2D Browser-based Multiplayer Game Development

Final year project report for Bachelor of Computer Science

Massey University

Jack Chen

Student ID: 16095664

Supervised by Dr Daniel Playne

2018

Acknowledgement

I would like to thank my supervisor Dr Daniel Playne for providing kindly support and guidance throughout this project. I have learnt so much not only from the help for this project but the knowledge and his personal experience he was willing to share with me. I would not be able to get to this stage without his generous assistance.

I also want to thank my sister, Helen Chen, for her willingness to proof read my project report and share her knowledge about the structure of the written report.

Abstract

The aim of this project is to build a 2D browser-based multiplayer game without using a game engine. The idea of this game is inspired by Agar.io, a 2D browser-based multiplayer action game, where the player controls a round object intending to eliminate other players.

This project is mostly written in JavaScript and designed by the concepts of diversity and cooperation. Players with different characters in a team cooperate with each other and compete against other teams.

The controls of the game are solely with mouse for simplicity. Three types of characters, Line, Circle and Rectangle, are made for users to choose from. Collision detections take an important role in this game to enable interactions between objects.

To allow users to play within a game arena on different machines, network communication is required. Node.JS with frameworks Express and Socket.IO are the technologies used to build the server. The program runs smoothly when the server is hosted in a local machine. However, a high latency issue occurs when the program is transferred into public web-hosting service. There are further efforts that need to be done to make the game more enjoyable and stable.

Table of Contents

1. Introduction

The game industry has been significantly influenced by the rapid digital advancement. Game companies tend to develop heavy graphics games as more and more powerful devices such as gaming desktops and consoles are launched. However, 2D browser-based games like Agar.io or Slither.io still own a significant number of players in the gaming market. According to Google Trend (2018), Agar.io and Slither.io were the most searched game related keywords in the US in 2016. Slither.io was even the 9th most searched general keyword globally in that year. The simplicity and easy-to-start characteristics of these '.io' games give them the ability to survive among those stunning 3D graphics games.

The aim of this project is to build a 2D multiplayer browser-based game like Agar.io, but also comes with different concepts and gameplay, and allow the game to be extensible with more choice of characters.

2. Inspiration

This project is inspired by a multiplayer online browser-based action game 'Agar.io' which was created by Matheus Valadares in 2015 (Figure 1). Players in the game are expected to control one or more 'cells' (round objects) in a plane. The goal of the game is to gain as much mass as possible by consuming the agars (public round objects) or other cells controlled by other players that are smaller in size than itself. The three main ideas, a primitive shape controlled by a player, public objects that every player can interact with, and a big canvas where all the players are within it, were heavily used for this project.

Figure 1. Agar.io

3. Game Design and Rules

Similar to 'Agar.io', this project is aimed at building a 2D multiplayer browser-based game with additional characteristics. Two concepts, diversity and cooperation, were used to make this game different from other games with similar gameplay. Diversity means there are multiple options of characters with different styles of gameplay that can be chosen. Cooperation means players have to cooperate with other players in the same team to survive and win.

3.1. Objectives and design strategies

Before the start of the game, players are asked to choose a character. There are three types of characters that can be chosen by users: Line, Circle and Rectangle. Each character has its specialities and weaknesses. They can all survive and even win on their own. However, players should aim at doing what the character is good at and cooperate with teammates to increase the chance of winning. The objective of the game is to eliminate other teams by cooperating with teammates of different types.

The design strategy cooperation is reinforced by minimising but not eliminating the ability of each character to win the game. For a circle, it can survive by itself, but it has little ability to attack other players, which makes it difficult to win solely. For line and rectangle characters, they both can attack opponent with large damages. However, they have to rely on circle players to feed them with energy balls to grow sizes or to be effectively healed. With rectangle pushing the point objects for a circle, the team could be built faster, which increases the chance of winning the game. These kinds of rules force players in a team to cooperate.

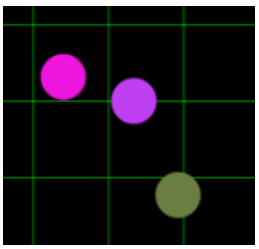
3.2. Controls

The game is intentionally made with controls solely by mouse for simplicity. To control the movement of player object, simply move the mouse cursor in the game canvas will make player object intend to move toward the cursor position. The speed of the player object can be determined by the distance with the mouse cursor, i.e. the distance goes further, the speed goes faster. Left or right clicking the mouse will activate different features depending on the type of character. The detail features and controls are described as followed.

3.3 Game Object

This section introduces all the objects including public and player objects that will appear in the game canvas.

3.3.1. Public point object



These are the public point objects rendering as small circles that each player can interact with (Figure 2). The total number of point objects in a canvas is fixed. Whenever a point object disappears in the canvas by either being consumed or crushed by players, it will be recreated at a random position by the server to maintain the total number in the canvas.

Figure 2. Public point objects

3.3.2. Circle player object

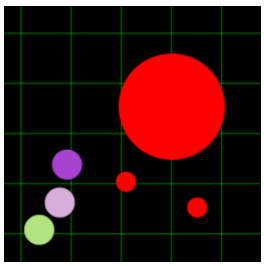


Figure 3. Player object and energy balls

As a circle player, the main job of it is to gain mass by consuming the public point objects (Figure 3). It is the only character that can grow size effectively. Circle players also need to feed up other players by emitting energy balls (small circle objects with the same colour as players). It is also the only character that can make other players grow sizes effectively.

Invisible mode: Circle player objects can become invisible within a timeframe by left-clicking the mouse. It can be used to avoid attacks from other players. This is also the only time that the circle player objects can grow bigger or restore health amount by consuming point objects. Circle players are back to normal mode after the timeframe. A cooldown time will be applied to allow Circle objects to become invisible again.

Energy balls objects: Energy ball is an object owned by a circle player that can only be consumed by other players. It can be emitted by right-clicking the mouse and can be consumed by colliding with player objects. Energy ball is a great way of restoring or growing size to a teammate of any type. Energy ball turns toxic to enemies and it will cause damages to them regardless of types. The size of a circle player will be reduced when it emits the energy ball. Circle player cannot emit energy ball when the minimum mass of the player size is reached.

Collision with others: There is no interaction between a circle player and other teammate players of any types. The size of circle players will be deducted when colliding with any type of opponent players. It becomes the weakest character once a collision occurred.

3.3.3. Line player object

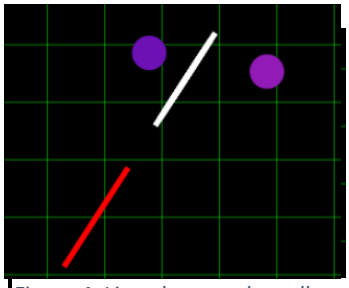


Figure 4. Line player and needle

A player as a line object is supposed to attack players of other teams (Figure 4). Like a circle player object, line player has the ability to emit objects (needles, white line segments) from the front end of the line, but it can only be used to attack others. It is the only character that can attack other characters without colliding. The front end of the line is controlled by the direction of the mouse cursor.

Needles objects: Needle is an object belonging to a line player. It can be emitted by left-clicking the mouse. It is used to attack the enemy players of any type by colliding and it has no interaction with teammate players. The number of needles available to be emitted can be shown at the bottom right of the screen during the game.

Restore mode: By right-clicking the mouse, a line object player can switch between either to restore health amount or reload ammos (needles) when touching the public point objects by the front end of the line or consuming energy balls from teammates.

Collision with others: There is no interaction when colliding with teammate players. When colliding with opponents of line or rectangle types, it will cause damages to the line player. While colliding with circle player, the circle player will get damaged.

3.3.4. Rectangle player object

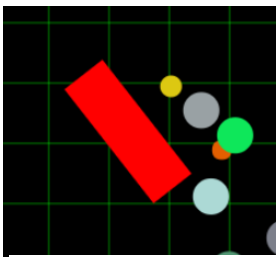


Figure 5. Rectangle player

Rectangle player object mainly acts as a protector and helper in a team (Figure 5). It has the most health amount and biggest size of all player objects. The main job is to push and collect the public point objects to an area for circle teammates to consume. Although circle can also push the point objects, the straight and wide edges and the rotatability of a rectangle make it easier to gather points than a circle with curvy outlines.

Rotation: the rotatability of the rectangle object allows it to control the point objects during collection easily, and to avoid attacks from needles objects. The rectangle can perform clockwise rotation by right clicking the mouse or counterclockwise rotation by left clicking.

Restoring: Restoring health amount or grow size for a rectangle player can be done by either crushing the point objects against the canvas boundaries or consuming the energy balls from circle teammates.

Collision with others: There is no interaction when colliding with teammate players. As the strongest character, when collision with opponent occur, the size of the opponent of any type will be minified when colliding with the rectangle object.

3.4. General Rules

3.4.1. Dependency and relationship

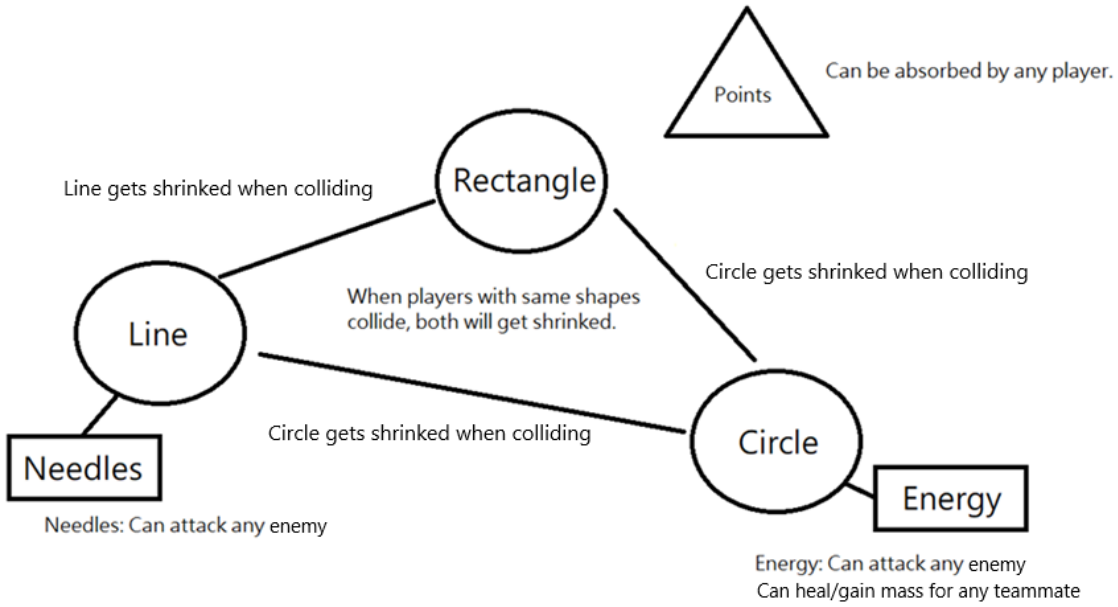


Figure 6. Relationship diagram

Figure 6 shows that each type of character has been defined a relationship with other characters. Needle and energy objects belong to Line and Circle characters respectively. Points are the public objects that any player can interact with.

3.4.2. Team balance

Before the start of the game, new players can choose one of the four teams with their preferences. Each team is supposed to hold a corner in the canvas as their base, and new-joined player will be assigned to a random position in a rectangular area near the base of their team, as shown in Figure 7.

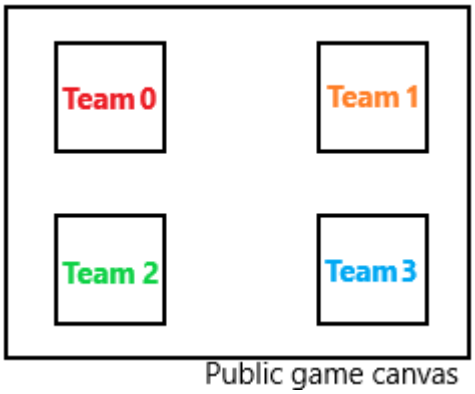


Figure 7. Team bases

Balancing the number of players in a team is an essential task for a multiplayer co-op game that allows new players to join any time without breaking the fairness of competition. To achieve this, the setting for this project is that each team requires at least one player in the beginning. Afterwards, there cannot be a team that has 3 more players than any other teams. If a new player selects a team that would result in breaking the rule, then it will be reassigned randomly to another team by the system to avoid this.

3.4.3. Eliminated condition

Each new player will start with a full health amount and a default size for the preferred type of character. Each type of character has a different maximum health amount and maximum size they can reach. Players will start losing their health amount when they reach the minimum size. Once the health amount reaches zero, the player is said to be eliminated, and will be kicked out of the game canvas. The player can choose to re-join the game by clicking the start button in the game menu.

3.4.4. Winning condition

The preset winning condition appeared when there is only one team with alive players. The issue of reaching the winning condition is that new players can join the game at any time. For example, two teams with few players fighting each other and several new players join the game in a sudden. This situation appears to break the fairness of competition. One team might almost be the winning team, but because of the newly-joined players acting as enemies, they have to fight against them again.

Limiting the total number of players that can join the team within a match might be a possible solution, but this has not been tested in this project.

4. Technologies

The front-end of this project is written primarily by JavaScript with the web page built using HTML/CSS. The back-end is also written by JavaScript under the environment provided with Node.js. Express and Socket.IO are the frameworks used to help set up the server.

4.1. Front End

As known, HTML, CSS and JavaScript are three of the core components of the World Wide Web. The front end of this project is fully done by them. HTML and CSS are used to form the menu and the game canvas, and JavaScript is used to handle user events, canvas rendering, and provide the network communication in the back-end.

At the early stage, programming languages such as Java and C# were also considered, but JavaScript had outperformed other options for this project because of the reasons:

- Google Chrome, as a currently most popular browser (Statista, 2018), does not support NPAPI, a technology needed for Java (Java, 2018). This makes Java not a good option for writing the font-end of a browser-based game.
- C# is a good programming language for game development, as some game engines with a fairly good quality like Unity and CryEngine have support APIs written for C# (Microsoft, 2018). However, the help of extra tool such as WebAssembly needed to run C# codes on the browser (Gaherwar, 2018) makes it not be considered for this project.

  WebAssembly is a relatively new technology that appeared first in March 2017. It can translate high-level languages such as C, C++ and C# into a binary format that can be run in the browser. Although WebAssembly can boost up the speed running in the browser that is nearly as fast as running native code, it is not fully compatible for all the browsers at the current stage (MDN Web Docs, 2018). With a game that needs to be run on the browser and does not require overwhelming graphics, there are better options available.
- The support in nearly all the modern browsers and the event-driven architecture makes JavaScript the best option among these languages for building the front-end of a relatively less graphic heavy browser-based game.

4.2. Back End

Known as a scripting language that handles user events in the front-end initially, JavaScript has been given the ability to work at the server side by Node.js. Extra Node.js frameworks ‘Express’ and ‘socket.IO’ are also used in this project to achieve more efficient communication with the front-end.

4.2.1. Node.js

As stated in Wikipedia (2018), “Node.js is an open-source, cross-platform JavaScript run-time environment that executes JavaScript code outside of a browser.” Node.js was first written by Ryan Dahl in 2009 and was built on Chrome’s V8 JavaScript engine. The event-driven and

asynchronous input/output processing architecture make Node.js a suitable option for the back-end of this project, as the game server will not be stuck at one place processing an event from a user. The utilisation of Node.js also allows to maintain codes with just one programming language as JavaScript is also used in the front-end.

4.2.2 Express and Socket.IO

Express framework is used to set up the server easier with Node.js. The main jobs of Express in this project are to set up an event handler for the server and to define the middleware and the route of the program.

Socket.IO is used to provide real-time and bidirectional communication between client and server in an easy way. Socket.IO provides both client-side and server-side libraries that allow both ends to send requests/responses to each other. The event-driven architecture is also a great characteristic to send corresponding updates for users. An easy example for the use of Socket.IO in this project is to send arrays with information needed for client-side rendering to each player from the server, and each client can also send user events to the server.

5. Implementation

5.1. Beginning

To start the project with Node.js environment, '*npm init*' needs to be entered in the command prompt at the project root. This command is to publish a new Node.js project in npm and to create a new package.json file in the project root. "npm", as described by Brown (2014), acts as a package manager for Node packages. The package.json contains information, such as author, version and dependency with other Node packages for our project.

5.2. Class/File Hierarchy

Figure 8 is the structure of the source code in this project. Index.html contains HTML/CSS for showing the web page to the end users. App.js in Client folder is the client-side JavaScript code that handles user events, catches server responses and draws on the canvas. The outputs will be displayed to users in Index.html.

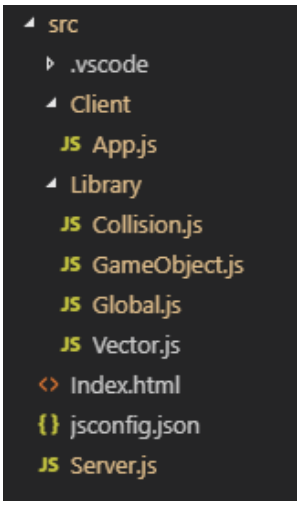


Figure 8. Files hierarchy

Inside the Library folder are the files that contain necessary classes/functions for Server.js. The usages of these files are listed below.

- **Vector.js** – It defines a 2D vector class that is used nearly everywhere in this project. All the movements, collisions, canvas coordinates etc. in the game rely on this class.
- **GameObject.js** – It contains classes for all objects of the game. Player information such as health amount, player position, team number, size of the player etc. are stored in the game object class.
- **Collision.js** – It contains functions required for game objects to perform collision detections.

- **Global.js** – It provides the ability to adjust the game-related parameters, such as shrinking rate for each character, canvas size, amount of public point objects, server update frequency etc.
- **Server.js** – This is where the server is. Server in this project is responsible for creating new players, processing client events, sending updates and recording game states etc.

5.3. Game menu and canvas

The game menu is a part of the Index.html that will be visible to users when they first get responses from the server (Figure 9). In the menu, users can choose to enter their usernames and select a preferred character or a specific team. These can also be left empty, as the character and the team will be assigned randomly by the system.

After clicking the start button, the menu will disappear, and the game canvas will show up immediately on the screen with the current game state being drawn (Figure 10). The canvas element is an HTML widget where 2D graphics (or 3D with WebGL) and Bitmap images can be drawn. User events, such as mouse movement and the mouse click will be registered on the canvas element.

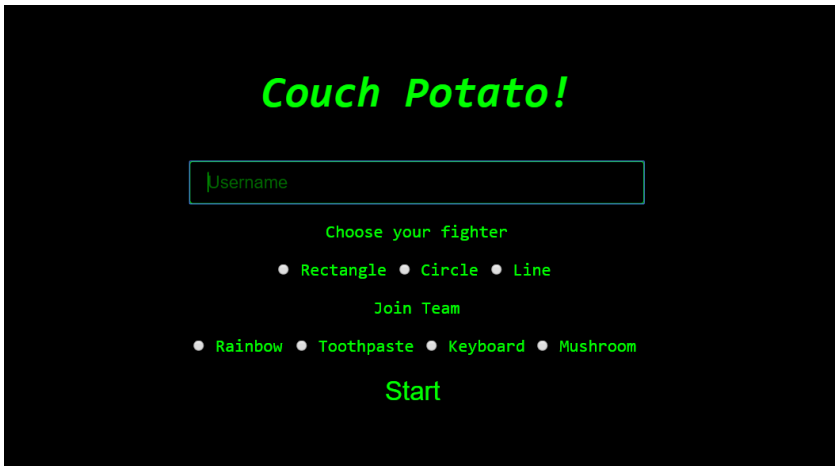


Figure 9. Game menu

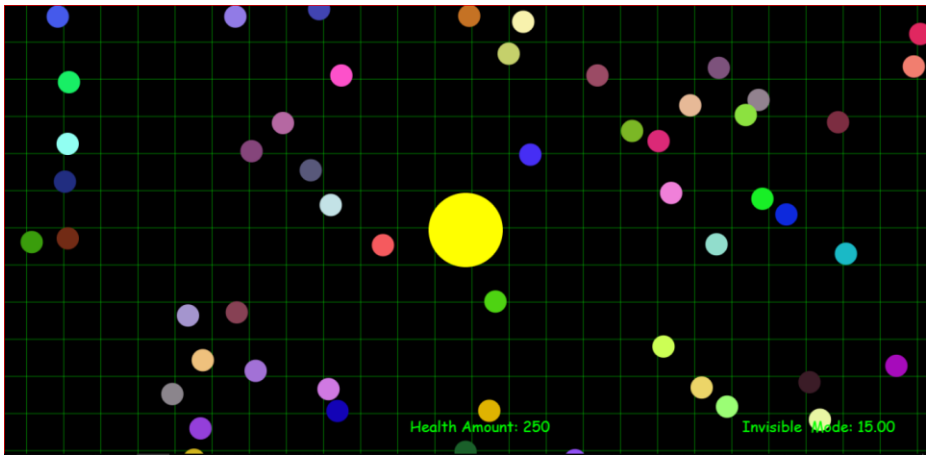


Figure 10. Gameplay

5.4. Player Movement

After the successful configuration and the menu setup, the first task of the project is to have a moveable character appearing on the screen. The initial task is done in a full-screen canvas at the user end:

- Set the canvas element in Index.html as the same size as user screen.
- Add mouse movement event listener to the canvas to get user mouse cursor position.
- Figure 11 shows how the player object can move to the expected position in a 2D plane within the same time span for x-axis and y-axis. By dividing the expected travelling distance, players are guaranteed to arrive the destination with a constant speed. In other words, for each second, the player will move by a constant distance regardless of how far the expected destination is. Constant variable c can be used to adjust the constant speed.

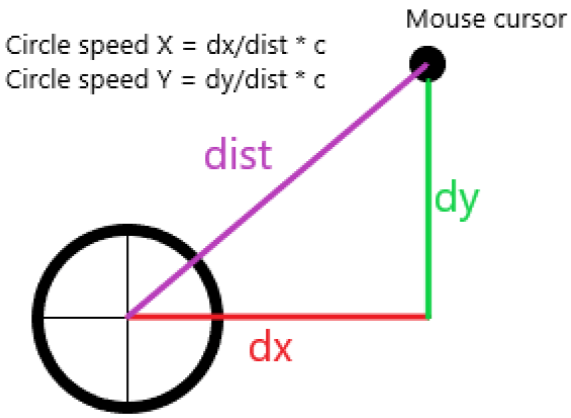


Figure 11. Player movement

As a multiplayer game, a canvas with the size of the user's screen is obviously not big enough for all the players to move around. To make players move in a canvas bigger than user screen, the visible part of the canvas for the player is moved as well. It means that after assigning the position for a player object, the player object is put in the middle of the user screen, and the coordinates for each corner are recorded by subtracting / adding half of the user screen height/width from player object. (E.g. The x position of the top left corner can be found by subtracting [screen width / 2] from player object x position, and y can be found by subtracting [screen height / 2] from player object y position.). Whenever a player is moving, the coordinates of four corners are also moved, so the player object will still stay in the middle of user screen.

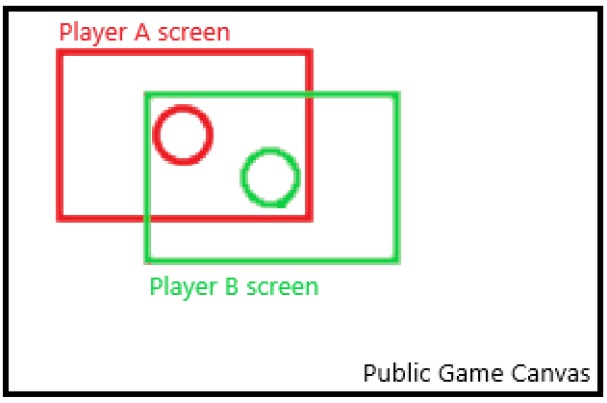


Figure 12. Players seen by each other

This technique creates an easy implementation of detecting whether there are other players or objects appearing in the current player's screen. As shown in Figure 12, player A is said to be visible to player B when the position of player object A is appeared to be inside the screen coordinates of player B, and vice versa.

5.5. Background Grids

Background grid is a public widget that is used to indicate the movement of player objects. When there is no object other than the player object itself inside the screen, the user needs to rely on the movement of the background grid to know the current speed of the player.

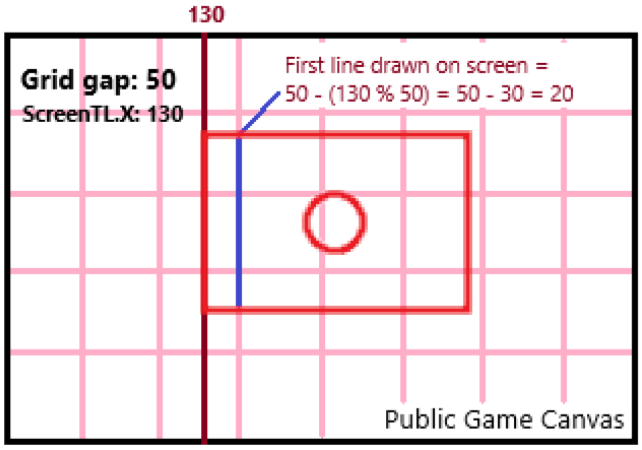


Figure 13. Assigning background grid

The background grid is assumed to be stuck on the canvas with a constant grid gap. The movement of the grid shown in the player screen can be found by calculating the first grid line position in the player screen point of view. The formulas used to find the first grid line on user screen are:

$$XFirstVerticalLineOnScreen = GridGap - (screenTopLeft.x \ \% \ GridGap)$$
$$YFirstParallelLineOnScreen = GridGap - (screenTopLeft.y \ \% \ GridGap)$$

Figure 13 is an example of finding out the x position of the first vertical line. The following vertical lines in the screen can be drawn by adding grid gaps to the current x position.

5.6. 2D primitives' collision detections

The collision detection allows an object in the game canvas to communicate with each other by touching the areas. In this project, the line segment, circle and rectangle are the primitives used to describe the shape of an object. The following section describes the methodologies used to achieve different kinds of collision detections.

5.6.1. Line segments intersection

The following methodology used to find the segments intersection is based on the material provided from the University of Glasgow (2000).

To find out whether two segments have an intersection, the orientation of three points needs to be introduced first. As described in Wikipedia (2018), "In mathematics, orientation is a geometric notion that allows one to say when a cycle goes around clockwise or counter-clockwise". If three points are treated as two vectors, the orientation can be determined by the determinant of them. A situation where two segments are collinear, i.e. determinant equals zero, also needs to be considered to find all possible orientations (Figure 14).

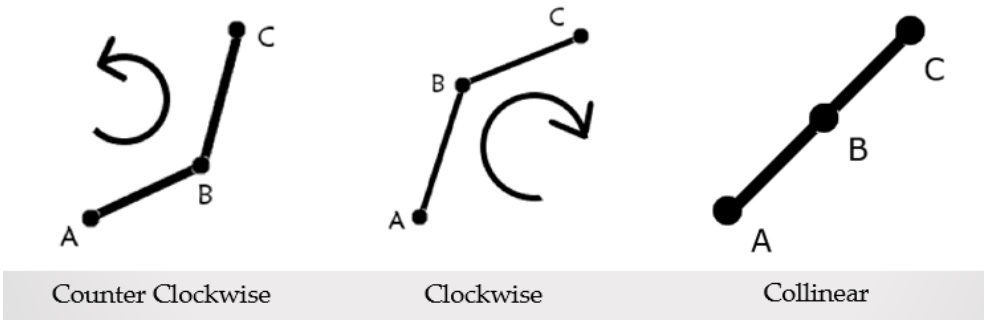


Figure 14. Possible orientations of three points

Next, it can be stated that two segments (p1, q1) and (p2, q2) intersect if and only if one of the following two conditions is verified (University of Glasgow, 2000):

- Condition 1:
  - (p1, q1, p2) and (p1, q1, q2) have different orientations
    AND
  - (p2, q2, p1) and (p2, q2, q1) have different orientations.
- Condition 2:
  - Given two segments are collinear:
    - One point of a segment lies inside another segment.

5.6.2. Line segment and rotatable rectangle intersection

To find out whether a line segment touches any point of a rectangle, the concept of the segments intersection in the previous section can be applied. Line segment and rectangle are intersected if the line segment intersects with one of the edges of the rectangle.

However, the problem comes in when a rectangle is rotatable. The actual rectangle presented in the client side is a rotatable rectangle, while in server side, only the centre position, width, height and current angle of the rectangle is recorded (Figure 15). The task of rotating points of either line segment or rectangle to simulate the client-side point of view is needed to check their intersection.

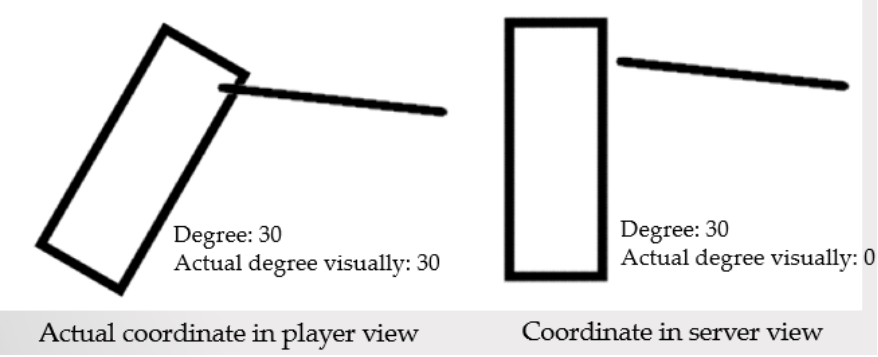


Figure 15. Rectangle in client and server views.

To be more computationally efficient, rotating line segments is chosen as it only requires rotating two endpoints instead of four corners for the rectangle.

The rotation matrix $R = \begin{pmatrix} \cos\boldsymbol{\theta} & -\sin\boldsymbol{\theta} \\ \sin\boldsymbol{\theta} & \cos\boldsymbol{\theta} \end{pmatrix}$ is applied here to get the rotation of the line segment around the centre of the rectangle.

By rotating the line segment, the occurrence of the intersection can then be computed by checking the intersection of the line segment and four edges of the rectangle.

5.6.3. Two rotatable rectangles collision detection

Like segment and rectangle intersection, rotating points of a rectangle is expected to get the actual presentation of the rectangles in client point of view. One rectangle needs to be rotated with its own angle and the negative angle of the opposite rectangle to be able to simulate the actual coordinates in the client screen (Figures 16, 17).

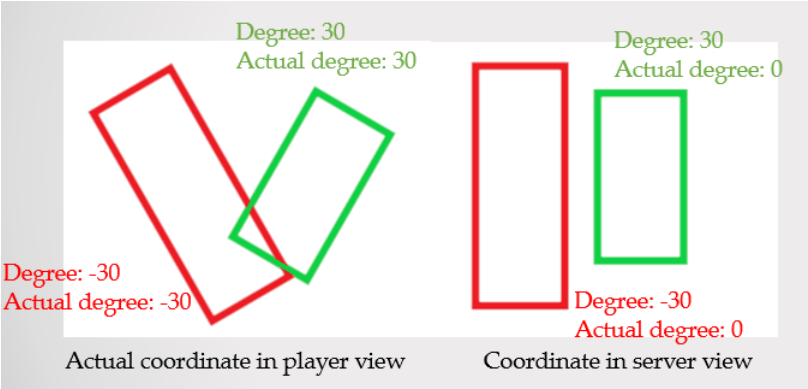


Figure 16. Rectangles in client and server views

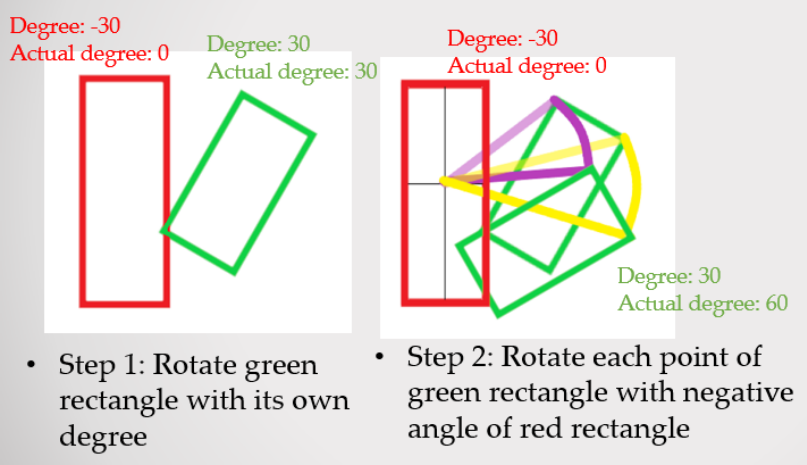


Figure 17. Rotate the green *rectangle by its angle and negative angle of the* red *rectangle to simulate client point of view*

5.6.4. Circles collision detection

Two circles collision detection is relatively simple than previously mentioned. Collision can be detected if the distance between the centres of two circles is smaller than the sum of the radiuses of two circles (Figure 18). There is no possible rotation for the circle object.

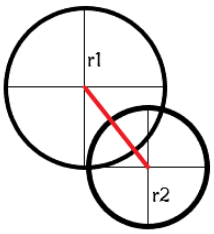


Figure 18. Circles collision detection

5.6.5. Circle and rotatable rectangle collision detection

Simulation of rotation needs to be performed to all collisions involving rotatable rectangles. To be computationally and logically efficient, a circle rotation with negative angle of the rectangle on the circle object is preferred to simulate the presentation in client aspect (Figure 19). The closest point on the rectangle from the rotated circle centre is then calculated to find the collision. Collision can be found by calculating whether distance from closest point on the rectangle to the rotated circle centre is smaller than the radius of the circle (Kano, 2012).

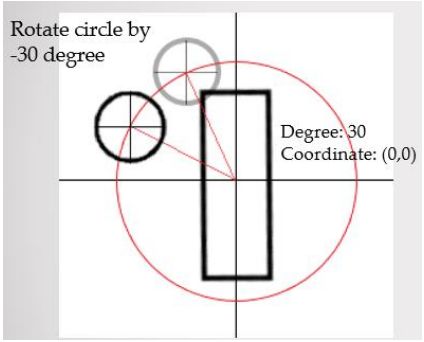


Figure 19. Rotate circle to simulate the client point of view.

5.6.6. Circle and line segment intersection

There are many approaches to get the intersection between circles and line segments. The method used here is based on the projection of the circle centre onto the line. Two cases needed to be considered here are that whether the projection point is inside the line segment.

If the projection is inside the line segment, then calculating whether the distance between the projection point and the circle centre is smaller than the radius of the circle centre can determine the occurrence of the intersection (Figure 20).

If the projection is not inside the line segment, calculating if an end-point of the line segment is located inside the circle can determine the occurrence of the intersection (Figure 21).

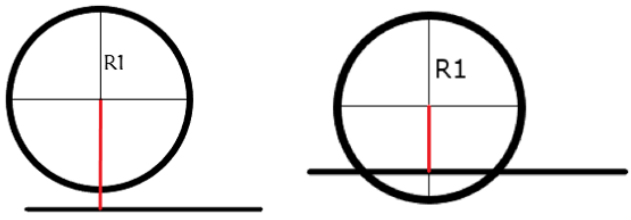


Figure 20. Projection point inside the line segment

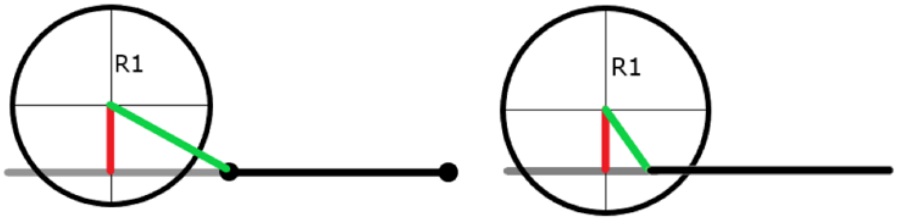


Figure 21. Projection point outside the line segment. Check if heads of the segment are inside the circle

5.7. 2D primitives' collisions and bounce effects

There are two types of bounce effects in this project. Circle and rectangle player objects can make public point objects (small circle) bounce away from them once the collision occurs. From the physic point of view, the bounce effect in this project is not perfectly correct. As one of the assumptions made for this project is that the conservation of momentum will not be obeyed during the collision between objects. When a player object attempts to push the public point object by applying a force, there will not be a negative force applied back to the player object. This allows the player object to move under a constant speed while having the ability to push any number of point objects away.

The following section demonstrates the process of making this effect.

5.7.1 Circle player and point object

As shown in Figure 22, the directional vector (red arrow) from the player centre pointing to point object centre is required to obtain the collision force applied from circle player (the bigger circle) to the point object. Next is to get the vector by subtracting the current speed of the point object from the player speed. The force can then be found by projecting this vector onto the directional vector, as shown by the orange line segment.

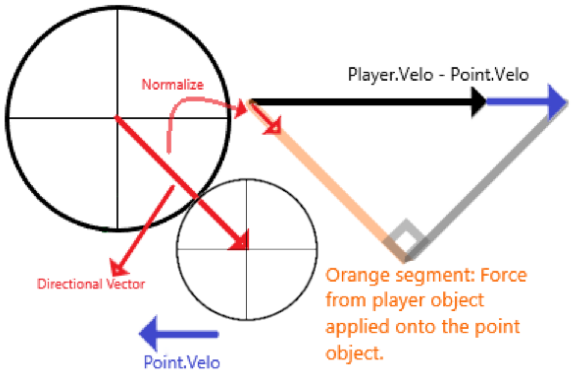


Figure 22. Find the force from circle player onto the point object

5.7.2. Rectangle player and point object

The bounce effect of the collision between the rectangle player object and the point object is similar to the circle player and the point object (Figure 23). The directional vector (red arrow) can be found by getting the orthogonal vector pointing to the public point object from the point of colliding (yellow dot). The only difference from the circle player is the rotatability of the rectangle player. The concept of angular velocity needs to be applied here by finding the distance from current colliding point (yellow dot) and that point at the previous update time (green dot). The final force applied onto the point object can be obtained by projecting the vector V ($V = Player.Velo + RotateVelo - Point.Velo$) onto the directional vector.

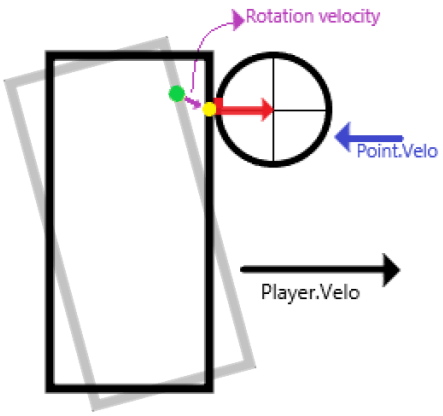


Figure 23. Find the force from rectangle player onto point object.

As said, the bounce off effect does not fit perfectly into the real physics. Assumptions need to be stated.

5.8. Network Communication

Network communication is crucial to allow this game to be multiplayer. Node.js is used in this project as a run-time environment that executes JavaScript code in server-side. To allow a simpler and more effective communication between client and server, the Express and Socket.IO frameworks are also used. The following section demonstrates how network communication is done for this project.

5.8.1 Building the server

The Command 'npm install -- save socket,io' has to be entered in command prompt at the project root directory for using Node.js framework Socket.IO. This is to install the framework to the project and to define the dependency in package.json file. Similar approach is needed for Express framework.

The code snippet in Figure 24 is an example of how the server of this project is built using Node.js with Socket.IO and Express frameworks.

```
var express = require('express');
var http = require('http');
var path = require('path');

var app = express();
var server = http.Server(app);

var io = require('socket.io')(server);

var port = process.env.PORT || 8081;

app.use('/Client', express.static(__dirname + '/Client'));

app.get('/', function(request, response) {
    response.sendFile(path.join(__dirname, '/Index.html'));
  });

server.listen(port, function() {
    console.log('Server running..');
    });
```

Figure 24. Code snippet for building the server

The '*app*' variable at line 5 in Figure 24 is declared as an Express top-level function. '*app*' then can be treated as an event handler for HTTP server (line 6).

What Express framework can do is not only handling the event for the server, but also defining the middleware and routing for the program. At line 14 of Figure 24, Index.html file will be served to the client who sent a GET request to the server, and similar to line 12, which serves the static JavaScript file contained in /Client directory that will be executed in client side.

The use of Socket.IO can be seen in line 8, whereas it requires the socket.IO module and then takes the HTTP server object as an argument. This is aimed to establish the connection of WebSocket on the provided HTTP server.

The last step of building the server is to allow the server to listen to a specific port, which can be seen in line 18 to 20. The server will now listen to either a port specified by the environment variable 'PORT' or 8081.

With the event-driven architecture, the server built with Node.js will wait for the request or the event sent from clients. Whenever a client sends a GET request by entering the expected URL (E.g. localhost:8081 when running server in local machine), the server will send a response with the Index.html which contains the game menu webpage to the client. The static JavaScript file 'App.js', will be served to the client side for handling user events and rendering the game state on the canvas.

5.8.2. Bidirectional communication between client and server through Socket.IO

After the client received App.js file, the corresponding code inside it will be executed in client-side, including line `var socket = io();` (Figure 25). As mentioned, Socket.IO contains both client-side and server-side libraries to enable the bi-directional communication. This line of code will try to connect to the Socket.IO in the server that hosted the current webpage from client-side.

```
1  //App.js
2  var socket = io();
```

Figure 26. Socket.IO in client side

Once the server received the request of connecting to the Socket.IO, code locating at Server.js in Figure 26 will be executed with a specific socket sent from a client. This is where we can do the corresponding actions for handling client events or sending events to a client.

```
1  //Server.js
2  io.on('connection', function(socket){
3    console.log("Player connected!");
4  });
```

Figure 25. Socket.IO in server side

The following section demonstrates a simple example of how the server handles the event sent from the client.

Figure 27 is the code executed in client-side (App.js) that tries to be registered as a new player in the server. Socket.IO uses an event-driven mechanism just like Node.js does. This line of code emits an event with information specified as parameters after the event name string 'newPLayer' to the server.

```
//Tell server to add new player.
socket.emit('newPlayer', type, window.innerWidth, window.innerHeight, Date.now());
```

Figure 26. New player event from client emitted to server

To handle the event, the server simply specifies the event name and an anonymous function that takes parameters from the client event as parameters in socket.on(…); inside the io.on('connection'…){…} block, as shown in the Figure 28.

This mechanism can also be used to send update from the server-side to the client-side.

```
io.on('connection', function(socket){
    //Handling new player event.
    socket.on('newPlayer', function(type, screenWidth, screenHeight, startTime){
        //Assign initial position and team color to the new player.
        let position = initPosition(actualTeam);
        let color = teamColor(actualTeam);

        //Assign new player
        if(type == "Circle"){
            currentPlayer = new GameObject.playerCir(socket.id, position.x, position.y, color,
        } else if(type == "Rectangle"){
            currentPlayer = new GameObject.playerRect(socket.id, position.x, position.y, color,
        } else if(type == "Line"){
            currentPlayer = new GameObject.playerLine(socket.id, position.x, position.y, color,
        }
        //Storing player and socket info
        if(currentPlayer != {}){ ···
        }
    });
});
```

Figure 27. Handler for the new player event from a client in Server.js

When a client leaves the page, the socket.disconnect() event will be sent to the server without explicit disconnection. To handle the event where a player reaches zero health amount, socket.disconnect() and socket.connect() can be called explicitly to reload the client page with a new socket. Table 1 specifies all the socket events used in this project.

Table 1. List of all socket events

Sent From	Event Name	Purpose	Timing
Client	connect	Connect to the server.	Client first loaded the page / Client reconnects to the server.
	disconnect	Disconnect from the server.	Client exits the page / Client reconnects to the server.
	checkTeam	Check the team availability.	When start button is clicked.
	newPlayer	Ask for a new player object from server.	*After the canvas is visible.
	lifeCounter	Record the total time of a player being alive.	*After the canvas is visible.

	mouseMove	Inform server the move of mouse position.	When mouse position is changed.
	mouseClick	Inform server the changes of pressed mouse keys.	When mouse key is clicked or released.
	Latency	Keep track of the latency of network connection.	Every 2 seconds.
Server	ReassignTeam	Client desired team not available.	After receiving checkTeam event.
	healthUpdate	To inform client the change of health amount.	When the health amount is changed.
	gameUpdate	Changed positions and states of visible objects.	60 times/second.
	dead	Inform the client the player has reached 0 health amount.	When health amount is zero.

***Note**: Due to the natural of Node.js asynchronous code execution, events like newPLayer and lifeCounter do not guarantee to have a fixed order of executing or arriving the server-side.

5.8.3. Publishing and high latency issue

Hosting the server on the Amazon Web Service (Elastic Beanstalk) is utilised at the final stage of the development. The content of this project can be found in the public URL: http://couchpotato.us-west-2.elasticbeanstalk.com/

However, the ongoing high latency issue as using public free-tier Amazon web service for hosting the content of this project cannot be solved. The latencies specified in Figure 29 are the response time from the server to the client. This is done by the following: the client first sends the current time and a callback function as an event to the server (Figure 30). Once the server received the event, it will call the callback function from the client-side with the timestamp of the event emitted by the client (Figure 31). The latency is obtained by getting the time gap of the current time and the time of the event emitted by the client.

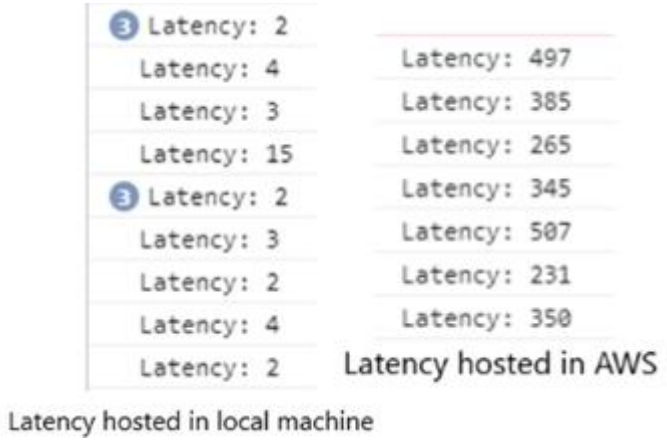


Figure 28.Latencies with the server in local machine and public-hosting service

```
//Client side
setInterval(function() {
    socket.emit('latency', Date.now(), function(startTime){
        var latency = Date.now() - startTime;
        console.log("Latency: " + latency);
    });
  }, 2000);
```

Figure 29. Latency checking event from the client

```
//Server side
socket.on('latency', function(startTime, callback){
    callback(startTime);
});
```

Figure 30. Handler for latency checking, the server will call the callback function with startTime parameter in client-side

There are two possible ways of solving this issue although the actual reason for the lengthy delay of network data transferring has not been found.

It might not be suitable for a web game that requires a relatively high frequency of updates from a server by using the free-tier web hosting service. As stated in Amazon (2018), “Elastic Beanstalk allows a low traffic application to run within the free tier “. Using a full version of web hosting service could be an approach to overcome this.

As mentioned, the high frequency of updates from a server is expected in this project. At the current stage, the server is expected to tell the next position of a player as well as all sorts of events happening on the game canvas, such as collisions, changed visible players in the screen for each client within an expected frequency of 60 times per second. The browser is expected to draw objects, to show information and to send users’ events to the server. By using this structure, it is highly possible to cause a noticeable delay for a client to move to the expected position, as it needs to wait for the next position information from the server. The following section describes an approach using client-side prediction and server-side reconciliation that could possibly improve this situation.

5.8.4.1. Client-side prediction and server-side reconciliation

Client-side prediction and server-side reconciliation are the techniques from Gambetta (2017) that could be potentially used to solve this issue. In this project, instead of moving the player object after telling the exact position by the server, client-side prediction can be achieved by moving the player to the expected position solely from client-side to avoid the delay time caused by network transferring. The prerequisite of this technique is that the game state needs to be deterministic. This means that given a set of inputs, the outputs are predictable. For example, when a client moves its mouse cursor position to the left of the player object (Inputs will be cursor and player object position), the expected position of the player object, left with a certain unit (output), can be predicted from client-side.

Server-side reconciliation is used to deal with the delay from client prediction and the actual game state sent from the server. With client-side prediction being done to avoid the heavy network transfers, the client still needs the server to tell the actual game state. As shown in

Figure 32 from Gambetta (2017), by the time the update from the server arrived at client-side for event one, the client has done the prediction for the event two. What server reconciliation does is to allow a prediction for the next move while updating the previous game state for the client.

A better implementation of this project is to update the player movement at the client-side rather than the server-side. The server tells clients whether they have collided with other objects based on client prediction. By doing this, issues where all players stuck at a position waiting for server to update could be potentially solved.

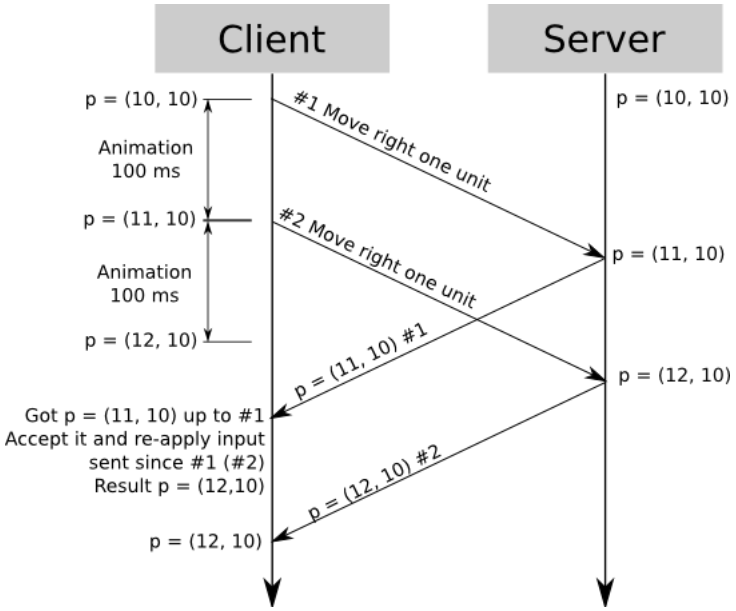
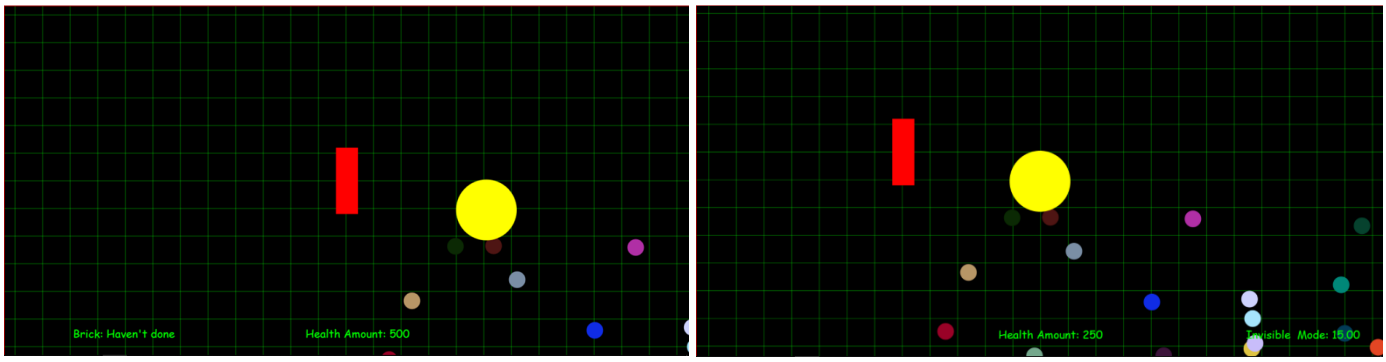


Figure 32. Client-side prediction and server-side reconciliation. (Gambetta, G. 2017)

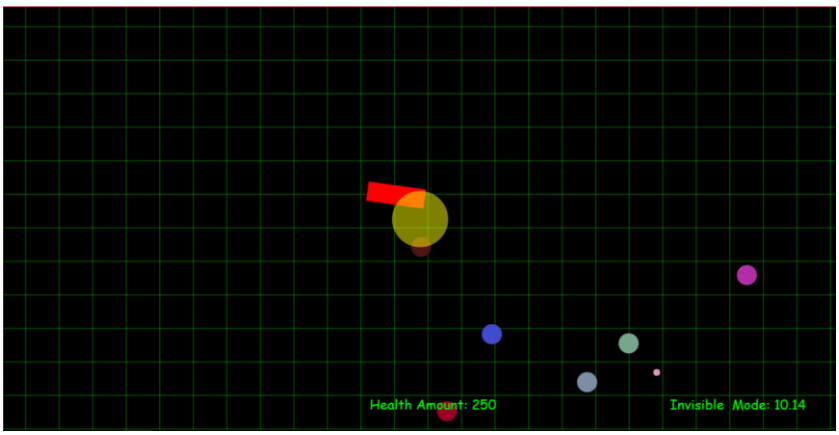
6. Game testing and log messages

This section shows screenshots of the actual gameplay and console log messages in server and client.

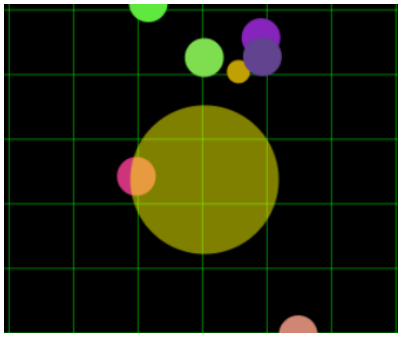
1. Yellow circle player and red rectangle player appeared in each other's screen:



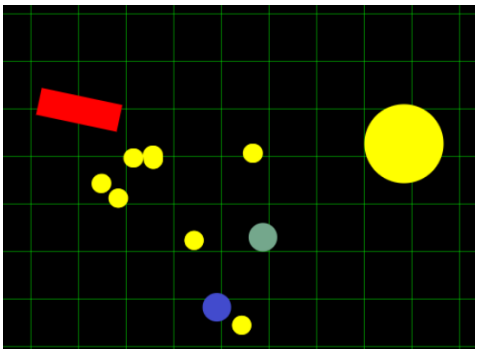
2. Yellow circle turns invisible to avoid damages from colliding with red rectangle (Invisible timer at the bottom right is reducing):



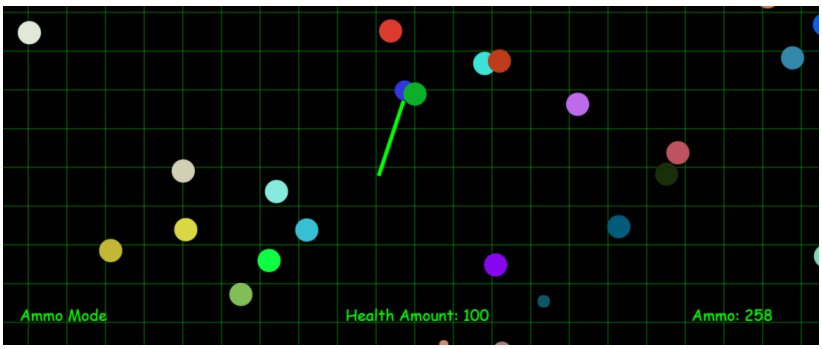
3. Yellow circle turns invisible to consume public point objects by covering its area:



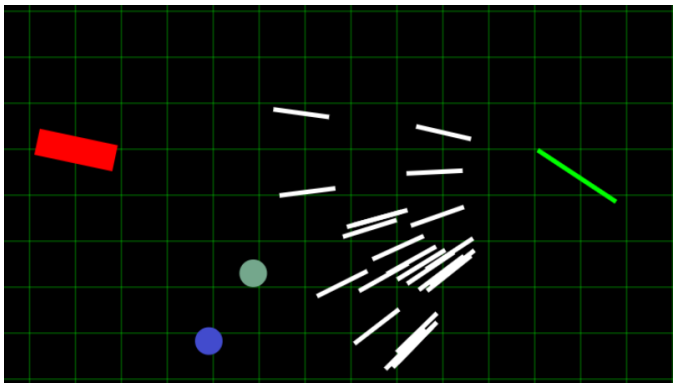
4. Yellow circle emits energy balls to avoid attack from red rectangle (Red rectangle will get damages if touching yellow energy balls):



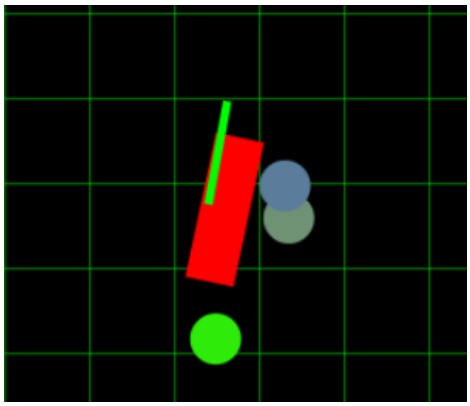
5. Green line player consumes point objects to add ammos (Ammo mode shown at the bottom left):



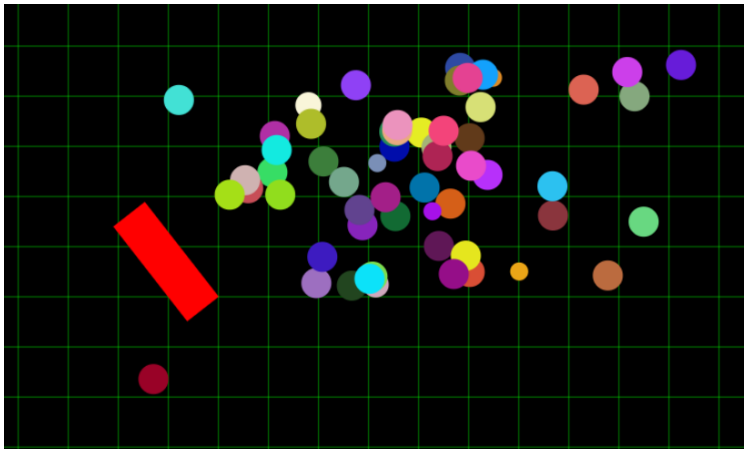
6. Green line player emits needles to attack red rectangle:



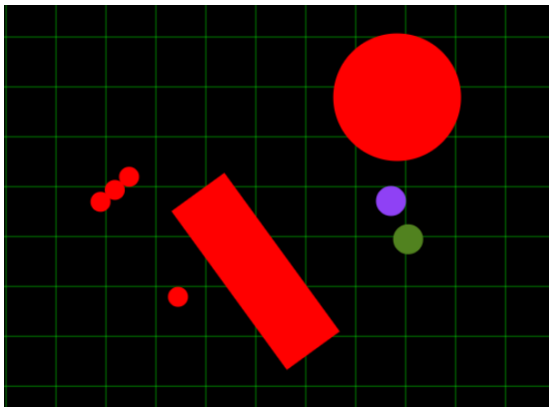
7. Red Rectangle attacks green line by colliding:



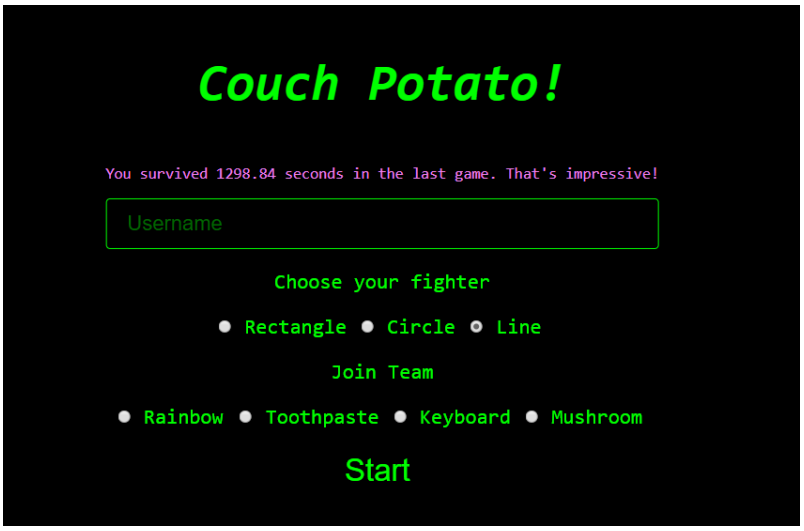
8. Red rectangle works on collecting public points for circle teammate:



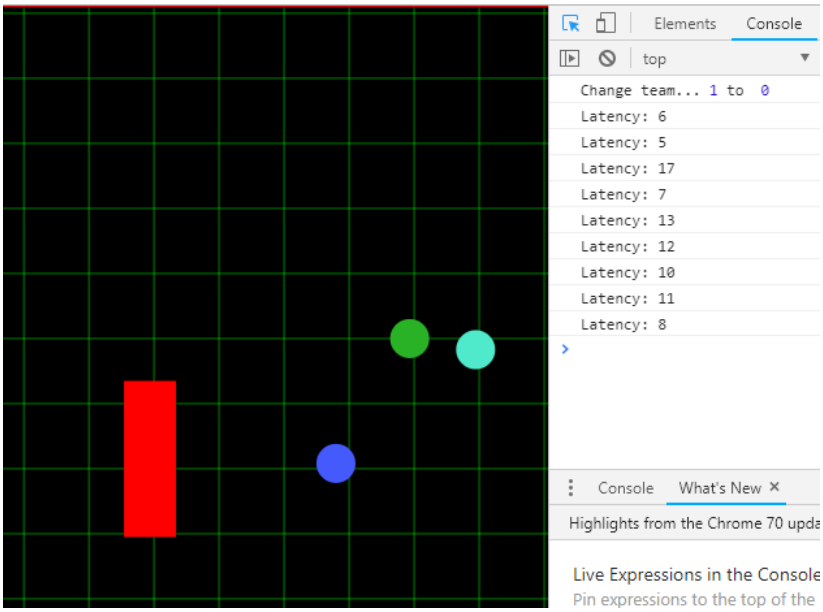
9. Red rectangle consumes energy balls (small red dots) from teammate circle to gain mass:



10. After player is eliminated, total time the player survived in the last game will be displayed in the menu (purple sentence):



11. Messages displayed in client console (right hand side). It includes change team message (for obeying team balancing rule) and latency messages emitted every two seconds from server.



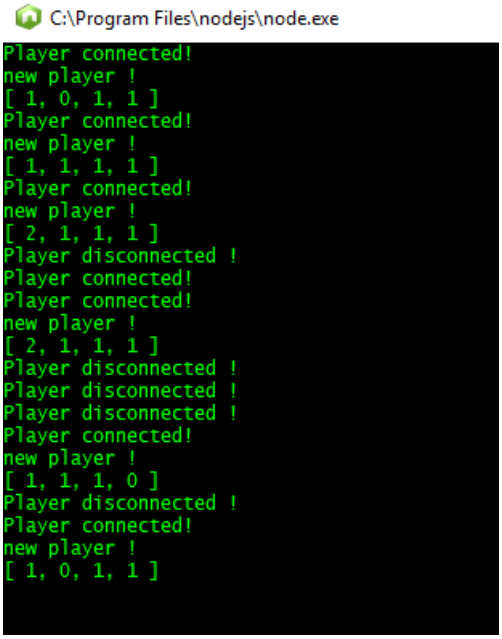
12. Log messages in server console:

Player connected: User connects to the URL.

New Player: Player clicks the start button and gets a new player object.

[1,0,1,1]: Shows the newest number of players in each team.

Player disconnected: User leaves the page or gets eliminated from the game.



7. Future Work

This project can be further developed to achieve a better gameplay environment and a more efficient computation. Here is a list of things that can be done in the future to produce a better result:

- Reduce the amount of data sent in the network to overcome the high latency issue.
- Remodify the code structure to reinforce code reuse.
- More efficient algorithms for collision detections.
- Handle the browser resize events.
- Functionalities of allowing multiple game arenas in one server and limiting the number of players in each arena for a fair gameplay.
- More users testing and adjustment of game-related parameters.
- Separation of Stylesheet from HTML file.
- Add more game related functionalities, create new available game characters/public objects.
- Bugs fixing.

8. Conclusion

Three primitive shapes are used to represent a total of six different objects in the game, where circle player, energy balls and public points are in circle shapes, the line player and needles are line segments, and the rectangle player is rectangle shape. Collision detections are important parts of the game to create a visually realistic and fairer gameplay, but they tend to be computationally expensive for the server to update under a high frequency.

The universal support in nearly all the modern browsers makes JavaScript a better option to build the front end of the browser-based game compared to languages like Java or C#. Node.js is used to build the back-end part of this project. With existing framework Express and Socket.IO, real-time and bidirectional communication between the server and the client can be easily achieved.

The problem occurs when the server is hosted by a public hosting service. When the latency is too high to allow high-frequency updates, client-side prediction and server-side reconciliation are the techniques that can be used to possibly fix this. Under the assumption that the game world is deterministic, which means that given a set of inputs, the outputs are predictable, client-side prediction can be used to reduce the server's stress by predicting the user movement based on user input. Server-side reconciliation allows the client predictions meanwhile provides the game state of the previous prediction. Overall, transferring overwhelming data across the network should be avoided as there is no guarantee that the internet can always be fast enough to transfer all the data under the expected frequency.

Reference

Amazon. (2018). AWS Elastic Beanstalk Pricing. Retrieved from: https://aws.amazon.com/elasticbeanstalk/pricing/

Bourke, P. (1988). Points, Lines, and Planes. Retrieved from: http://paulbourke.net/geometry/pointlineplane/

Brown, E. (2014). *Web Development with Node and Express*. O'Reilly. Retrieved from: http://www.vanmeegern.de/fileadmin/user_upload/PDF/Web_Development_with_Node_Express.pdf

ECMAScript 6 compatibility table. (2018). Retrieved from:

http://kangax.github.io/compat-table/es6/

Express. (2017). Express 4.x - API Reference. Retrieved from: https://expressjs.com/en/4x/api.html

Gaherwar, M. (2018). Blazor - Running C# On Browser Using Web Assembly. Retrieved from: https://www.c-sharpcorner.com/article/blazor-running-c-sharp-on-browser-using-web-assembly/

Gambetta, G. (2017). Client-Side Prediction and Server Reconciliation. Retrieved from: http://www.gabrielgambetta.com/client-side-prediction-server-reconciliation.html

Java. (2018). Java and Google Chrome Browser. Retrieved from: https://www.java.com/en/download/faq/chrome.xml

Kano, M. (2012). Circle and Rotated Rectangle Collision Detection. Retrieved from: http://www.migapro.com/circle-and-rotated-rectangle-collision-detection/

MDN Web Docs. (2018). WebAssembly Concepts. Retrieved from: https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts

Microsoft. (2018) .NET Game Development. Retrieved from: https://www.microsoft.com/net/apps/gaming

Rai, R. (2013). *Socket.IO Real-time Web Application Development*. Packt. Retrieved from: https://doc.lagout.org/programmation/tech_web/Socket.IO%20Real-time%20Web%20Application%20Development.pdf

Sen, A. (2017). Reflecting Upon on a Revolution in .IO Games. Retrieved from: https://www.engadget.com/2017/01/12/reflecting-upon-on-a-revolution-in-io-games/

Socket.IO. (2018). Socket.IO - Docs. Retrieved from: https://socket.io/docs/

Statista. (2018). Desktop internet browser market share 2018 | Statistic. Retrieved from: https://www.statista.com/statistics/544400/market-share-of-internet-browsers-desktop/

University of Glasgow. (2000). *GEOMETRIC ALGORITHMS*. Scotland. UK. Retrieved from: http://www.dcs.gla.ac.uk/~pat/52233/slides/Geometry1x1.pdf

Weisstein, Eric W. (2018). Vector Orientation. Wolfram MathWorld. Retrieved from: http://mathworld.wolfram.com/VectorOrientation.html

Wikipedia. (2018). JavaScript. Retrieved from: https://en.wikipedia.org/wiki/JavaScript

Wikipedia. (2018). Orientation (vector space). Retrieved from: https://en.wikipedia.org/wiki/Orientation_(vector_space)