# Game Summary

## Objectives

Smashville is a 1 – 4 player 2D physics based fighting game, where the player’s objective is to try and knock players off the stage and out the of the arena.

## Rules

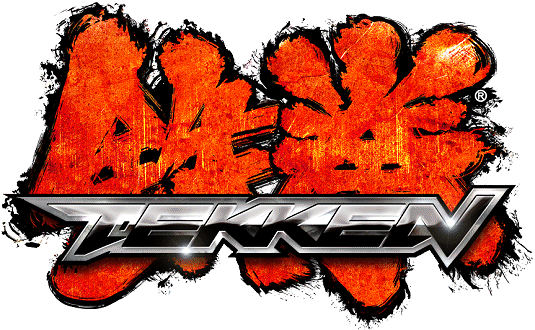
There are a set of rules in Smashville:

* Players can move left, right and can double jump
* Players can be hit by other players left, right, up or down
* Getting hit will increase the knockback of the players
* Getting knocked off the screen will result in the player losing a life
* Once a player has been knocked off 3 time by the opposing player they have lost

## Game Play

Smashville is a medium paced fighting game, that makes use of the physics engine to provide an unusual spin on traditional fighting games. None of the players have health bars, instead each player has a base percentage that gradually builds up the more a player is hit, which influences the velocity of the knockback caused by other player’s attacks. This adds an element of strategy to the game, as planning your plan of attack against the enemy player and the timing are paramount to your survival.

# Background Research



Early in the design phase of the assignment there was a lot of research surrounding traditional fighting games. More specifically games such as Street Fighter and Tekken were potential inspirations for the game, both franchises have been very prominent in the Arcade fighting community, making heavy use of combo’s and chaining abilities to overwhelm your opponent. Although the combo system in these games is interesting, it can be rather complicated to develop, and fighting games are traditionally developed with a controller in mind, and not a keyboard. So, a simpler approach to the genre was needed, one that would translate well to the keyboard.



The game drew heavy inspiration from the Super Smash Brother’s series on the Nintendo consoles, which is a physics based fighting game. What is great about the Smash Brothers format is that it is remarkably easy to replicate on a PC. This is because combo systems and complex button mapping do not play a prominent role in the game itself. Smash Brothers is a game that can be picked up by anyone of any age and not require them to know every combo combination and the best way to chain them, which was a very attractive aspect of its system.

# Screen/Level Map

#Will work on this when back in Brighton, don’t have NODE JS here

# Network Utilisation

## Network Architecture:

**Server**

Receive Packet

|

Calculate Logic

|

Send Packet to player(s)

**Client**

Send Packet

Receive Packet

|

Apply Packet Logic

The diagram shows a high level approach to how the client and server communicate. The lines indicate the direction of the packets (Who shall be sending and who shall be receiving them).

## Component Interactions:

There are several kinds of messages which are exchanged between the server and the client. This is because we have several client states which are looking for different kinds of packets. This section will cover these different states.

The first state is the menu state, where the client is looking for information regarding chat messages and session statuses. The client will be awaiting for the server to send information when a new session has been created, when a session status has updated or when another user has sent a message. At this stage the user will send packets when they have entered a message, joined or hosted a server.

Next we have the session state where clients have joined a game session with other players, or hosted one that players can join. The client will be looking for packets regarding the other connected players character selections or waiting for the sessions host to begin the game. The client will send a message when they’ve selected a character, left the session or (If they’re the host) begun the session.

Finally we have the in-game state, where the host has begun the session and the players are actually playing the game. Clients will be looking for messages when other players have moved or been hit. The user will also be sending the same kinds of messages that they are expected to receive.

There are also other handling packets for when a client has disconnected from the session or passing players details when a new client has connected.

# Client

## Objects

There is a total of 4 objects on the client side of the application.

### Player.js

The player is responsible for all interaction involving the player, this includes: movement, attacking, checking for hits, checking for ring outs, and dying. The player object also keeps track of its lives (also known as stock), its X and Y position and its sprite dimensions. It also has its own internal update function to update its state

### Enemy.js

The enemy object is very simple, as most of the logic surrounding its interactions with the other players is handled by the server. On the client side of the application all we keep track of is: its X and Y, its sprite dimensions, its characterID which determines its sprite, its name, and its lobbyID, which determines its position in the lobby. We also handle removing the sprite on the client side.

### Session.js

The session object is made up of 4 property’s: its ID which is used by the server to determine which server it is, its body content which is rendered by the client, its playerCount for the number of players present in the lobby, and its state.

### Sound.js

The sound object handles the music that plays throughout the application. It can play, stop, or loop the music, it can queue up the next song, and it can adjust the volume of the music. It also contains its own update function.

## Server

## Data Structures

The server holds several objects for managing the different players and sessions. This section will go over each of the objects and explain their purpose.

First we have the Client object which, as the name suggests, holds details regarding each individual client. These details include their id, name, x and y positions and hit percentage. The server will use these details to distinguish which client their communicating with and manage their overall gameplay experience.

Next we have the Session object which holds information about individual sessions, such as their name, state, selected level and an array of all the connected players. The server needs these details to manage the overall session state and cut down the time required to find players who are part of that session.

The Message object is, as the name suggests, designed to hold messages which have been exchanged in the “chat” screen. It simply holds the name of the player who sent the messaged and the body of the text.

Next the Logic object holds several functions which calculate in-game events, such as checking collision, registering hit damage and calculating knockback effect. This object was designed to give the developers the ability to tweak how effective certain actions are or add additional logic later on into the game.

Finally we have the Debug object which runs several small tests on the Logic object (Although more could be added in the future) that get run whenever a client starts the server. This was designed as a mini Unit Testing class so developers could easily expand or alter how testing the server works.

## Code Structure

*Document the overall flow of the code and where it is located within the source file(s). Provide a table with one row for each of the functions that you have created within your server.*

<https://esdoc.org/>

Paste directory link to live Github page:

<https://malithium.github.io/Smashville/doc/index.html>

# Implementation Evaluation

*Provide a summary of the strengths and weaknesses of the technologies (platforms/frameworks etc) that you used to implement your game.*

## Strengths

### Easy communication between Phaser and Socket.IO

Due to the both technology’s being JavaScript based, having them communicate and work with each other is very straightforward. Simply Initializing a socket object in Phaser with an IP and a port allows the application to connect, and then that socket can be referenced anywhere to send packets. It’s a very straightforward and comprehensible system.

## Weaknesses

### Partially Peer-to-Peer

One major weakness of Smashville is that it’s partially Peer-to-Peer. This is because the program couldn’t run Phaser HEADLESS on the server side due to Node.js executing socket.io from the terminal (Not from a browser). There are better ways that this could be handled, such as using an engine rather than the Phaser library (Unity for example). The server could also have additional checks to prevent players “hacking” or passing across “broken” values. Due to the nature of these issues though there would need to be a lot of checks required to counter this.

### Lack of built in textbox support in Phaser

Phaser does not have any built-in support for textbox’s, this meant that DOM JS had to be implemented to make up for this. This has resulted in an odd combination of Phaser and DOM JS working in tangent between states which has complicated the code base and made it harder to manage, also getting the HTML elements to scale and function within the Phaser canvas and between states is difficult, scaling the screen distorts some of the UI due to this.

Strength:

Weakness:

# References

<https://upload.wikimedia.org/wikipedia/en/e/e1/Tekken_series_logo_as_of_2012.gif>

<https://upload.wikimedia.org/wikipedia/en/e/e9/Street_Fighter_Logo.png>

<https://upload.wikimedia.org/wikipedia/en/a/af/Super_Smash_Bros_4_merged_logo%2C_no_subtitle.png>

Joshua Petherick Review:

Concept Presentation 50%:

Josh created and styled the presentation, and adjusted it according to my input, he was also the one who suggested Super Smash Brothers as potential inspiration. I did much of the background research but Josh provided a lot of direction here. As for presenting the actual presentation we both practiced and discussed what we would be talking about and the best way to portray our idea. We both did an equal amount of work here.

Implementation:

I will break this down into 2 parts because I feel it’s important for clarity

Server Implementation 55%:

Josh worked a lot more on the server side of the application than I did, he was rather confident in the knowledge he had built up on this topic due to his final year project, so I felt it was in capable hands. My input on the server side was mostly regarding how it communicated with the client, I worked a lot more on the client side than Josh Implementing every menu, so I knew best how it was supposed to communicate with the server and what data it needed. However, Josh made the large majority of the decisions regarding the server’s behaviour, and I don’t intend to take credit for that, the extra 5% above the 50% is because of this.

Client Implementation 45%:

In contrast to Josh’s server work, he did not have as much influence on the client side of things as I did. Josh and I did both create the design for the menus and how they should flow, but the actual implementation was left largely up to me. This was not an issue though, me and Josh kept ample communication with each other and he made up for his shortcomings on the client side of things by working more on the server side of things Josh did implement much of the code in the play state, including movement, double jumps and attacking, he also implemented the music system in the game.

Overall I feel the fact we worked more on the various aspects of the application was unavoidable, I had the necessary skillset to work on the more complicated aspects of the client, such as the interactions between the DOM JS and Phaser and state transitions. Whereas Josh had the better skillset for the server logic, I feel tackling the assignment in this manner helped us in the long run, as we sped up production by making up for each other’s short comings.

Report 50%:

The fact there are aspects of the report Josh explains better than I do and vice versa shows that we both had equal input on this report. As Josh understands the server architecture better.