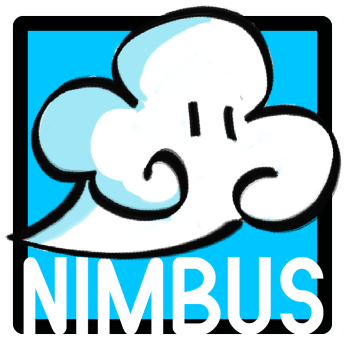
Nimbus Run



An Epic Race

By Lemon Inc (group 12):

Kevin Tae Kim (1001180)
Lim Zhenyang (1001006)
Nikki Tibrewala (1000943)
Chng Wei Sheng (1000978)

Table of Contents

Introduction

“When life give you lemons, make lemonade.” was what inspired us to name our group Lemon Inc. We wanted to carry this mindset into creating our project, which we knew was going to be fraught with many challenges.

When we first came together to brainstorm for ideas, we wanted to create something simple, yet fun to play. One of the games that we aspire to surpass, is the japanese arcade game named “Bishi Bashi”. It is made up of a series of games where multiple players compete with each on a series of mini games on a real-time basis.

However, we realised that implementing the game mechanic for so many games was time-consuming. The mini game ideas that we came up with were pretty impressive and relatable to among our peers. Eventually, we settled on one out of the many “out of the box” games that we came up with.

This final product is none other than Nimbus Run. It’s a Fun Run-inspired racing game with our Asian mythological and legendary twist to it. It comes everything epic – epic backgrounds, epic gameplay, epic sounds and of course epic characters that are based on cloud. Hence the word Nimbus.

We hope you will enjoy the game as much as we did designing it.

System Requirements

We identify our system requirements as a real-time, multiplayer online game on Android platform. As such, certain functional and nonfunctional requirements are set:

Functional:

1. An Android-based game that is multiplayer and not turn-based
2. A networking system to allow players to connect and play together real-time

Nonfunctional:

1. Game must be fun and easy to pick up for players
2. Replayability must be high (since we are not producing a lot of content)
3. Game should have minimal lag-time (since it is real-time)
4. Game should be easy to start playing (not much loading; setup required)

As such, we brainstormed and came up with a few features that we should implement:

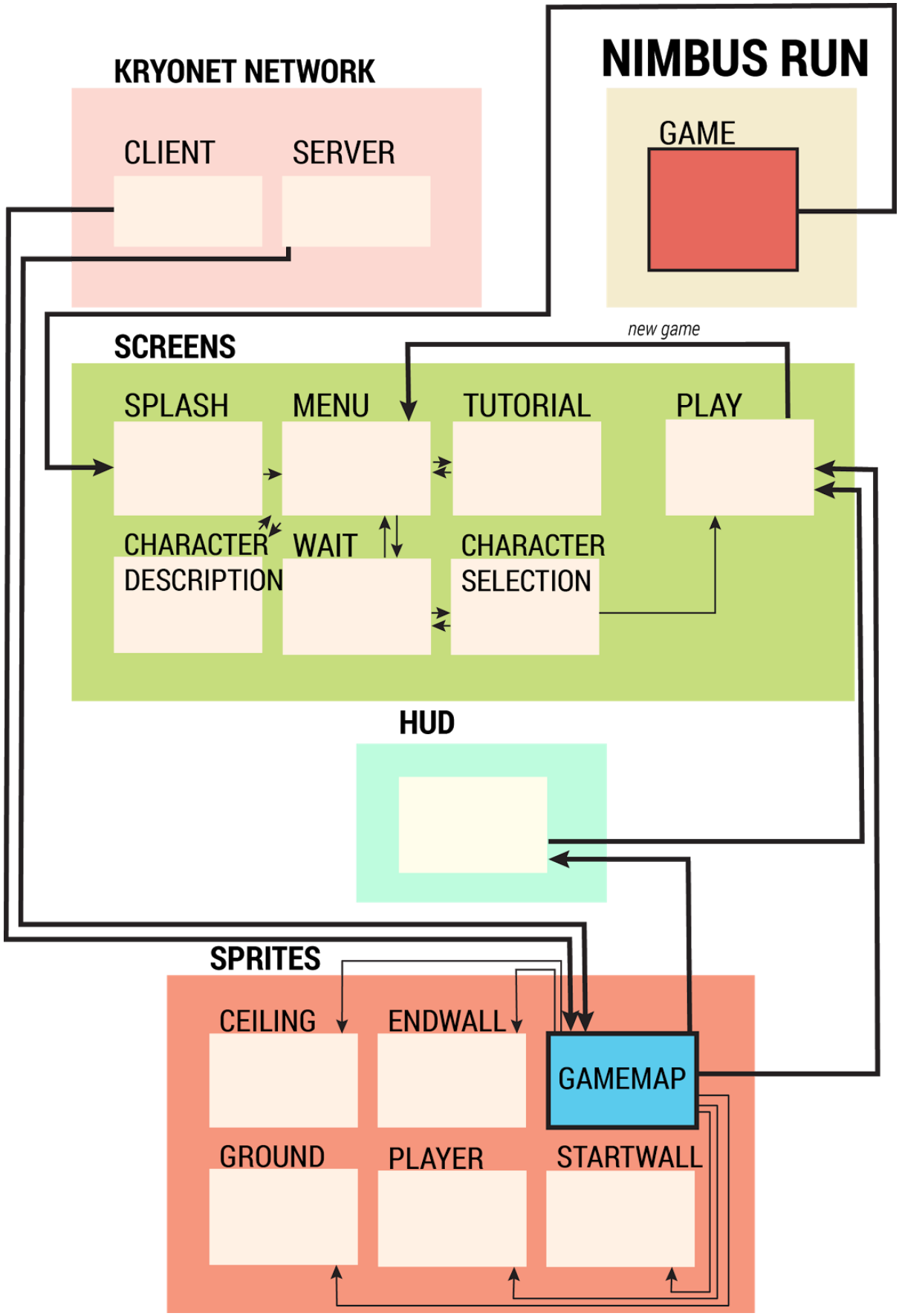
1. A competitive game (high replayability; more engaging)
2. As few menus/screens as possible (easy to start playing)
3. Concurrency management systems to minimize lag time in gameplay

System Design

For our framework, we chose the LibGDX library due to our familiarity with Java and because it had more relevance to what we are learning in our module. After doing some tutorials to introduce ourselves to what LibGDX was capable of, we decided to use the Game and Screen interfaces offered in the library to be our game backbone because of its ease of use and its in-built methods suited our needs.

As for the networking system, we decided to use KryoNet instead of Google Play Services for two reasons. First, we aimed to deliver a seamless gaming experience and it was imperative that our game UI was aligned to a single theme. Google Play's android theme was unsuitable for our UI and hence, KryoNet was chosen for our networking library. Second, we wanted to make the most out of the learning experience the project could offer. Instead of relying on Google Play API, we chose to use KryoNet to learn the lower level implementation of networking and also to apply the Networking knowledge gained from our Computer Systems Engineering and Elements of Software Construction courses.

To start off, the diagram below will give an overview of Nimbus Run's system design. The four main categories – Network, Sprites, HUD and Screens – are organised based on Android and libGDX's design. Sprites are primarily used to generate content on screens, and screens are in turn used to organise these content. For more information on the flow from each class to the next, as well as their interaction within the game, refer to the UML diagrams in *Use Cases*.


NIMBUS RUN
KRYONET NETWORK
CLIENT
SERVER
GAME
new game
SCREENS
SPLASH
MENU
TUTORIAL
PLAY
CHARACTER DESCRIPTION
WAIT
CHARACTER SELECTION
HUD
SPRITES
CEILING
ENDWALL
GAMEMAP
GROUND
PLAYER
STARTWALL

Network Design

In every game session, there is one host and a maximum of three clients. One can choose to either play as a host or join an existing game at the Waiting Screen, which is displayed in our User Manual. Once, a player decides to play as a host, both the server and client logic is instantiated in his game at the Character Selection Screen. If not, only the client logic is instantiated in the player's game.

At the Character Selection Screen, a random map is generated only if the player is a host. When a new client establishes connection with the server, the client sends a Login packet to the server that contains the client's username. Once the server validates the username, the server numbers the client for its own internal use and sends back to the client about the initial position coordinates of the client when the game starts and the map data that the host has generated. The server also updates the new client about any existing clients' initial coordinates and vice versa.
in order for the game to start, or transit to the Play Screen, all clients must have selected their characters and pressed the "JOIN GAME" button. This sends a packet from the client to the server that indicates which character the client has selected, and that he is ready to play the game. The host can start the game only after all clients have selected their characters.

At the Play Screen, every client, including the host, renders the game world according to the data received from the server e.g. map data, player coordinates, player character. During the gameplay, clients send the movement states of their players to the server via a MovementState packet that contains the client's Player coordinates and linear velocity. The server applies the movement states to its internal game map and broadcasts the packets to the other clients. To reduce latency, packets are sent from the client to to the server's UDP port. Unfortunately, due to a Kryonet/Android bug, the server could not broadcast the movement states to the clients via UDP, but instead TCP.

When a player disconnects, the client sends a PlayerJoinLeave packet that contains a boolean variable for hasJoined that is set false. When the client receives the packet, it destroys the disconnected client's Player from the box2d world and broadcasts the disconnection to the other clients so that the Player can be destroyed accordingly.

For more clarity, please refer to the Network section of the Class Diagram in the Appendix section.

Installation Manual

Our game can be found on Google Play at this link:
https://play.google.com/store/apps/details?id=com.lemoninc.nimbusrun

Alternative, you may pull our repo from GitHub and run it in Android Studio (v1.4):
https://github.com/zhenyangg/LemonInc

User Manual

The three themes of the game are myth, legend and clouds. Characters in the Asian world are chosen from various Indian, Japanese, Korea, Malay and Chinese legends and given a cloudy touch.

The character's abilities are detailed here. Take full advantage of each character's ability by reading up on their descriptions, and get ready to be the race champion of Nimbus Run!

laughing buddha

ABILITY NAME: infectious laughter

ABILITY DESCRIPTION: BRING EVERYONE TO THE GROUND LAUGHING NON-STOP WITH HIS DISTINCT, SIGNATURE GUFFAWING

nine-tailed fox

ABILITY NAME: charm

ABILITY DESCRIPTION: USED TO SEDUCE YOUNG MEN FOR A QUICK MEAL, THE KUMIHO'S CHARM CAN BE USED TO BAFFLE GODS TEMPORARILY

kappa

shifting tides

ABILITY NAME:

ABILITY DESCRIPTION: TURN THE TIDE WITH POWERFUL CURRENTS THAT WILL REVERSE THE FLOW OF CLOUDS AND WATER ALIKE.

krishna n sheshnag

Light the World

ABILITY NAME:

ABILITY DESCRIPTION: LIGHTEN UP THE WORLD WITH KRISHNA'S POSITIVE VIBES!

pontianak

wail of terror

ABILITY NAME:

ABILITY DESCRIPTION: SEND SPIRITS AND MEN ALIKE FLEEING WITH AN EERIE CRY OF SADNESS

madame white snake

Debilitating venom

ABILITY NAME:

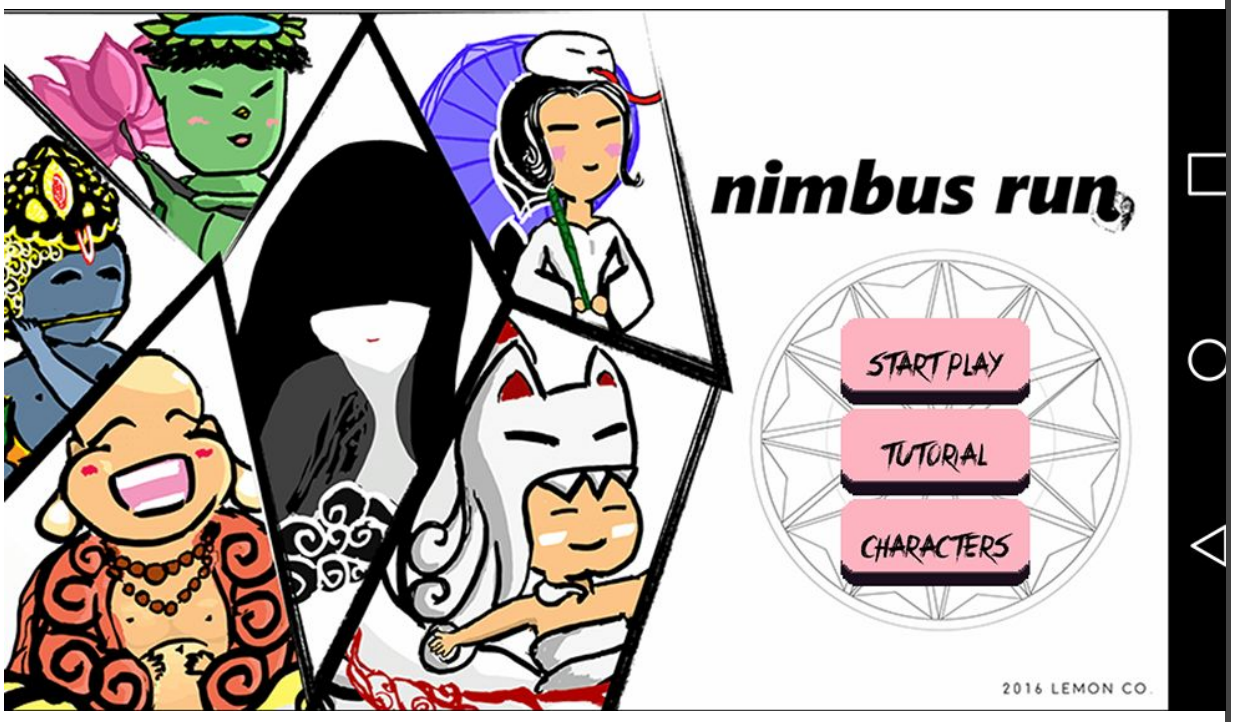
ABILITY DESCRIPTION: WHILE DEADLY TO MORTALS, MADAME'S VENOM WILL CAUSE A NASTY STOMACH FOR SPIRITS AND DEITIES.

In this section, a series of screenshots will be used to guide you through the steps to navigate Nimbus Run. They are arranged such that you can go to any section corresponding to the screen you are on – there's no need to read from the beginning to the end.

When the game starts, the orientation of the phone will automatically change to landscape. Tilt your phone to the correct orientation and let's start playing!

[1.0] START SCREEN | [1.1] WAITING SCREEN | [1.2] CHARACTER SELECTION SCREEN | [1.3] PLAYING SCREEN
[2] TUTORIAL SCREEN | [3] CHARACTER DESCRIPTION SCREEN

This is the start screen. There are a total of three buttons.
Touching the buttons will bring you to:
[Start Play] > Waiting Screen
[Tutorial] > Tutorial Screen
[Characters] > Character Description Screen



[1.0] START SCREEN | [1.1] WAITING SCREEN | [1.2] CHARACTER SELECTION SCREEN | [1.3] PLAYING SCREEN
[2] TUTORIAL SCREEN | [3] CHARACTER DESCRIPTION SCREEN

This is the waiting screen. You can either play as a host or as a client. Please take note of these rules. This game supports up to 4 players, including host and clients. Only one player can join as a host. The rest of the players should join as clients

[1.0] START SCREEN | [1.1] WAITING SCREEN | [1.2] CHARACTER SELECTION SCREEN | [1.3] PLAYING SCREEN
[2] TUTORIAL SCREEN | [3] CHARACTER DESCRIPTION SCREEN

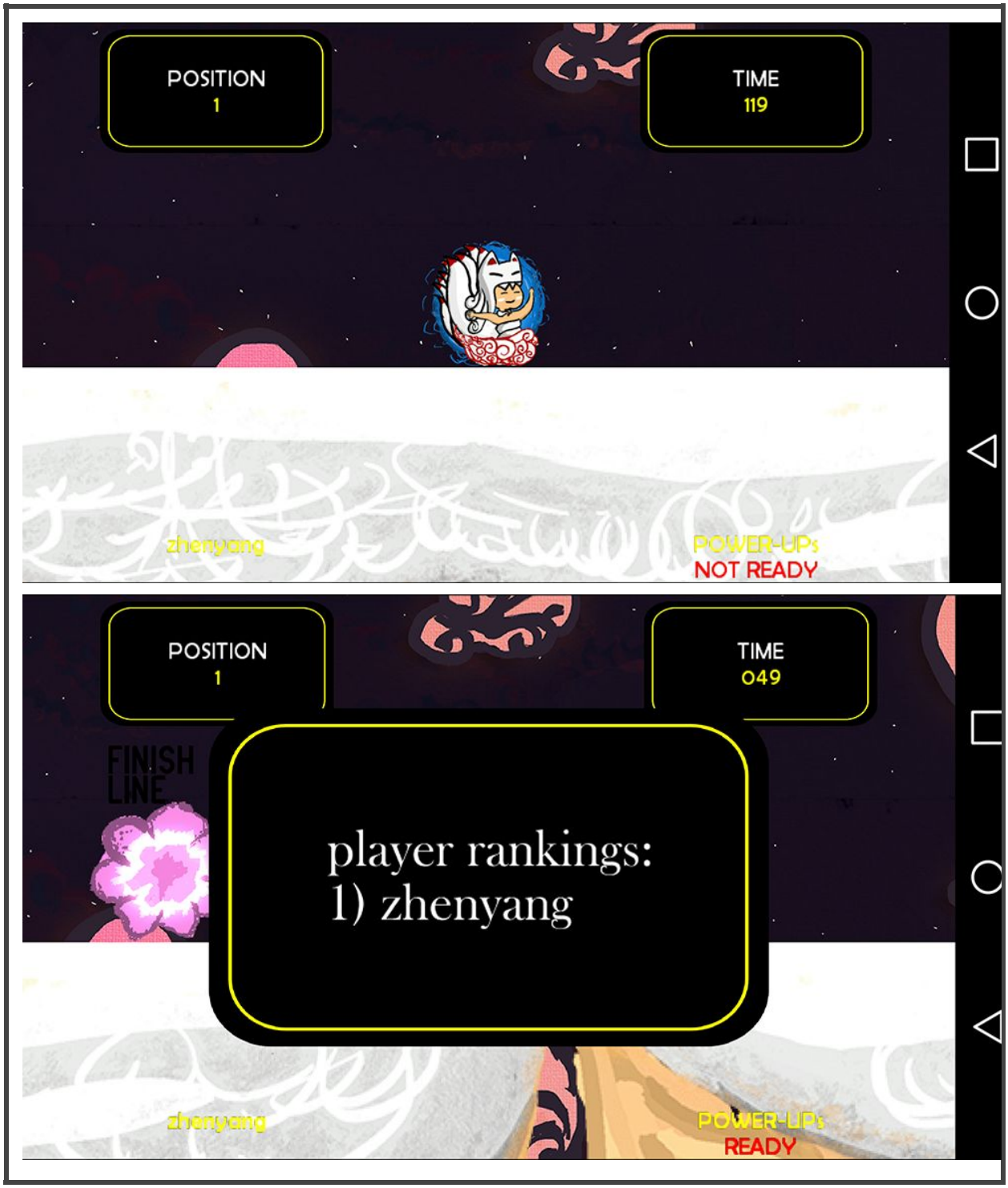
This is the character selection screen. After joining as a host or client, you can choose your character using the buttons on the left side of this screen. Once you've chosen your character, the character selected will be displayed on the right side of the screen. Clients should press *Join Game* first, followed by the Host.

[1.0] START SCREEN | [1.1] WAITING SCREEN | [1.2] CHARACTER SELECTION SCREEN | [1.3] PLAYING SCREEN
[2] TUTORIAL SCREEN | [3] CHARACTER DESCRIPTION SCREEN

This screen starts with a countdown timer of 4. During these 4 seconds, no player is allowed to move. After the countdown, the timer on the top-right of the screen will start. The game will end when either:

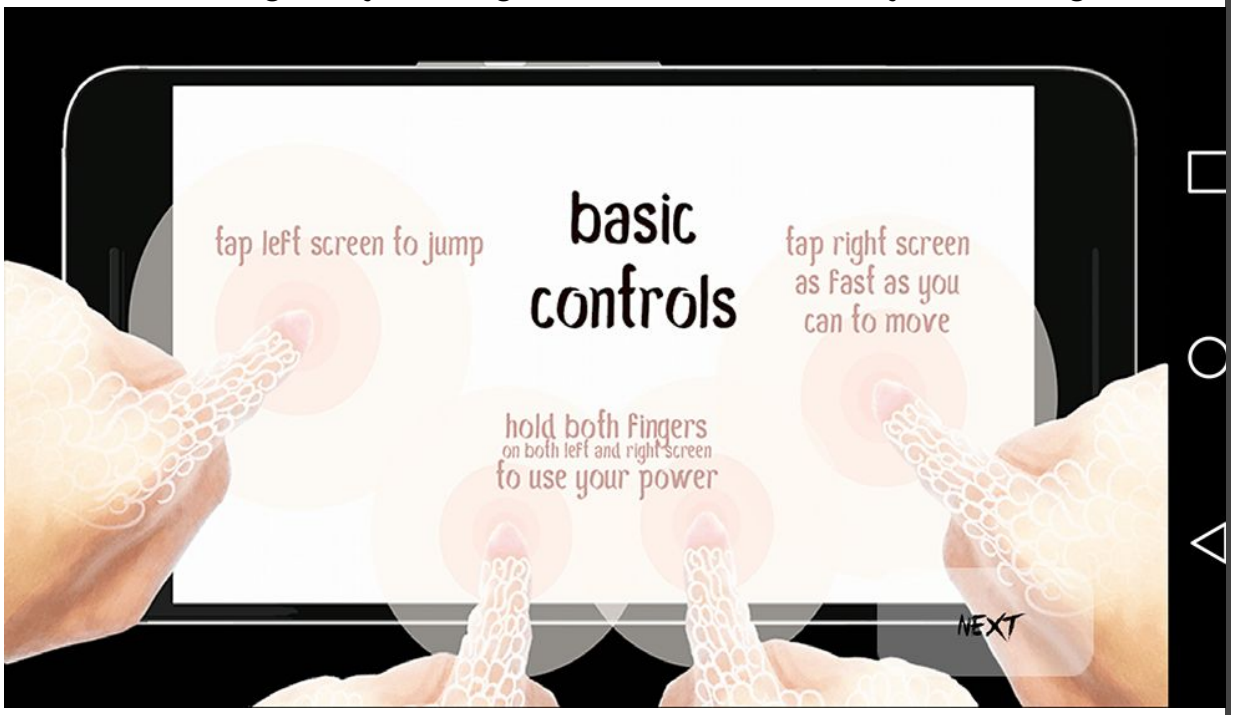
[1] Time runs out
[2] All players reach the end point

In both cases, players will be ranked accordingly to the distance travelled when the game ends.



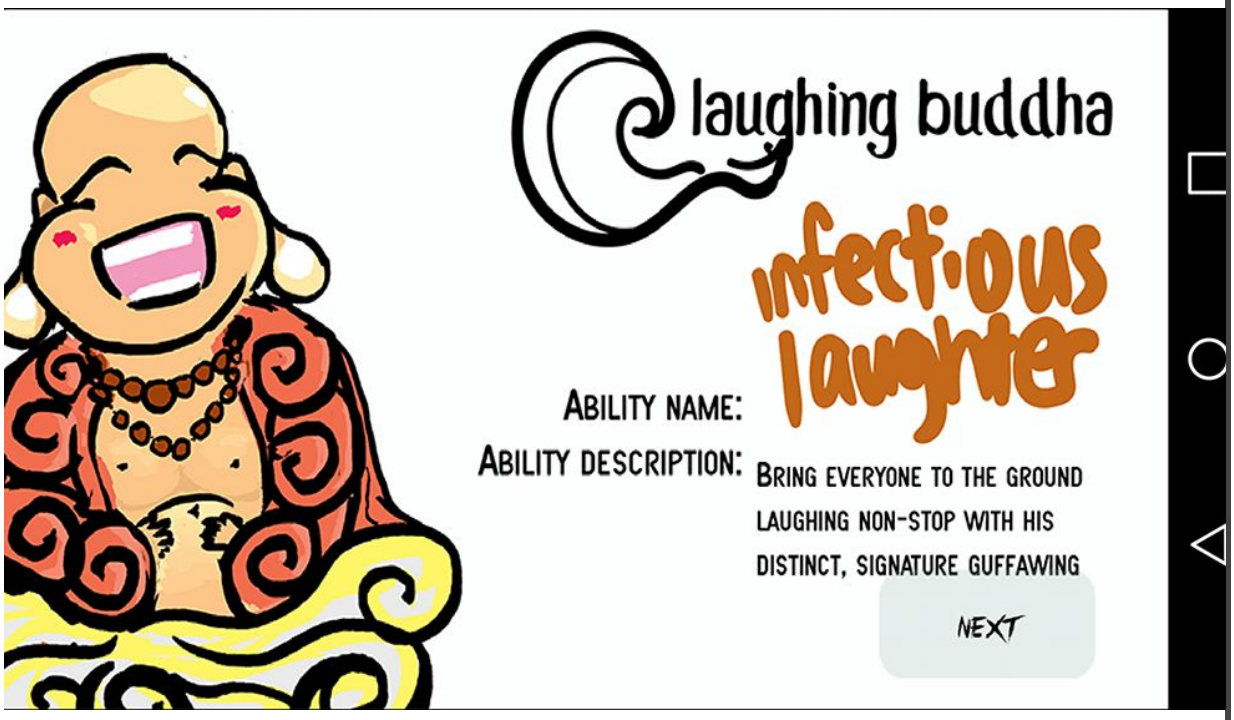
[1.0] START SCREEN | [1.1] WAITING SCREEN | [1.2] CHARACTER SELECTION SCREEN | [1.3] PLAYING SCREEN
[2] TUTORIAL SCREEN | [3] CHARACTER DESCRIPTION SCREEN

The tutorial screen guides you through the basic controls and ways to win the game.



[1.0] START SCREEN | [1.1] WAITING SCREEN | [1.2] CHARACTER SELECTION SCREEN | [1.3] PLAYING SCREEN
[2] TUTORIAL SCREEN | [3] CHARACTER DESCRIPTION SCREEN

Each of the six characters has its own special ability. Each ability comes with a description. Press the next button located on the bottom right of the screen to proceed to the next character's description. After all six images are shown, you'll be able to return to the Start Screen.



Testing

Method:

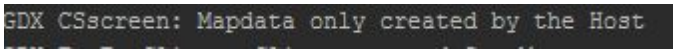
The flow of the game was checked by inserting libGDX's log method in strategic parts of the code. This allows us to check whether the game uses certain parts of the code and whether they are going in the right direction. We have below the important log locations:

Server start-up Testing:

In order to make sure the devices has been connected, we have our first log which connects and configures all the events. As soon as the server has started it starts waiting until the next log is displayed.

Random World-Map Render Test:

As soon as the connection has been configured, the server renders a random sequence for our map. The sequence of the map is then saved in the server before making the connection. In order to verify if the mapdata has been created, we have created a log to make sure the instance is created and saved.



Client-Server Connection Testing:

The client is connected to the server through the localhost and the transfer of data containing the mapdata is sent from the host to client, other information regarding other players name, are all first sent to the server and then server broadcasts these information to other client. All this info is kept track of using logs in each methods upon success. We can see the records below:

1. **Client Side:**

```
GDX WaitScreen: Finished connecting & configuring events
GDX TapTapClient: Client created GameMap
GDX Client: Client instantiated
GDX CSscreen: Player connecting to LAN.
00:00  INFO: [kryonet] Discovered server: /127.0.0.1
00:02  INFO: Connecting: /127.0.0.1:12080/12082
GDX TapTapClient: Connected to LAN successfully!
00:02  INFO: [kryonet] Connection 2 connected: /127.0.0.1
GDX TapTapClient: Connected to server
00:02  INFO: [Network]: Login initialised by Client nikki23
GDX Client: Connection handled, sent Login
```

2. **Server Side:**

```
GDX WaitScreen: Finished connecting & configuring events
GDX CSscreen: Mapdata only created by the Host
GDX TapTapClient: Client created GameMap
GDX Client: Client instantiated
GDX GameMap: GameMap instantiated in Server
00:00  INFO: [kryonet] Server opened.
GDX Server: Server instantiated
00:00  INFO: Connecting: localhost/127.0.0.1:12080/12082
00:00  INFO: [kryonet] Connection 1 connected: /127.0.0.1
00:00  INFO: [kryonet] Connection 1 connected: localhost/127.0.0.1
GDX TapTapClient: Connected to server
00:00  INFO: [Network]: Login initialised by Client nikki23
GDX Client: Connection handled, sent Login
GDX Server: Login received
GDX Server: Added a player
00:00  INFO: [Network]: PlayerJoinLeave initialised by Server for Client 1 null
GDX Server: sent Player coordinates to new player
GDX TapTapClient: Received message
GDX TapTapClient: Client received MapDataPacket
GDX Server: Stored new player in Server's map
GDX TapTapClient: Received message
GDX TapTapClient: Client received PlayerJoinLeave
GDX Client: onConnect called
00:06  INFO: [kryonet] Connection 2 connected: /127.0.0.1
GDX Server: Login received
GDX Server: Added a player
00:06  INFO: [Network]: PlayerJoinLeave initialised by Server for Client 2 null
GDX Server: sent Player coordinates to new player
GDX Server: Stored new player in Server's map
```

User Input Testing:

To test if the inputs are being recorded correctly by the program, we assigned controls to the keys in the keyboard and tested it through logging information in each click event. The handleinput() function handled the keyboard events as well as the ontouch events. Hence we verified the information by checking logs in case of each events and to check the touch event, we looked at the X and Y coordinates when it was clicked in the desktop mode to verify the touch encountered is within the gamewidth and gameheight region.

To check the special power testing, we assigned each special power with a different letter keypress. They are described below:

1. **Powerup Controls Testing:**

```
if (Gdx.input.isKeyJustPressed(Input.Keys.A)){       //testing purposes only
    Gdx.app.log("Attack type", "Stunned");
    return this.stun();}
if (Gdx.input.isKeyJustPressed(Input.Keys.S))   {    //testing purposes only
    Gdx.app.log("Attack type", "Poisoned");
    return this.poison();}
if (Gdx.input.isKeyJustPressed(Input.Keys.D)) {      //testing purposes only
    Gdx.app.log("Attack type", "Reversed");
    return this.reverse();}
if (Gdx.input.isKeyJustPressed(Input.Keys.F)){       //testing purposes only
    Gdx.app.log("Attack type", "Terror");
    return this.terror();}
if (Gdx.input.isKeyJustPressed(Input.Keys.G)) {      //testing purposes only
    Gdx.app.log("Attack type", "Flashlight");
    return this.flash();}
if (Gdx.input.isKeyJustPressed(Input.Keys.H)) {      //testing purposes only
    Gdx.app.log("Attack type", "Confused");
    return this.confuse();
}
```

2. **General Control Testing:**

```
if (Gdx.input.isKeyJustPressed(Input.Keys.UP))
    if (isConfused()) {
        Gdx.app.log("Attack type", "Confused");
        return this.moveRight();
    } else {
        Gdx.app.log("Input type", "Jump");
        return this.jump();
    }
if (Gdx.input.isKeyJustPressed(Input.Keys.RIGHT))
    if (isConfused()) {
        Gdx.app.log("Attack type", "Confused");
        return this.jump();
    } else {
        Gdx.app.log("Input type", "Move front");
        return this.moveRight();
    }
```

Player Attacked updates:

When a special power is casted, we check if all players receive this information in their own screen. The sound clip associated with a special power has to be checked against too.

Players Move message transfer:

Whenever a player make a move, a UDP packet is sent to the server and eventually broadcasted to all the clients. To make sure each packets are successfully transferred, we logged the info and tested by moving a player and checking if it was received by the other player. Also to know what kind of UDP packet was sent at each point, we logged information like, “client received movement State”, “client received GameRoom Full” etc.

Real-time movement updates:

To keep track of the moves, we send UDP packets only when a specific handleinput request has been handled by a particular player. The UDP packet contained the information of the coordinates where the player was at and then it was rendered accordingly to each player's local device. Whenever a packet is received, an info log is displayed to acknowledge the receipt of the packet. We log if the package is sent and received to debug precisely where the drop occur. We also log the player id number to the corresponding package received to ensure that the moves are being mapped appropriately.

1. **Server side packet updates (local player & server logs) :**

```
GDX GameMap: Sent MovementState to Server
GDX TapTapServer MovementState: MovementState received
GDX GameMap: Player null moved
GDX Player: set Movement State
GDX TapTapClient: Sent packet UDP
GDX GameMap: Sent MovementState to Server
GDX TapTapServer MovementState: MovementState received
GDX GameMap: Player null moved
GDX Player: set Movement State
GDX TapTapServer MovementState: MovementState received
GDX GameMap: Player null moved
GDX Player: set Movement State
GDX TapTapClient: Sent packet UDP
GDX GameMap: Sent MovementState to Server
GDX TapTapClient: Sent packet UDP
GDX GameMap: Sent MovementState to Server
GDX TapTapServer MovementState: MovementState received
```

2. **Client side packet updates (sent and received logs) :**

```
GDX GameMap: Sent MovementState to Server
GDX TapTapClient: Sent packet UDP
GDX GameMap: Sent MovementState to Server
GDX TapTapClient: Sent packet UDP
GDX GameMap: Sent MovementState to Server
GDX TapTapClient: Sent packet UDP
GDX GameMap: Sent MovementState to Server
GDX TapTapClient: Sent packet UDP
GDX GameMap: Sent MovementState to Server
GDX TapTapClient: Sent packet UDP
GDX GameMap: Sent MovementState to Server
GDX TapTapClient: Sent packet UDP
GDX GameMap: Sent MovementState to Server
GDX TapTapClient: Sent packet UDP
GDX GameMap: Sent MovementState to Server
GDX TapTapClient: Sent packet UDP
GDX GameMap: Sent MovementState to Server
GDX TapTapClient: Sent packet UDP
GDX GameMap: Sent MovementState to Server
GDX TapTapClient: Received message
```

Game and Decision:

Our game has 2 possible conclusions. In case the timer runs out and no one reaches the end, the results displayed are on the basis of the distance they have travelled. Else if all the players manage to reach the end, the player who crosses the line first is ranked first. To check the positions in each case, we log the information and make sure the results displayed are common to all players.

Player Disconnected:

In case a player disconnects from the game due to network reasons, we made sure that the game still continues with the other players and the disconnected player could still join in the next game. Also, once the game is over, we made sure that all the players are in the end game state and are disconnected from the server so that the next time they join they can either host or act as client to play the game. We also ensured that the players were redirected and the cleanup was complete and the scoring was cleaned up according to the number of players left at the end, who actually managed to stay connected till the end of the game. The log info was maintained in case a player is disconnected at any instance, either at end of game play or due to network issues.

1. Player disconnected in between gameplay:

```
GDX TapTapClient: Client disconnected from server
00:41  INFO: [kryonet] Connection 2 disconnected.
```

2. Player disconnect end game:

```
GDX TapTapClient: Received message
GDX TapTapClient: Client disconnected from server
00:45  INFO: [kryonet] Connection 1 disconnected.
00:45  INFO: [kryonet] Closing server connections...
00:45  INFO: [Network]: PlayerJoinLeave initialised by Server for Client 1 nikki23
00:45  INFO: [Network]: PlayerJoinLeave initialised by Server for Client 2 nikki23
00:45  INFO: [kryonet] Connection 2 disconnected.
00:45  INFO: [kryonet] Connection 1 disconnected.
00:45  INFO: [kryonet] Server closed.
```

Thread Safety Management

Most of the threading were abstracted by LibGDX and KryoNet, but operations specific to Nimbus Run were also made thread-safe to ensure overall thread-safety.

Screen Transition

Regarding the LIBGDX framework, there are two main thread types running in the game: the render thread and the other threads for anything else. One issue that we faced was ensuring smooth screen transition. To go from screen A to screen B, we need to ensure that screen B is not rendered during the rendering frame of screen A. Hence, we use Gdx.app.postRunnable to pass in a new Runnable that sets the screen to the next screen. This allows the new screen to be rendered in the next frame before render() is called.

Player Ordering

When players establish connection with the server, the players are given a number in the order they connect to the server and this number is used to determine which position the player starts the race at. On the server side, it keeps track of the ordering using an integer variable called PLAYERS, which is initialised as 0. When a player connects, in TapTapServer class, if PLAYERS < 4 (maximum player allowed to connect to server is 4), the server increments PLAYERS by 1 and passes on PLAYERS, connection ID and connection NAME to a Packet-making Factory. Thus, the comparison, incrementing and packet making have to be atomic. To ensure atomicity, the above operations are synchronised to a playersLock Object using a synchronised block.

Players Data Structure

In GameMap, every Player is stored in the players ConcurrentHashMap (likewise for DummyPlayer). “Players” contains Player classes, which is indexed by their player IDs, and is used to store the states of the various Player classes during the gameplay. The ConcurrentHashMap data structure allows concurrent updating of each Player during the gameplay when the Player connects, disconnects, moves, attacks, etc while ensuring thread safety at the same time.

Player Position Update

When a player receives movement states about other players, it first receives a MovementState packet from the server containing the coordinates and the linear velocity of the player moved. It then sets the movement state of that player in the player’s own game map, followed by applying the physics involved using world.step() (Box2d’s API). The nature of Box2D is that during world.step(), no variable involving the Box2D bodies should be changed. This means that the Player coordinates and linear velocities received from the MovementState packet should not be updated during

world.step(). To tackle this, we used synchronised block to lock world.step() and player.setMovementState() to the GameMap class in GameMap.java to serialise the process.

Use Cases

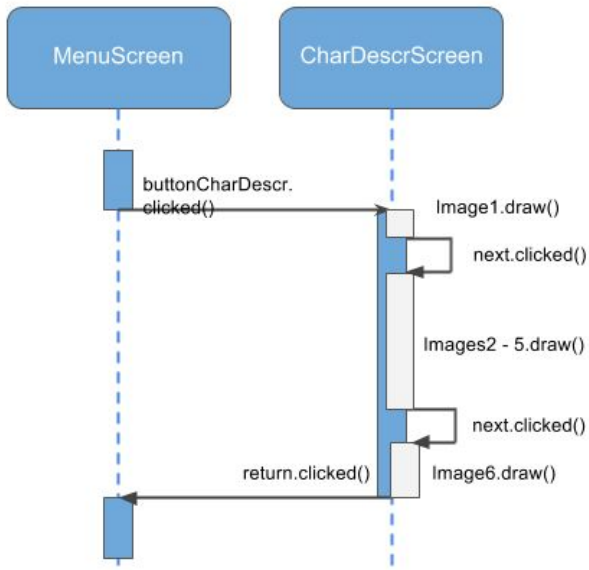
Use Case 1: Tutorial Screen

Name	Tutorial Screen
Objective	To help player understand the basics of the game with tutorial guidance with all controls.
Pre-conditions	As soon as the player reaches the menu screen, he can go to the tutorial screen.
Post-Conditions	Success ● Player goes through two textures, and needs to go through each texture background in order to return to the main menu and start game. Failure ● Game fails to load tutorial screen. ● Game fails to load the second texture of the tutorial screen and crashes.
Actors	Primary ● Player
Trigger	The player clicks on tutorial button on main menu screen.
Normal Flow	1. Player reads the tutorial. 2. Player moves back to the menu screen
Alternative Flow	Once clicked on tutorial button, player needs to go through this screen.

Use Case 2: Character Description Screen

Name	Character Description Screen
Objective	To help player understand the options they have for their choosing of characters, with descriptions of the special ability of each character.
Pre-conditions	Currently in the Menu Screen. Press the “Character Description” button to access the Character Description Screen.
Post-Conditions	Success ● Player goes through six background textures, and need to go through each texture to return to the main menu and start game. Failure ● Game fails to load character screen. ● Game fails to load the subsequent texture background of a particular character in the character screen and crashes.
Actors	Primary ● Player
Trigger	The player clicks on characters button on main menu screen.
Normal Flow	1. Player reads the the abilities of each character type.. 2. Player moves back to the menu screen
Alternative Flow	Once clicked on characters button, player needs to go through this screen.

Use Case 2: Character Description Screen



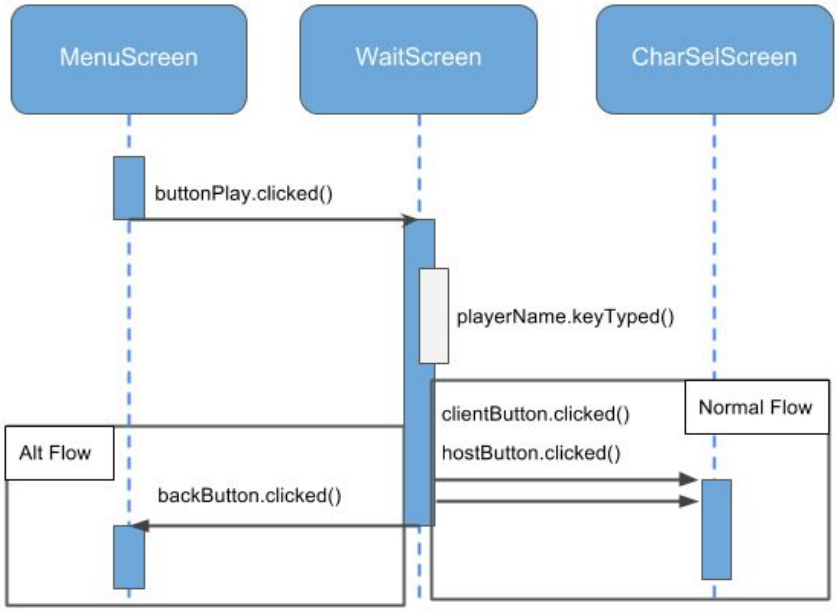
Use Case 3: Wait Screen

Name	Wait Screen (Connection Room)
Objective	Allow players to enter his/her name and either host the game or join the game as clients
Pre-conditions	1. Player must press the start button in the menu screen. 2. Player must have active internet connection and all the players need to be in the same LAN network. 3. Players need to decide amongst themselves for who is hosting the game.
Post-Conditions	Success ● The player has joined the game either as a server or a client. ● If player joins as client, the server instantiates the player with his/her player name. ● The connection is made and players are taken to the character selection screen. Failure ● More than one person clicks on join as host. ● More than 4 players try connecting to the same host.
Actors	Primary ● Player (Host or clients) Secondary ● Kryonet connections
Trigger	Players click on join as host or join as client buttons.
Normal Flow	1. Player clicks on join button. 2. Player is directed to character selection screen, which is also our lobby room.
Alternative Flow	1. Player can go back to menu screen instead of joining the game.

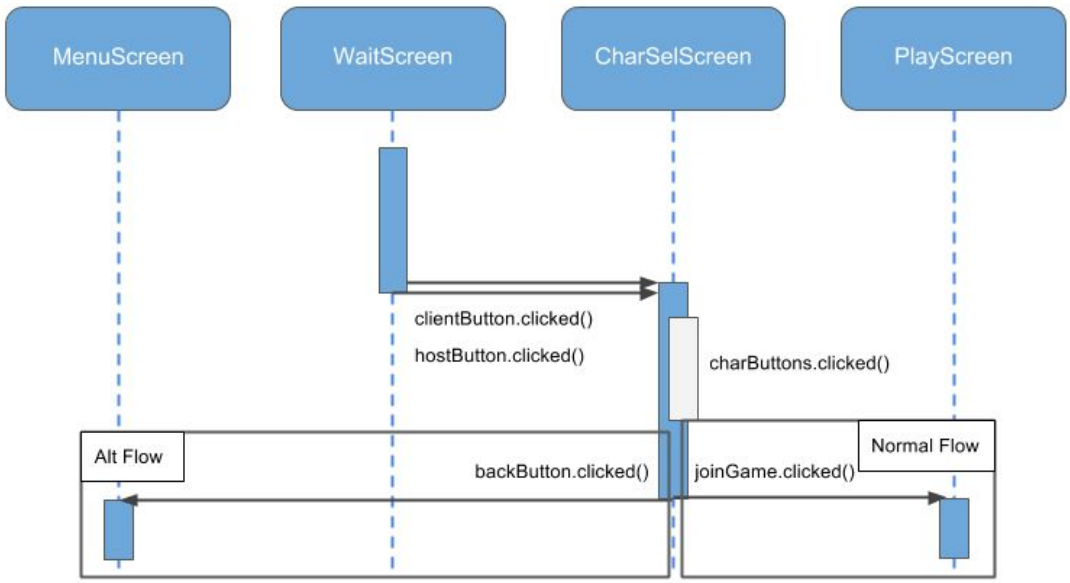
Use Case 4: Character Selection Screen

Name	Character Selection Screen (Wait Room)
Objective	Allow players to choose which character they want to play as in the main game.
Pre-conditions	1. Player must join the game and instantiate the socket connection as server or client. 2. Player must have active internet connection and all the players need to be in the same LAN network.
Post-Conditions	Success ● The player can decide which character his desired character and proceed by clicking join game. Failure ● Player is unable to join the game and cannot reach the screen due to being unable to connect to host, or there is no host (null server)
Actors	Primary ● All connected Players (Host and clients)
Trigger	Players click on join game
Normal Flow	1. All players choose their desired characters swapping through the buttons. 2. All the client players need to press the trigger first. 3. If all clients have not pressed their trigger yet and the server has pressed the trigger, the game waits for the client to press the trigger. 4. The server players presses the trigger. 5. The game starts.
Alternative Flow	1. Player can go back to wait screen or menu screen and exit the ongoing game.

Use Case 3: Wait Screen



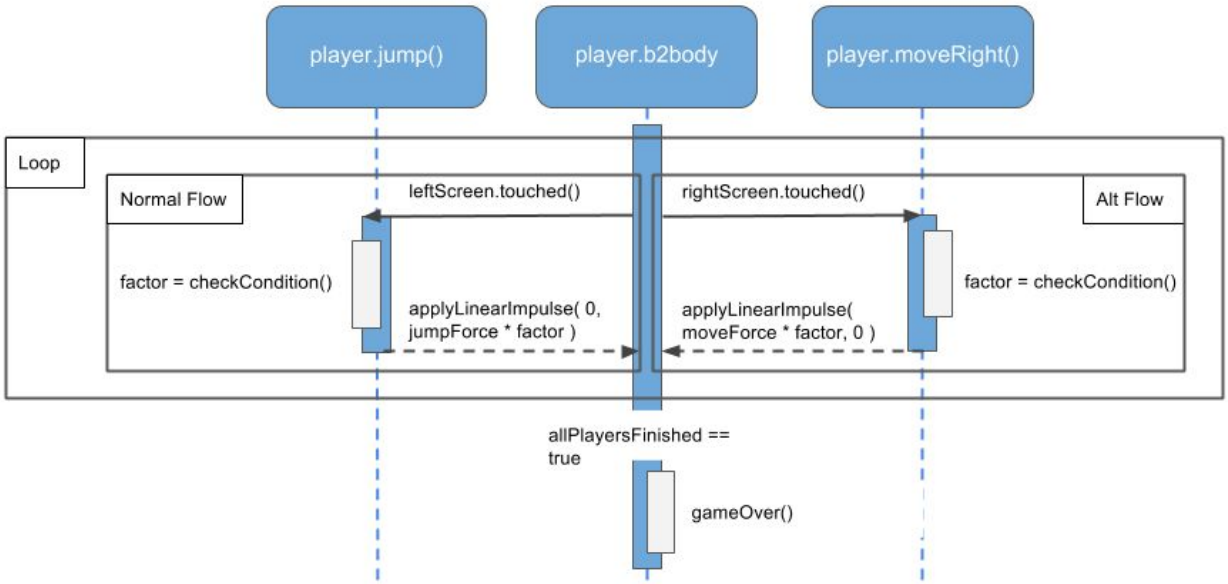
Use Case 4: Character Selection Screen



Use Case 5: Character Movement

Name	Character Movement
Objective	Player to jump and run to the end of the map
Pre-Conditions	Player must be in PlayScreen and GameMap instantiated
Post-Conditions	Success: ● Player jumps with appropriate force when triggered ● Player runs with appropriate force when triggered Failure: ● Player moves when not supposed to (i.e. after reaching finish line) ● Player does not move when supposed to (faulty trigger mechanism)
Actors	Primary: ● Local player Secondary: ● Other players
Trigger	Player touches left or right side of the screen
Normal Flow	1. Player touches left side of screen 2. Check if player is debuffed/finished race 3. Applies corresponding jump force
Alternative Flow	1. Player touches right side of screen 2. Check if player is debuffed/finished race 3. Applies corresponding move force
Interacts With	Player ability use cases
Exceptions	Race has not yet begun, cannot move Player finished race, cannot move Player is stunned, cannot move Player is poisoned, movement is greatly hindered Player is reversed, runs backwards instead Player is confused, jump and run triggers swapped Player is terrored, involuntarily pulled backwards Player is flashed, unable to see screen and navigate past obstacles effectively

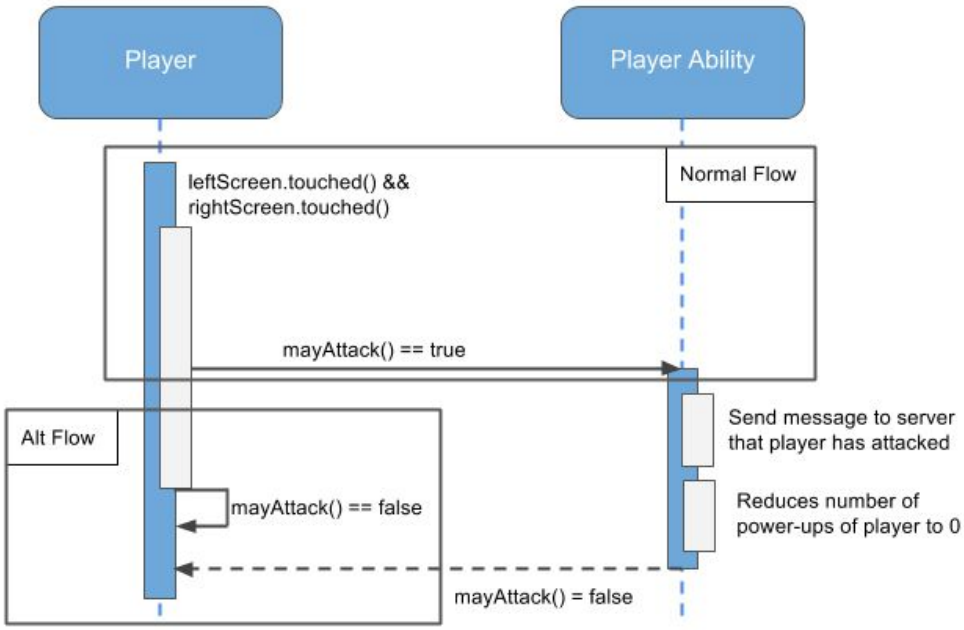
Use Case 5: Character Movement



Use Case 6: Player Ability

Name	Player Ability
Objective	Player to use player ability when ready
Pre-Conditions	Player must be in PlayScreen and GameMap instantiated
Post-Conditions	Success: ● Player calls his/her ability when ready Failure: ● Player calls his/her ability when not ready
Actors	Primary: ● Local player Secondary: ● Other players
Trigger	Player touches both left and right sides of the screen simultaneously
Normal Flow	1. Player touches both sides of screen when ability ready 2. Ability is called, all other players affected 3. Ability not ready
Alternative Flow	1. Player touches both sides of screen when ability NOT ready 2. Nothing happens
Interacts With	Character movement use case

Use Case 6: Player Ability

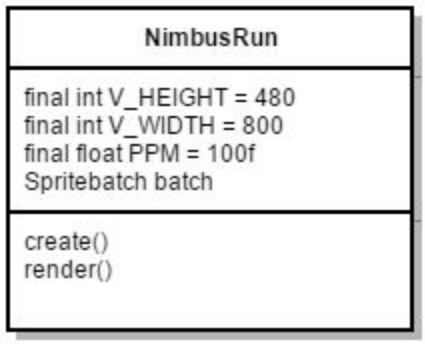


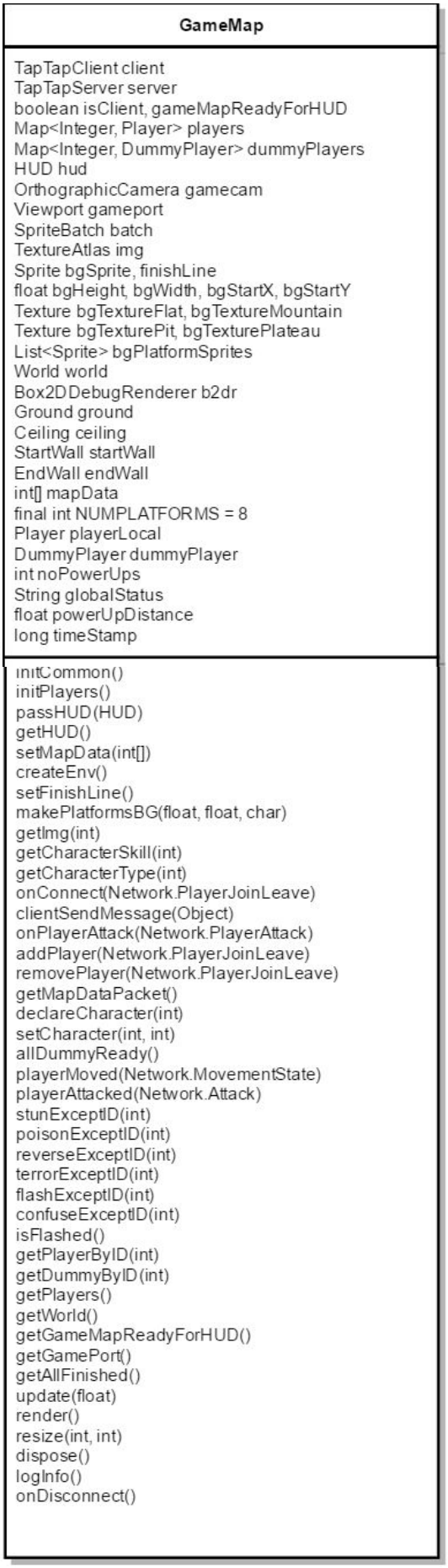
Appendix

Class Diagrams

Game Logic

1. NimbusRun.class: Entry point class that uses LibGDX. Overarching class that controls the state of the game, most notable used to set the main menu screen and start game.
2. GameMap.class: Consists of all the logic used for the main gameplay (race), facilitates a game session and manages all the dynamic graphics




GameMap
TapTapClient client
TapTapServer server
boolean isClient, gameMapReadyForHUD
Map<Integer, Player> players
Map<Integer, DummyPlayer> dummyPlayers
HUD hud
OrthographicCamera gamecam
Viewport gameport
SpriteBatch batch
TextureAtlas img
Sprite bgSprite, finishLine
float bgHeight, bgWidth, bgStartX, bgStartY
Texture bgTextureFlat, bgTextureMountain
Texture bgTexturePit, bgTexturePlateau
List<Sprite> bgPlatformSprites
World world
Box2DDebugRenderer b2dr
Ground ground
Ceiling ceiling
StartWall startWall
EndWall endWall
int[] mapData
final int NUMPLATFORMS = 8
Player playerLocal
DummyPlayer dummyPlayer
int noPowerUps
String globalStatus
float powerUpDistance
long timeStamp
initCommon()
initPlayers()
passHUD(HUD)
getHUD()
setMapData(int[])
createEnv()
setFinishLine()
makePlatformsBG(float, float, char)
getImg(int)
getCharacterSkill(int)
getCharacterType(int)
onConnect(Network.PlayerJoinLeave)
clientSendMessage(Object)
onPlayerAttack(Network.PlayerAttack)
addPlayer(Network.PlayerJoinLeave)
removePlayer(Network.PlayerJoinLeave)
getMapDataPacket()
declareCharacter(int)
setCharacter(int, int)
allDummyReady()
playerMoved(Network.MovementState)
playerAttacked(Network.Attack)
stunExceptID(int)
poisonExceptID(int)
reverseExceptID(int)
terrorExceptID(int)
flashExceptID(int)
confuseExceptID(int)
isFlashed()
getPlayerByID(int)
getDummyByID(int)
getPlayers()
getWorld()
getGameMapReadyForHUD()
getGamePort()
getAllFinished()
update(float)
render()
resize(int, int)
dispose()
logInfo()
onDisconnect()

Screens

1. SplashScreen.class: This screen is the first screen that pops up and stays there for few seconds, as the app opens in the phone.

2. MenuScreen.class: As soon as the splashscreen is over, menuscreen is displayed wherein the users can learn more about the game, the characters and how to play by using the various control buttons to go to other classes.

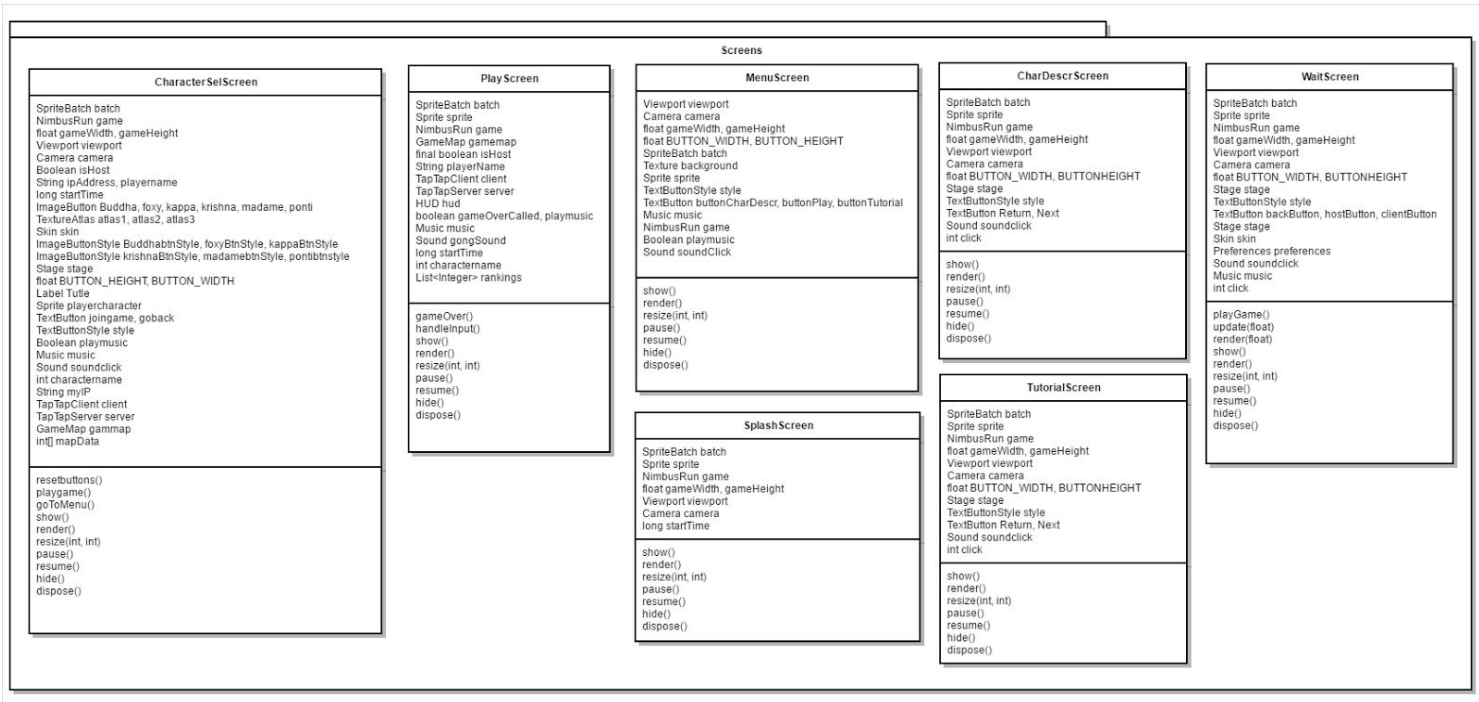
3. TutorialScreen.class: Users can learn about the gameplay, the controls and what exactly are they supposed to do in the game through this screen.

4. CharactersScreen.class: This class describes the characters and their special abilities that they can later choose from the character selection screen so that they are familiar with the attacks/power-ups that they can either use or be used upon during the gameplay.

5. WaitScreen.class: The start of the server and the connections between the server and clients are made in this class. Users can choose to host the game or instantiate as clients and proceed towards choosing their character.

6. CharacterSelectionScreen.class: Before proceeding to the Playscreen.class where the real game starts, the players can choose their desired avatar according to what they read through in the CharacterScreen.class.

7. PlayScreen.class: This is the main class where characters/players interact with each others and which uses the GameMap.class and controls all the other classes in a hierarchical order until the game has ended and the results have been displayed..



Sprites

1. Ceiling.class: Used for Box2D rendering. It is an edgeShape (box2D line) to represent the ceiling which marks the highest jump the player can make during the gameplay.

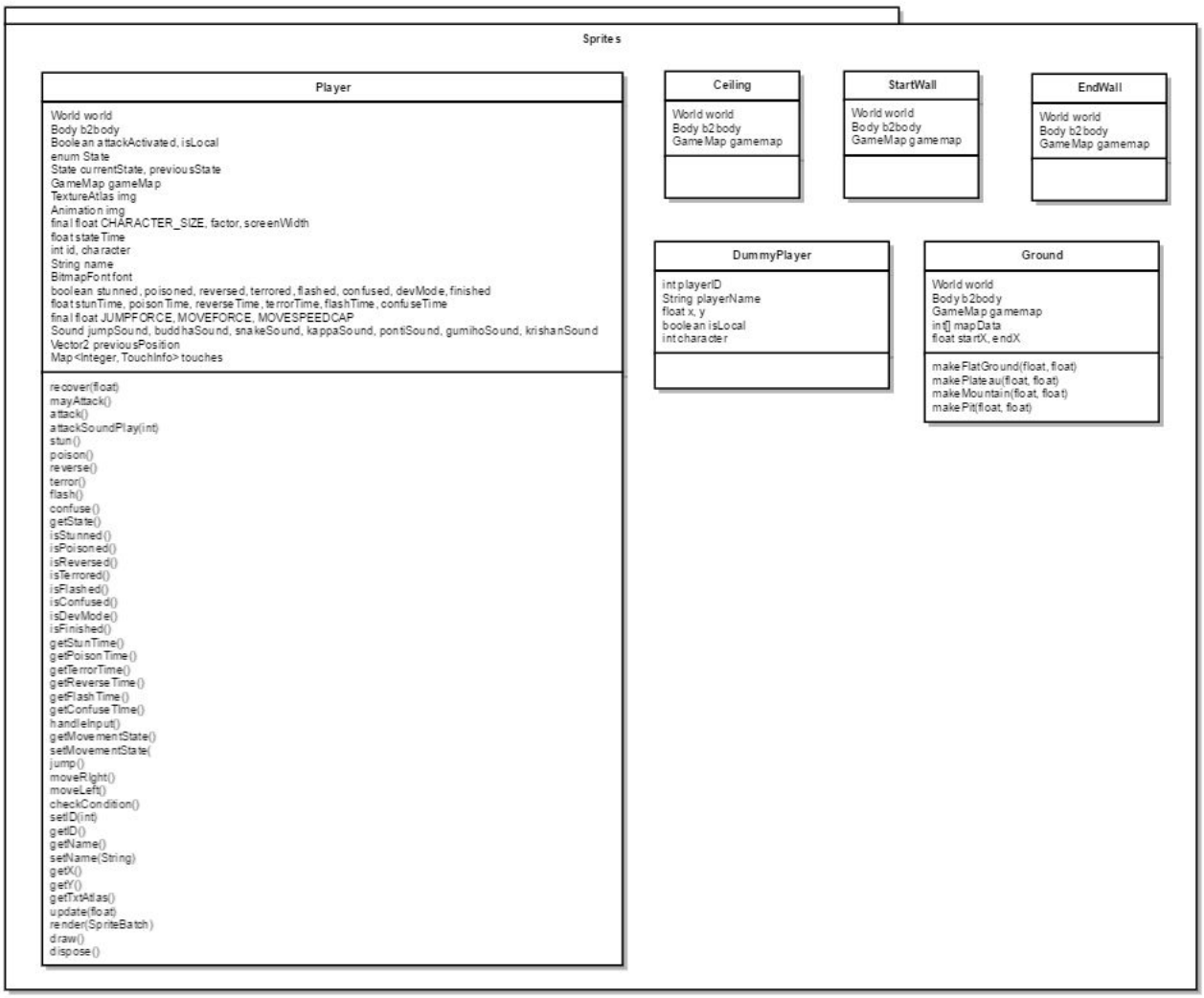
2. Ground.class: Used for Box2D rendering. It is an edgeShape (box2d line) to represent the endWall which marks the end of the gameplay when the player crosses it.

3. StartWall.class: Used for Box2D rendering. It is an edgeShape (box2D line) to represent the startWall which marks the start of the gameplay where they players are actually positioned during start of the game.

4. GameMap.class: Used for Box2D rendering. Renders the map wherever the game is hosted. It allows interaction between players and initializes the client, the server, the HUD and monitors through the network and interactions.

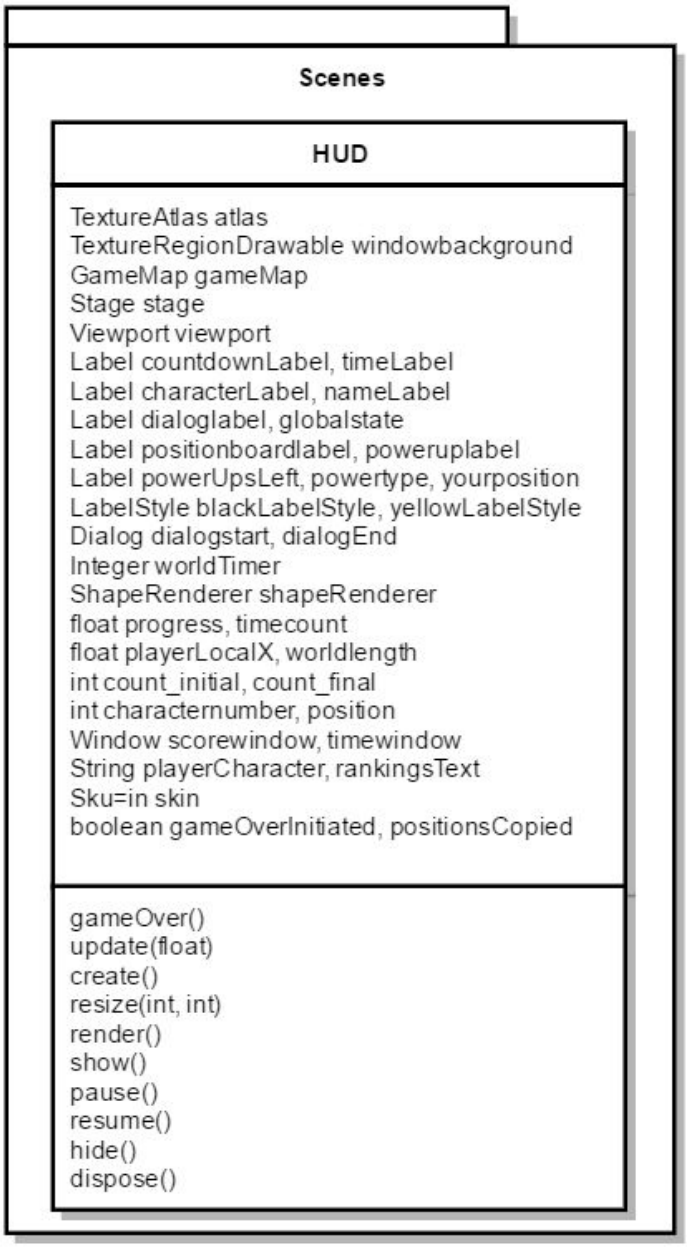
5. Player.class: Used for Box2D rendering. Basically renders each players on the gameMap depending upon whether it is a local player or some other player. Also update the power ups, creates the sounds and imposes the attack when on the local player when a particular type of attack is made by some other player. This class also handles the inputs from the users.

6. EndWall.class: Used for Box2D rendering. It is an edgeShape (box2D line) to represent the endWall which marks the end of the gameplay where the game ends and player has successfully crossed the endline.


Sprites
Player
World world
Body b2body
Boolean attackActivated, isLocal
enum State
State currentState, previousState
GameMap gameMap
TextureAtlas img
Animation img
final float CHARACTER_SIZE, factor, screenWidth
float stateTime
int id, character
String name
BitmapFont font
boolean stunned, poisoned, reversed, terrored, flashed, confused, devMode, finished
float stunTime, poisonTime, reverseTime, terrorTime, flashTime, confuseTime
final float JUMPFORCE, MOVEFORCE, MOVESPEEDCAP
Sound jumpSound, buddhaSound, snakeSound, kappaSound, pontiSound, gumihoSound, krishanSound
Vector2 previousPosition
Map<Integer, TouchInfo> touches
recover(float)
mayAttack()
attack()
attackSoundPlay(int)
stun()
poison()
reverse()
terror()
flash()
confuse()
getState()
isStunned()
isPoisoned()
isReversed()
isTerrored()
isFlashed()
isConfused()
isDevMode()
isFinished()
getStunTime()
getPoisonTime()
getTerrorTime()
getReverseTime()
getFlashTime()
getConfuseTime()
handleInput()
getMovementState()
setMovementState(
jump()
moveRight()
moveLeft()
checkCondition()
setID(int)
getID()
getName()
setName(String)
getX()
getY()
getTxtAtlas()
update(float)
render(SpriteBatch)
draw()
dispose()
Ceiling
World world
Body b2body
GameMap gamemap
StartWall
World world
Body b2body
GameMap gamemap
EndWall
World world
Body b2body
GameMap gamemap
DummyPlayer
int playerID
String playerName
float x, y
boolean isLocal
int character
Ground
World world
Body b2body
GameMap gamemap
int[] mapData
float startX, endX
makeFlatGround(float, float)
makePlateau(float, float)
makeMountain(float, float)
makePit(float, float)

Scenes

1. HUD.class: Though the HUD.class also implement screen, it basically draws another layer of screen above the playscreen with the labels describing the timer, the power-up if it's ready to use or not, and your own player information.



Networking

1. TapTapClient: This class handles the Kryonet Client logic. It contains listeners for Kryonet connection, packets and disconnection. GameMap is also initialised in this class to be rendered for the user.
2. TapTapServer: This class handles the Kryonet Server logic. It contains listeners for packets from clients and client disconnection. Special unrendered GameMap is instantiated for the server in the class to check for valid moves from the clients.
3. Network: This class contains static methods and static final variables that are used by both TapTapClient and TapTapServer and static classes that define packets to be sent between the clients and the server.