer, buffer_string);
        var passwordHash = buffer_read(buffer, buffer_string);
```

(figure 5.2.1)

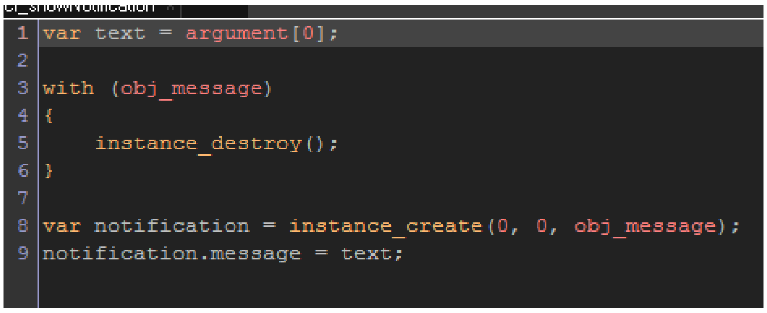
This script contains all the responses to the data being sent by the client. Each packet sent by the client starts with a message id which tells the server what to do. These ids are linked to different cases.

Latency request: The client sends a request every step which was set when the room was created. The client sends its current time to the server which reads it and sends it back to the client.

Registration request: The client sends a username and a hashed password, the server reads in the values and checks if the username has already been taken. If not the player is registered locally on the server side machine and the server displays a notification. A response is sent to the client telling them if registration was successful or not.

Login request:  The client sends a username, a hashed password and the played id. The server checks to see if the player exists and if the passwords match. Then the server sends a response to the client.
Player joining the game: When a player joins a game, the client sends the player id. The server gets the player's name and then sends both to the other players in the game world. The server then sends the other players names and ids to the player who had just joined.
Player movement: When a player moves, the client sends the player’s id, x position, y position, sprite number, image index and their direction. The server then sends all this information to the other players.
Door request: When a player collides with a door when they have the correct key, the server is sent the colour of the door which it then sends back to the clients to the more with this information.
Key request: When a player picks up a key, the server is notified of which key then tells the other players.
Gate requests: When a player uses a lever the corresponding gate opens and the server is told. It then sends back to the other players the gate is open. When the player lets go the server is alerted and then informs the others the gate is closed.

### *scr_showNotification*



```
var text = argument[0];

with (obj_message)
{
    instance_destroy();
}

var notification = instance_create(0, 0, obj_message);
notification.message = text;
```

(figure 5.2.2)

Destroys any remaining instances of the message object. Then creates an instance of the message object using the text it receives from the handle incoming packets script.

## 5.2.2 Server Objects

### *obj_message*

Create Event: Initialize variables.
Step Event: If alpha greater than zero lower alpha, else destroy instance.

Draw Event: Draw text using variables given.

#### *obj_controller*

Create Event: Initialise variables, create server and buffer using variables, create player list and counter. Use notification script to display “Server is up”.
Game End Event: Destroy server, buffer and player list.
Networking Event: When a client connects, counter goes up, id is assigned, server player instance created. Player is added to the list id is sent to the client. Use notification script to display “New player has connected”. When a client disconnects, player deleted from list other players notified, player instance is destroyed and notification script is used to display “Player has disconnected”. Other data received handle incoming packets script is used.
Draw GUI Event: Draws text of server status, total clients and all of each player’s information.

#### *obj_player*

Create Event: Initialize variables.

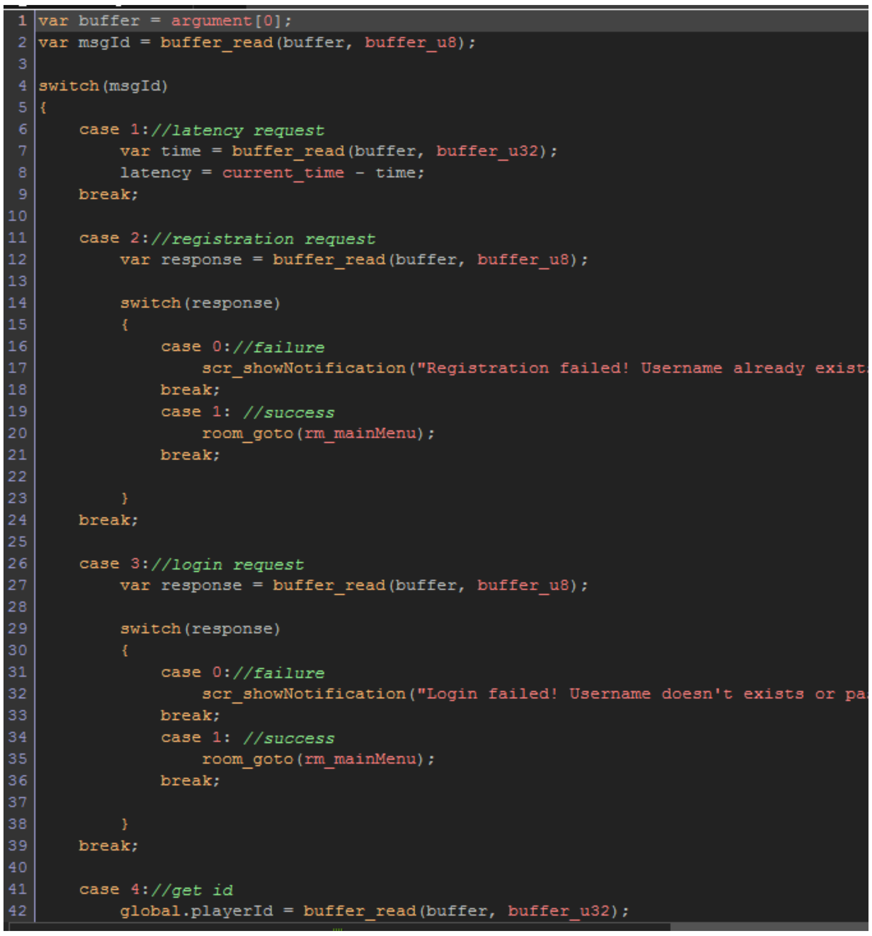
### 5.2.3 Server Rooms

#### *room0*

Blank room with *obj_controller* placed in it. Loads everything.

## 5.2.4 Client Scripts

### *scr_handleIncomingPackets*



```
var buffer = argument[0];
var msgId = buffer_read(buffer, buffer_u8);

switch(msgId)
{
    case 1://latency request
        var time = buffer_read(buffer, buffer_u32);
        latency = current_time - time;
    break;

    case 2://registration request
        var response = buffer_read(buffer, buffer_u8);

        switch(response)
        {
            case 0://failure
                scr_showNotification("Registration failed! Username already exist
            break;
            case 1: //success
                room_goto(rm_mainMenu);
            break;

        }
    break;

    case 3://login request
        var response = buffer_read(buffer, buffer_u8);

        switch(response)
        {
            case 0://failure
                scr_showNotification("Login failed! Username doesn't exists or pa
            break;
            case 1: //success
                room_goto(rm_mainMenu);
            break;

        }
    break;

    case 4://get id
        global.playerId = buffer_read(buffer, buffer_u32);
```

(figure 5.2.3)

This script contains all the responses to the data being sent by the server. Each packet sent by the server starts with a message id which tells the client what to do. These ids are linked to different cases.

Latency Request: Uses the time it received from the server, it subtracts it from the current time to get a latency value.

Registration Request: Uses the response from the server to either display a message using the notification script saying the registration failed or if registration is successful will go to the main menu.

Login Request: Uses the response from the server to either display a message using the notification script saying the login failed or if login is successful will go to the main menu.

Get Id: Reads the id the server sent and places it in a global variable then uses notification script to display “Our *playerId* has been received!”.

Other Players Ids: Gets the player ids from the server. Sets the remote player’s id.

Other Players Name and Ids: Receives ids and names from server, checks if other players exist and if the local player is in the game world create a remote player using the variables.

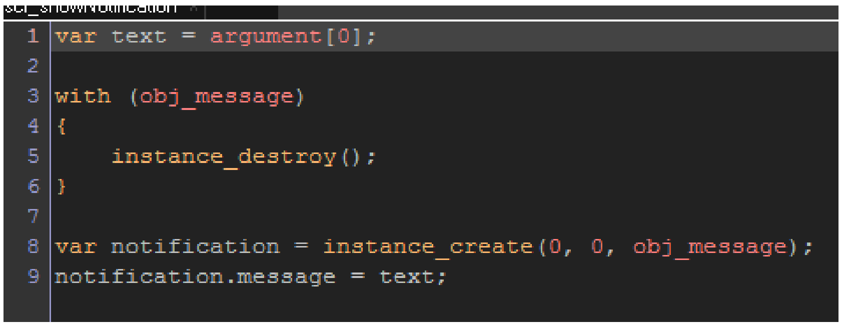
Other Players Positions: Receives player's id, x position, y position, sprite number, image index and direction. If remote player's id matches the sent id change their position, sprite, image index and their direction to what was sent.

Door Request: Receives door colour from the server deletes the matching door.

Key Request: Receives key colour from the server deletes the matching key.

Gate Requests: Receives either a one or a two from the server if one gate is open if two gate is closed.

### *scr_showNotification*



```
var text = argument[0];

with (obj_message)
{
    instance_destroy();
}

var notification = instance_create(0, 0, obj_message);
notification.message = text;
```

(figure 5.2.4)

Destroys any remaining instances of the message object. Then creates an instance of the message object using the text it receives from the handle incoming packets script.

## 5.2.5 Client Objects

### *obj_message*

Create Event: Initialize variables.

Step Event: If alpha greater than zero lower alpha, else destroy instance.

Draw Event: Draw text using variables given.

### *obj_controller*

Create Event: Create Socket, connect to server, create buffer and display "Client initialised" by using notification script.

Alarm 0 Event: Creates blink effect when typing.

Step Event: Update latency.

Game End Event: Destroy socket and delete buffer.

Networking Event: Sends data to handling incoming packets script.

Draw GUI Event: Draws latency in top left corner.

Press <any key> Event: Gets typed in text.

### *obj_remoteplayer*

Create Event: Initialize variables.

Draw Event: Draws player with their name and id.

### *obj_textbox_parent*

Create Event: Initialize variables.

Alarm 0 Event: Text blink.

Draw Event: Draw text box.

Press <any key> Event: Gets typed in text.

Children: *obj_textbox_username, obj_textbox_password1, obj_textbox_password2 and obj_ipEntry.*

### *obj_textbox_username*

Create Event: Event inherited.

Alarm 0 Event: Event inherited.

Mouse Left Pressed Event: If clicked select this object and deselect other objects.

Draw Event: Event inherited. If no text in box display "Enter Username", if there is text display what is typed.

Press <any key> Event: Event inherited.

### *obj_textbox_password1*

Create Event: Event inherited.

Alarm 0 Event: Event inherited.

Mouse Left Pressed Event: If clicked select this object and deselect other objects.

Draw Event: Event inherited. If no text in box display "Enter Password", if there is text display asterisks.

Press <any key> Event: Event inherited.

### *obj_textbox_password2*

Create Event: Event inherited.

Alarm 0 Event: Event inherited.

Mouse Left Pressed Event: If clicked select this object and deselect other objects.

Draw Event: Event inherited. If no text in box display “Confirm Password”, if there is text display asterisks.
Press <any key> Event: Event inherited.

#### *obj_ipEntry*

Create Event: Event inherited.
Alarm 0 Event: Event inherited.
Mouse Left Pressed Event: If clicked select this object.
Draw Event: Event inherited. If no text in box display “Enter ip”, if there is text display asterisks.
Press <any key> Event: Event inherited.

#### *obj_button_register*

Mouse Left Pressed Event: If not in *rm_register* go to that room. If username, password or confirm password is blank or passwords don’t match, use the notification script to alert player. If none are blank and passwords match send information to the server which will try to register the player.

#### *obj_btn_login*

Mouse Left Pressed Event: If not in *rm_login* go to that room. If username or password is blank, use the notification script to alert player. If none are blank send information to the server which will try to log the player in.

#### *obj_enter*

Mouse Left Pressed Event: If not in *rm_start* go to that room. If ip is blank, use the notification script to alert the player. If ip is entered set *global.ip* to the text entered in and go to *rm_boot*.

### 5.2.6 Client Rooms

#### *rm_start*

First thing the user will see, it contains the *obj_ipEntry* and a *obj_enter*. The user enters in the host ip and clicks on the button. They will then connect to the server if it is up. Goes to *rm_boot*.

#### *rm_boot*

Blank room with *obj_controller* placed in it. Loads everything. Goes to *rm_loginRegisterSelect*.

### *rm_loginRegisterSelect*

This room contains two buttons, an *obj_button_register* and an *obj_btn_login*. If the user clicks register they go to *rm_register* if they click on login they go to *rm_login*.

### *rm_register*

This is the registration room, it contains three textbox objects and a register button. The textboxes get the username, password and confirm password which get sent to the server when the register button is clicked. If registration successful go to *rm_mainMenu*.

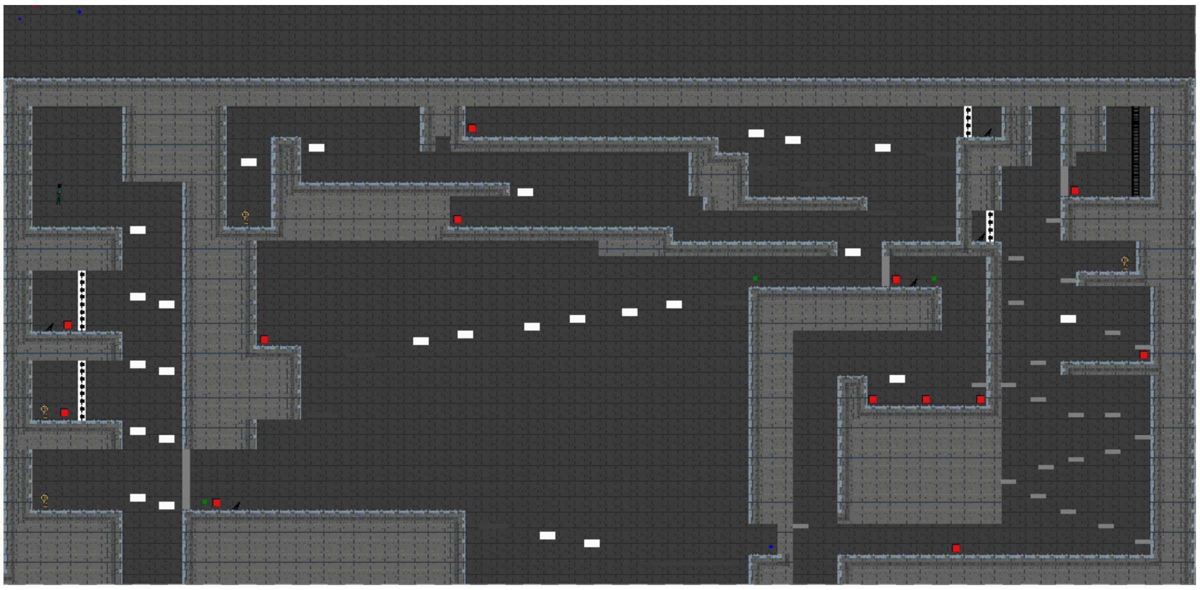
### *rm_login*

This is the login room, it contains two textbox objects and a login button. The textboxes get the username and password which get sent to the server when the login button is clicked. If login successful go to *rm_mainMenu*.

### *rm_mainMenu*

This is the main menu where a user can join a game or exit. If the user clicks the join button go to *rm_gamenew*. If they click the exit button close the program.

### *rm_gamenew*


(figure 5.2.5)

This is the game's level, it contains creation code to alert the server that a player has joined the game world. The only other differences are the platforms won't move unless both players are in the world and there are extra levers after each gate.

#### *rm_end*

A blank room that displays the player's score.

## 5.3 HTML5 Version

### 5.3.1 Objects

These objects are from gameSync which is an extension that is available for Game Maker Studio. It allows the connection of a MySQL Database by using PHP via Game Maker's asynchronous functions.

#### *objLoad*

Create Event: Using the player's username, activate the PHP script.
Alarm 0 Event: Use the string explode script to change the data from a string to an array. The values in the array are then assigned to global variables. There are now variables for the user's name, their score and the scores and names of the two players above and below the user on the leader board. Then destroy the instance.
Async Event HTTP: Pull the information out of the database. The information comes in the form of a comma delimited string. Set *alarm[0]* equal to five.

#### *objDrawData*

Create Event: Set *dt* equal to zero and *alarm[0]* to fifteen.
Alarm 0 Event: Set *dt* equal to one.
Draw Event: If *dt* is equal to one, draw the data taken from the database.

#### *objLoginButton*

Create Event: Initialise variables.
Alarm 0 Event: This event reacts to the return of the login request. If login was successful, the user is sent to the main menu. If not a message is displayed stating the username or password is incorrect.
Mouse Left Released Event: Display the login box.
Mouse Enter Event: Highlight the button during mouse over.
Mouse Leave Event: Remove highlight the button on mouse out.
Async Event HTTP: This gets the return from the login box request and assigns it to a variable
so, we can compare it. The alarm creates a slight delay to ensure we have it and are

ready to go before we try to use it.
Async Event Dialog: This takes the strings entered in to the login box, assigns them to a variable, and sends them to the database for validation.

### *objCreateAccount*

Create Event: Initialise variables.
Alarm 0 Event: This event reacts to the return of the create account request. If login was successful, the user is sent to the main menu. If not a message is displayed stating the username is already in use.
Mouse Left Released Event: Asks for the username they want to use.
Async Event HTTP: Gets and process the return for the username and passwords insert. If successful *alarm[0]* is set to ten.
Async Event Dialog: Gets the username, sets it to a global variable and asks for password. Gets the password, sets it to a global variable and asks for password again. Gets confirmed password, sets it to a global variable and compares the passwords. If the passwords match insert the username and password into the database, if they did not match display a message informing the user.

### *objRestart*

Create Event: Set image index and speed to zero.
Mouse Left Released Event: Restart the room.
Mouse Enter Event: Highlight the button during mouse over.
Mouse Leave Event: Remove highlight the button on mouse out.

### *objLogout*

Create Event: Set image index and speed to zero.
Mouse Left Released Event: Restart the game.
Mouse Enter Event: Highlight the button during mouse over.
Mouse Leave Event: Remove highlight the button on mouse out.

### *objSaveButton*

Create Event: Initialise variables and set image index and speed to zero.
Alarm 0 Event: If the save was successful display message game saved. If it failed show message there was a problem when saving.
Mouse Left Released Event: Activate the save function in the PHP using the players

name and score.
Async Event HTTP: Get the result of the PHP and store it as a variable. Set *alarm[0]* to ten.

### 5.3.2 Rooms

#### *rm_login*

This room is the first thing a user will see, it contains the *objLoginButton* and *objCreateAccount*. When the user clicks on a button a message box appears asking for the required data. After the user fills in the data they are sent to *rm_mainMenu*.

#### *rm_mainMenu*

This room is almost identical to the main menu of the multiplayer version the only difference is there is a button that sends the user to the leader board which is in *rm_main*.

#### *rm_gamenew*

The differences between this version of *rm_gamenew* and the multiplayer version, is there are no levers after the gates.

#### *rm_end*

This room shows the player’s score, has a save button so they can override their previous score and has the same button that is in *rm_mainMenu* that will send the player to *rm_main*.

#### *rm_main*

This room displays the leader board to the user. To display the information from the database the *objLoad* object is used to get the data and *objDrawData* displays it. There are also three buttons at the bottom of the room they are used to restart the room, restart the game and to save data to the database.

## 5.4 PHP

The PHP script was also made by gameSync but it has been adapted slightly so it would work with this game. The script is separated into twelve sections. The first section is error reporting, at the moment this is commented out it was only used when trying to get the database connection set up. The second section is where the

MySQL connection variables are declared, the hostname, the MySQL username and password, the name of the database and the name of the table where the data will be stored. The third section creates the connection to the database. The fourth section is the variables that will be passed from the URL. The fifth section encrypts the user’s password. The sixth section is the start of the functions. The seventh section is the function for creating an account, it inserts the player’s name, password and zero for their score. The eight section is a function which saves information to an existing account, it will update the score that matches the player’s name. The ninth section is a function that pulls out information from the database that is used for the leader board, the current player, the two players just above them and the two that are just below them. The tenth section is a function that checks if a user exists, it is used when the user is logging in. The eleventh section is a function that checks if the script and database are working. The twelfth section is a switch statement that which determines which function to call based on the parameters in the URL.

# 6.Testing and Results

## 6.1 Explanation:

This document explains the different activities completed as part of testing the game Dave. Dave is a game that is supported by HTML5 browsers. The user creates an account in order to play the game and see how he/she compares in scores with other players that have also created accounts. The game has two versions the HTML5 version and a standalone LAN multiplayer version that differs slightly from the HTML5 one. If the two players are on the same network one of them must run an instance of the server while both run the exe client that runs the main game, both players connect to the one server on the one machine running it.

## 6.2 Original Test Plan:

Testing will be done on both versions of the game. The first part that is going to be tested is the networking. The server is going to be stress tested, this will be done by setting up a server object and connecting multiple instances of the client object. More and more instances will be created to see how long the server will last with the addition of new users. After this, the server will be restarted and each user will have their own client object and will attempt to register, if they are successful they will re-run the application and try to login. The next part will be the basic gaming controls: can the player move their character, can he jump, how do the animations look? Each animation is going to have to be checked for each movement and for every other action (death/crouching). The game physics will then be tested, see if the player will go through walls, check if checkpoints work, see if the enemies will kill the player etc.

## 6.3 Testing:

The multiplayer version was tested first. At the point of making the test there was no decision on how many players will be allowed in the game, now the decision is there will only be two players allowed on the server at one time if a third person tries to connect the connected player’s game will crash. The server runs the game smoothly with no noticeable loss of packets. The next thing to be tested was the registration, the user enters in a username, a password and confirms the password. The menu will alert the user if they have not entered information into a field, the passwords do not match or the username is already taken. The usernames and hashed passwords are saved on the computer that is running the server. The login menu, was tested after the users entered their usernames and passwords. The login menu also notifies users if fields are blank or don’t match what is saved on the server. The user then entered the level, when only one user is in the level the platforms don’t move. When the second user enters the room the platforms start to move. This rule was added since players play on two different instances of the game and dude to this the platforms will run at different times due to difference in server join times. The user is able to see the other user move throughout the level. The player sprites change accordingly to the keys being pushed by either user. The player masks are still slightly off from perfect. Users can pick up a key, keys disappear for player two when picked up by player one and vice versa, as it is supposed to, the player who has key can open the door, door is open for player two as it is supposed to. User kills enemy, still there for player two, supposed to happen. When a user touches a lever the gate opens for both players, player two goes to second lever gate is open for player one and so on. User grabs power up, still there for player two, this is supposed to happen. If users die after hitting checkpoint, they respawn at the last checkpoint not at beginning. For the HTML5 version, the game is very similar, the biggest difference in game was float values working differently in HTML5 then in the executable file for the multiplayer. This affected the one-way platforms and the moving platforms. The other difference was the menus being connected to a MySQL database by using PHP, which also a Leader-board that gets all player scores. There could be more collision bugs because a lot of the sprites are placeholders just made so the functionality could be done. After the new sprites, another quick test will be done to make sure everything is still working.

## 6.4 Results:

### 6.4.1 The One-Way Platform Bug:

The problem with one way platforms is that in the HTML5 version of the game the player will fall through it. This is because of float values behaving differently and the code *mask_index* = -1; not working in HTML5. This wasn’t a big problem because at this stage the platforms no longer needed to be one way so they were just changed to solid platforms. The Locked Door Bug The problem with the locked doors is when the player jumps into a door when he does not have the required key his y value will not change, he will still be floating when attached to the door. The reason for this is when the doors were being coded they were for a top down game. Looking at the code there was an easy solution, all that needed to be done was comment out a line of code that said, y = *yprevious*. Without this line, when the player jumps into a door without the required key he will fall to the floor as he is supposed to.

### 6.4.2 The Enemy Collisions Bug:

The problem with the collisions was that enemies could go through the locked doors and the gates. If a player stood at the side of a door the enemy would notice him go through the door and kill the player. The solution to this problem was to make the door and gate objects children of the block object, which is what the walls and floors are made of, they already have the suitable collision code for the enemy.

### 6.4.3 The Power-Up Bug:

The problem with the power up is that they were not respawning so if player got the power up and wasn't able to get where he was supposed to go before the power up ran out he couldn't get any farther in the game. The solution to this problem was to move the re-spawn code from the power up into to its own object and place the new object in the room that will create a new instance of the power up after five seconds.

### 6.4.4 The Multiplayer Death Bug:

This problem occurred when two players were in a game and one of the players lost all their lives, they were both pushed back to the menu but the server wasn't told about this change, therefore the other player's screen (the player who had died) was still in-game. Although everything seems fine for this player, the platforms are still moving, when the other player tries to join back in they can longer see the player and the platforms stop moving. This problem was fixed after it was discovered because there already was code that did what was needed. When the player pushes escape the leave the level, the server gets told of this change. The same code is able to be used for when a player dies, either by enemy or falling out of the map. Now when a player dies the server is told and the platforms stop until the player re-joins.

### 6.4.5 The Collision making:

This error occurred due to a less accurate masking of sprites assigned to the main character object. The less accurate one has to be used since it is a lot faster and requires less system resource. This can be fixed by making the mask a bit smaller and adding some code that limits the sprites proximity to the wall, this code acts as faux collision and is a clever way to overcome complications and sprite glitches. Some small errors still occur during the sprite change cycle while in the falling animation and sometimes while standing on the very edge of platforms. This will be fixed in the future but for now it remains an issue.

### 6.4.6 The Sprite Resize Issues:

Sprite resize bug, this was not really a bug but was still an issue that was encountered while changing the main game sprites to some more hi-res ones. The issue was that while resizing the game would output the resized sprite height and the original sprite width which made things look distorted, this issue was fixed by using the resize tool in the actual Game Maker app rather than Photoshop.

# 7.Discussion and Conclusion

The aim for the project was to gain a better understanding of the software development process, with a focus on game development. To be able to start with a plan and be able to follow it and change it when necessary to achieve the project goals.

The idea at the start was that one person would be the main programmer and the other would be the main designer. There was a lot of different ideas thrown around when brainstorming on what was going to be in the game and what is was going to be about. The original idea was to have multiple levels but our project supervisor advised us to focus on making one great level than a few decent levels because adding more levels wouldn't give anything extra to the project.

Originally there was only supposed to be one version of the game but after finding about game maker's networking code not being compatible with HTML5 the decision was made to make an executable version for the local multiplayer and an alternate version for HTML5 that is able to connect to the database. If the research had been more thorough this issue would have been found much sooner and a more favourable solution could have been found.

## 8.References

http://ekalia.com/gm/gmsdb/doc/
http://www.destronmedia.com/ngm/pages/gamesync/gamesynchs.php
http://help.yoyogames.com/hc/en-us/articles/216754698-Networking-Overview
http://www.yoyogames.com/blog/6
https://docs.yoyogames.com/source/dadiospice/002_reference/networking/index.html
https://docs.yoyogames.com/source/dadiospice/002_reference/buffers/index.html
https://docs.yoyogames.com/source/dadiospice/002_reference/buffers/using%20buffers.html
https://docs.oracle.com/javase/tutorial/networking/sockets/definition.html
http://www.tutorialspoint.com/java/java_networking.htm
http://www.iusmentis.com/technology/tcpip/networks/
https://csanyk.com/2012/09/position-and-motion-tutorial-gamemaker/
http://help.yoyogames.com/hc/en-us/articles/216754018-HTML5-Issues-And-Differences
(https://crew.co/how-to-build-an-online-business/build-ios-app-or-android-app/)
Jonathan Hassell (http://www.computerworld.com/article/2518482/enterprise-applications/developing-for-the-iphone-and-android--the-pros-and-cons.html)
John Andrews (http://www.zco.com/blog/html5-games/)
Michael VK (http://ezinearticles.com/?Advantages-to-Android-Game-Development&id=6620773)
(http://gamedev.stackexchange.com/questions/859/what-are-the-advantages-and-disadvantages-to-using-a-game-engine)
Heba Soffar (http://www.online-sciences.com/technology/the-advantages-and-disadvantages-of-android-mobile-phones-2/)
Ellen Wills (http://www.selfgrowth.com/articles/android-game-development-advantages-not-to-be-ignored)
(http://videogamedesign24.com/game-maker-studio/)