# The Process of Making (.io) Web Games

Gurpreet Dhillon

**Abstract**—The purpose of this project is to bring the technical side of io games into light. The io games genre are accessible and free on the modern web, but the methods in which they are established are not usually delved into. I want to educate those who read my project on how gaming applications work, so that they can potentially make their own games.

**Index Terms**—Collision Detection, Multiplayer, K-Dimensional Trees, io games

## 1 Introduction

THIS project, I would like to discuss the technical side of how io games are made. This will include an introduction on io games and will discuss topics, such as client and server-side programming, client-side prediction, networking, and collision detection using the k-d tree algorithm. For the k-d tree algorithm specifically, I will talk about its time complexity and speed compared to using a regular collision detection approach. Overall, I will go into depth of each topic explaining its use and purpose in gaming applications. In the end, I will explain how all these topics come together to form multiplayer games.

## 2 What are io games?

io games are free multiplayer browser games in which you don't need an account or install anything. The .io domain is not related to the gaming genre and just represents the country code domain for the British Indian Ocean. The first and arguably the most popular io game, Agar.io used .io as their domain of choice, and subsequently other multiplayer web games followed, creating the io game genre.

## 3 Fundamental Tools

Before going into each technical aspect, I will discuss the tools that make web development convenient and easy. First, we are going to need Node.js which functionally is the runtime environment for the server. After successfully setting up Node.js, we can use default package manager (NPM) to install Express and Socket.io. Express is a library that will serve as our web framework for Node.js, essentially powering our web application. Socket.io is another library which will be used for networking, transferring data between the client and server. Lastly, we need to figure out how to render each client, but luckily we can use HTML5's built in feature, Canvas. This allows for dynamically rendering 2D images.

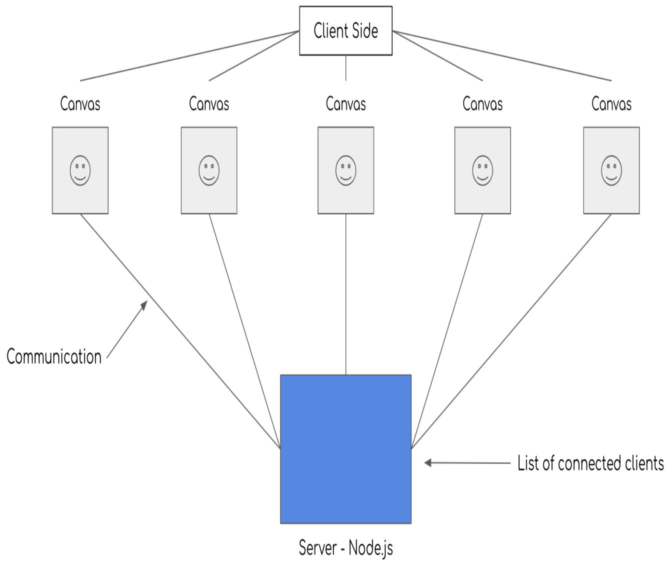

Fig. 1: A Basic Overview of Multiplayer Web Games: Client-Server Model

## 4 HIGH LEVEL OVERVIEW

An example of a working multiplayer game can be seen in Figure 1. Here, we can observe that on the client-side, there are five connected clients where each has its own Canvas. On the server-side, we have the Node.js server that is keeping track of every client connected to it. The reason of keeping a record of the connected clients is to allow every client to render every other client in the game, resulting in the multiplayer aspect. The bi-directional communication between the clients and the server allows for the server to achieve this.

## 5 NETWORKING

We will be using socket.io for the networking aspect between web clients and the server. This library is divided into two parts: client-side library for the browser and the server-side library for Node.js. These two parts have nearly the same API in which we can either listen for or emit information on both the client and server side, hence the ”bi-directional” communication.

## 6 CLIENT-SIDE PROGRAMMING

This section will cover the required components that we want to receive and provide to clients that connect to our game. The first thing we need to do on the client-side is setup a promise, an object which enacts the eventual completion of an asynchronous operation and returns the result. For example, a client connecting to the main menu of a multiplayer web game enacts a promise where it listens for specific actions. The client clicking ”play” invokes this promise, allowing the user to enter the game. Using a promise makes entry-points to games easy to establish since it all depends on what the client does, instead of having the game run synchronously in the background. Now that we discussed how to create an asynchronous entry-point, we can talk about the components that are invoked once in the game.

### 6.1 Capturing Input

One of the most important features in multiplayer games is keeping track of the player’s location. We can accomplish this by taking in user input, specifically capturing mouse movements. This information will be sent to the server along with other information about the client. Then the server sends this information back to the same client and the other clients that could be in the same game as well, so we can render everyone in the game on each player’s device.

### 6.2 Rendering via Canvas

This is an interesting part of client-side programming where we have to render the multiplayer game, but only render what the client can see on their screen to minimize cpu usage. Using HTML5 Canvas, we can set the size of what a client can see on their screen and proceed with rendering the game at their location and size of screen. This leads to a design decision that is up to the programmer: creating a map, boundaries, stateless features and objects, players, etc. Once the programmer considers and makes their decision choices, we can move to arguably the most difficult part, rendering the player relative to everything else in the game.

Luckily, HTML5 Canvas makes this process easier for the programmer, as it provides a translate(x, y) method and we only need to consider the following parameters: size of the map, size of the canvas window, and the player’s location. The translate method is used to move the canvas and its origin to a different position, and this position can be calculated by finding the offset of the player’s location relative to the map size.

The last thing we need to do is invoke rendering at the same number of FPS (frames per second) that is specified by the server, since we will be receiving updates at the same rate, in our case we will be drawing the ”frames” at this rate. We can use Javascript’s setInterval() method, which executes a specified function multiple times at set time intervals specified in milliseconds (1000ms = 1second) [1].

## 6.3 Client-side Prediction

Client-side prediction is the method in which we can hide the unintended affects of high latency connections. Sean A. Pfeifer summarizes client-side prediction as ”a method that attempts to perform the client’s movement locally and just assume, temporarily, that the server will accept and acknowledge the client commands directly” [2]. We would picture this as an ideal situation where there are no delays in receiving and sending updates, but in reality, it’s a much different story. We will discuss this problem further and propose few temporary solutions.

### *6.3.1 Frame Rate Problems*

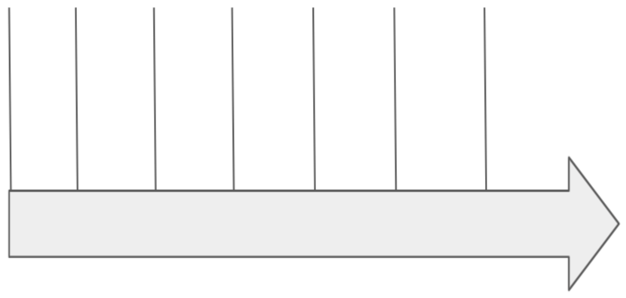

Fig. 2: Ideal updates where the bars represent the updates and the arrow represents the server’s run-time [3]

Figure 2 represents an ideal situation where the client is receiving game updates every X amount of milliseconds consistently from the server. This is what we want to replicate as best as possible for every client to give them the best experience, but there are many issues, such as unpredictable lag.

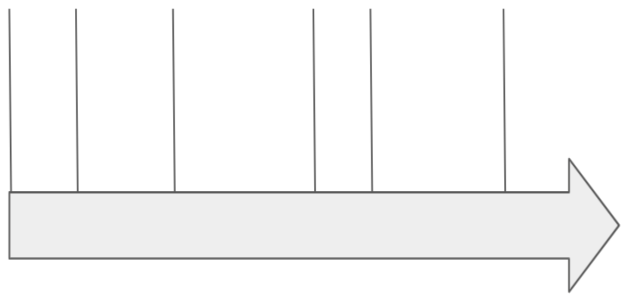

Fig. 3: Real updates where the bars represent the updates and the arrow represents the server’s run-time [3]

In this situation, we can see inconsistency in the updates that the client is receiving from the server. To understand why its a problem, Pfeifer says the following, ”when there is lag in the connection, the client uses the last command acknowledged by the server and attempts to simulate using the most recent data from the server” [2]. This means that if the client is expecting an update to arrive an X amount of milliseconds and it ends up arriving late, the client will freeze for an X amount of extra milliseconds because its rendering the previous update. This can cause a very bad experience for players in the game and we need to find ways to mitigate these affects.

### *6.3.2 Addressing Frame Rate*

The first method we can use is creating an arbitrarily sized buffer to tolerate unpredictable lag.

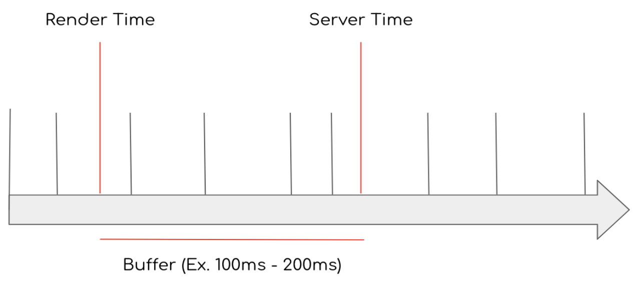

Fig. 4: Buffer used to separate the render and server time [3]

Using an X millisecond buffer can make it easy to tolerate unpredictable lag. For example, if the server is at 200 milliseconds, the state that is rendered on the client will be what the server's state was at time 100 milliseconds if we used a 100 millisecond buffer. Pfeifer describes the concept of a buffer as lag compensation and states ”Lag compensation can be thought of as taking a step back in time, on the server, and looking at the state of the world at the exact instant that the user performed some action. So, this technique doesn't perform client-side actions, but rather deals with the state of objects on the server. Note that we are completely moving the state of the object back in time, and not simply the location” [2]. This buffer or ”lag compensation” creates somewhat smooth, consistent gameplay which is more tolerable than unpredictable lag, but a drawback is constant input lag of the X millisecond buffer. Overall, players can enjoy the game without seeming to experience latency.

Another method we can use on top of this is the concept of linear interpolation. Pfeifer says ”interpolation can be viewed as always moving objects somewhat in the past with respect to the last valid position received for the object. In this method, you buffer data in the client, and display the data after a certain period of time. This method will help with the visual smoothness of other objects in the game-world” [2]. Due to the render delay we discussed previously, we will usually have at least an update ahead of the current client render time. This means that we can slowly move or ”interpolate” towards a certain position in between updates, causing the smooth movements, instead of the flashing or ”teleporting” from freezing that can be caused by unwanted lag.

## 7 SERVER-SIDE PROGRAMMING

Within the server, we first initiate the web server with express and configure socket.io to listen for connections from clients. Afterwards, we can construct our game into a class specifying the details and attributes we need. This will consist of all the connected clients, along with their sockets. Furthermore, if our game is a shooter game, then we would need a list of every active bullet in the game or if our game includes dynamic objects, then we would need a list of those too. Lastly, we can send game updates to our clients at any rate (Ex. 60 FPS) and these updates will include what the client needs to render which includes every other player, their locations, and game objects. We can accomplish this by using the setInterval() method discussed earlier [1]. Now, we can discuss how we can deliver the multiplayer aspect of our game with collision detection.

### 7.1 Collision detection

Collision detection is the method in which we detect the intersection of two or more objects. Graham Morgan, the author of the journal Scalable collision detection for massively multiplayer online games, describes the detection of objects as ”A state update message indicating that local object, say obj1, of node N1 has moved may result in a receiving node, say N2, identifying a collision between N2's local object, say obj2, and obj1. However, before N2 enacts collision response both nodes (N1 and N2) must agree that a collision has or has not taken place. If agreement is reached that a collision has occurred then N1 and N2 must enact a suitable collision response (e.g., change direction of movement) for their local objects” [4]. This is how the detection between two objects work and the following is a visual representation.

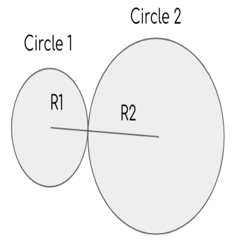

Fig. 5: Objects collide if the distance between each of their centers is less than or equal to the sum of both their radii

This is the minimalist approach for detecting collision, though this can become more complex for massively multiplayer online games, in which objects become deformable and that we will have to rely not only on the radius, but many other attributes too.

Now the real question is how we can apply collision detection. Morgan says "A simple algorithm for determining collisions would be to compare all objects with all other objects: a brute force approach to collision detection. Such an approach is an O($n^2$) problem and satisfying collision detection requirements for large numbers of objects may not be possible in real-time" [4]. An O($n^2$) is the idea of using two loops, where we check each object against each other, except itself. This algorithm won't be efficient and very slow because we have to go through every single object. There are algorithms that can perform much better than O($n^2$) as we will see in the next section.

### 7.2 K-d tree method

The K-d tree data structure is a very efficient method to check for collision detection / nearest objects without the need to check every possible candidate. To summarize, a K-d tree is a K dimensional binary tree where each node has K attributes. For every level of the tree, we are only focusing on one dimension of the points. We want to go level by level, alternating the dimension we are looking at and when we are at a given level for a corresponding dimension, we act as a Binary Search Tree. The article Fast neighbor search by using revised k-d tree states "K-d tree is a typical partition tree, which is widely used in many applications, and has various variants, such as optimized k-d trees, FRS, and buffer k-d trees" [5]. There are many ways to optimize a k-d tree, since there are situations where the k-d tree doesn't reach the nearest neighbor, so I will be discussing an optimized k-d tree, mainly a backtracking algorithm to find the nearest neighbor.

We can explain the nearest neighbor search as "Let $\xi$ be a cell of node s, and t be a point, there is a case between $\xi$ and range (t , e), as follows: range (t , e) covers (includes) $\xi$: all points within cell $\xi$ are all neighbors of t...Check whether there could be any points that can become current best on the other side of the splitting plane, by the way of judging whether the splitting hyperplane intersect with range (t , e) where e = dist (t, s)" [5]. What the article is saying here is that we can compare the range between the node and the result with the range between the node and the parent node's subsection. If the latter is less than the former, we can backtrack one step and visit the other children node section. We can explain this visually as follows:

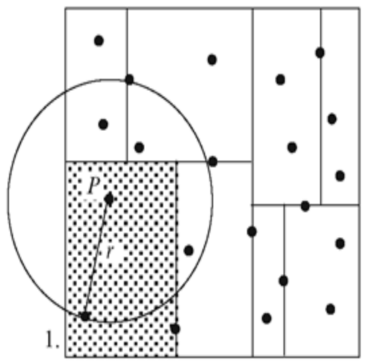

Fig. 6: Regular K-D tree solution [6]

Though the k-d tree is known for its O($nlog_n$) time complexity, sometimes it can result in the wrong solution, as seen in Figure 6. The current distance found didn't result in the nearest neighbor. "Since the hyperplanes are all axis-aligned, the algorithm simply makes a comparison to check whether the distance between the splitting coordinate of the search point and current node is lesser than the distance from the search point to the current best" [5]. This is how we can backtrack to check the node on the other side of the splitting plane if the case is true.

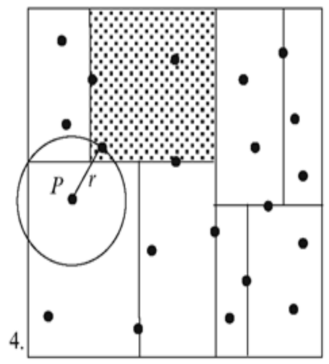

Fig. 7: Optimized K-D tree solution [6]

"If the hypersphere intersects with the plane, there could be nearer points on the other side of the plane, so the algorithm has to check the other branch to find them, following the same way as the entire search. Otherwise, ignore the whole branch on the other side of the current node." [5]. Applying this technique, we can see the distance between node P and parent node's subsection is smaller, so we can visit its child node on the other side. This backtracking algorithm will result in finding the nearest neighbor, shown in Figure 7.

Now that we know how to implement k-d trees and collision detection, we are able to check for collision detection in O($nlog_n$) time instead of O($n^2$) which results in a much faster and efficient program.

## 8 Discussion on Applications

For the application section, I would first like to discuss how these technologies are applied to popular io games, then discuss how they can also apply to an application that is outside the realm of the io genre.

The popular io game I would like to discuss here is Agar.io. This web application was the first ever io game created, which was created by Matheus Valadares back in 2015. Agar.io is a multiplayer survival web game where the objective is to become the biggest player in the server and this can be accomplished by either eating pellets (food) scattered around the server or other players (if they are smaller than you). The concept of the game is very straightforward, but the process of implementing each component and scaling the web application to thousands of players is much more complex.

For the purposes of this discussion, we will talk about how the methologies we discussed earlier in this report can be applied to Agar.io. The client-side programming and implementation can be directly applied to most io games, and Agar.io is definitely one of them. We will still need to create an asynchronous client entrypoint, capture mouse and touch movements as input from the user, network with sockets to transmit and received communications, render 2D graphics, and update the current state of the player based on server feedback. For server-side programming is where I believe the most interesting aspects come into play because now we have full control on how our game will be played.

Whenever you join or create a game in Agar.io, you notice how there are preloaded pellets and other objects that the player can interact with. This is all setup by the server, before the game is rendered. In the section of server-side programming, we discussed how we can initialize the backbone of the game with a class structure containing its features as attributes and this is how we can exactly create these preloaded objects. Therefore, we would first need to create a server entrypoint, initialize our preloaded objects, which in this case of Agar.io would be the pellets, and listen for a client's connection and input. Once a client connects, we can send them information, containing all the players and pellets attributes, and start rendering everything that would appear on the client's screen.

Furthermore into server-side programming, the collision detection and k-d tree combination can be very useful for the situations when a player collides with a pellet or player and if there are thousands of pellets and players in the same game. For Agar.io, we can use the same minimalist collision detection approach discussed earlier, since the only information we need for detection is the radius of the two objects colliding. To calculate the result of the collision, we can simply use the radius to find the total area of both objects, then find the new radius from the total area and set it to the winning object. Now that we know how to check and calculate collision detection for Agar.io, we need to make comparisons for a player relative to every other consumable object in the game.

Before, we traverse the k-d tree and start making comparisons, we need to first setup the k-d tree. We can have two separate k-d trees for the in-game objects like pellets and another for every player in the game or one k-d tree that stores every consumable object, but this is more of a design choice that is up to the programmer. Also, the ways in which we update and re-balance a k-d tree, for instance when a pellet is consumed and thrown to some other arbitrary position in the map or when a player dies/joins, is out of the scope of this report, since these are also design choices and there are many ways to implement this in terms of replicating Agar.io.

Once we setup our k-d tree, we can create a function to traverse the tree where the current player's location and radius is the input. We will be alternating between the player's x and y coordinates to traverse the k-d tree. When we find our nearest neighbor, we can start checking and calculating the potential collision detection between the player's radius and the nearest node's radius. I say potential because we will be sending and receiving updates from clients at hopefully 60 frames per second and of course the player won't always be colliding for objects. In the case there is a frame where a collision occurs, the method described above should be able to traverse to the exact node that the player is colliding with and make those updates.

Now that we have a proper client-server model implemented, the next step would be to host your program on the .io domain and this can be accomplished by using a cloud platform like Google Cloud. Though, this step is optional for the purposes of this report, but it is another very interesting topic, since that's how most io games, like Agar.io, are able to host and scale to thousands of players across the world. Going through the whole process, from programming the game to hosting it on the cloud, is definitely a useful skill that I'd recommend achieving. Overall, these are how the implementations can be adapted to Agar.io, but these techniques are not only limited to io games.

Another application of the technologies described that is not only limited to the io realm are multiplayer shooter games where a big component of them is the bullet (collision) detection. Imagine that every bullet fired had an ID associated with the player that shot the bullet. Now, we can simply check if there is any collision between that bullet and another player using the implementations discussed above. "In most of the shoot'em up genre games, brute force is used for collision detection. This is not a problem if the game is run on a computer but it might cause lagging issues when it is on a mobile platform. Even if the computer is able to support, it is always encouraged if there is a way to reduce the computation speed" [7]. The article states that using the method of brute force (linear search) is not the best idea in terms of performance.

Luckily, we can use k-d trees to compute collision detection for nearby bullets without having to traverse to every single active bullet in the multiplayer game. "As for other algorithm that is commonly used for optimizing collision between groups is spatial partitioning. Spatial partitioning is basically a process of subdividing a space into a number of cells and places the object primitives into the cell" [7]. The spatial partitioning refers to dividing parent sections in subsections within a binary tree, hence the k-d tree. "This greatly reduces the complex data structures and the information can be generated very fast and efficiently" [7]. Overall, we can reduce high CPU usage and time by combining collision detection with the k-d tree algorithm to lead to better, and more responsive gameplay in multiplayer applications, especially when dealing with hundreds in players in the same active session.

## 9 Conclusion

To conclude, we successfully were able to discuss each key aspect of multiplayer web games in a high level overview. We examined the purpose of each component in gaming applications and tied it all together towards the end. We discussed how to implement a client-server model, by describing the programming needed on each side of this system. Then we talked about how to improve collision detection by using the optimized k-d tree algorithm. Lastly, we applied what we learned to how io games work, namely Agar.io, and other applications such as shooter games.

Now, we can see each component's practicality in multiplayer games, but that doesn't mean that its limited to just gaming. For example, concealing the negative effects of connections between a client and a server is very important for any type of application that doesn't want lag. Also, the k-d tree is practical for storing information efficiently in any scenario, like searching for medical records of each patient of a hospital. Overall, the methodologies described can be applied to many applications.

## Acknowledgements

I would like to thank Algokodabra (Youtube), The Coding Train (Youtube), Victor Zhou [3], and Sean A. Pfeifer [2] for helping me better understand the basics of collision detection, k-d trees, and .io fundamentals in order to replicate Agar.io and before delving into the research of the topics discussed in this report.

## References

[1] BitDegree, "Tips on javascript setinterval method: Learn to set interval," Aug 2017. [Online]. Available: https://www.bitdegree.org/learn/javascript-setinterval

[2] S. A. Pfeifer, "Real-time multiplayer gaming: Keeping everyone on the same page." [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.94.6528&rep=rep1&type=pdf

[3] V. Zhou, "How to build a multiplayer (.io) web game, part 1." [Online]. Available: https://victorzhou.com/blog/build-an-io-game-part-1/

[4] G. Morgan and K. Storey, "Scalable collision detection for massively multiplayer online games," in *19th International Conference on Advanced Information Networking and Applications (AINA'05) Volume 1 (AINA papers)*, vol. 1, 2005, pp. 873–878 vol.1.

[5] Y. Chen, L. Zhou, Y. Tang, J. P. Singh, N. Bouguila, C. Wang, H. Wang, and J. Du, "Fast neighbor search by using revised k-d tree," *Information Sciences*, vol. 472, pp. 145–162, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0020025518307126

[6] "Collision detection (advanced methods in computer graphics) part 6." [Online]. Available: http://what-when-how.com/advanced-methods-in-computer-graphics/collision-detection-advanced-methods-in-computer-graphics-part-6/

[7] K. W. Ng, Y. W. Yeap, Y. H. Tan, and K. I. Ghauth, "Collision detection optimization on mobile device for shoot'em up game," in *2012 International Conference on Computer Information Science (ICCIS)*, vol. 1, 2012, pp. 464–468.