Novaglide - an enhanced air hockey

Tomáš Barhoň - Matěj Schrödl

ČVUT–FIT

schromat@cvut.cz, 58461337@fsv.cuni.cz

6. ledna 2024

1 Introduction

In our project, we wanted to create a real-time fast-paced multiplayer game with a high skill ceiling. The game's goal is to score more goals than the opponent in a limited time. If no winner can be established after the regular time the **tiebreaker** starts and the first player to score wins. The game is essentially air hockey where players can cast two spells **hook** and **dash** with **cooldowns** of 20 and 10 seconds respectively. The **hook** shoots a harpoon from the player in the direction of the mouse until it reaches the game border. After that, the player is **pulled** towards that wall and cannot move until he reaches it, the only possibility how to cancel the **hook**, which is a very quick movement that follows the position of the mouse called **dash**. Our goal was to create both **1v1** and **2v2 matches** and design an **elo**, **ranking system** and **matchmaking system** for the game. We have also set some long-term goals such as adding music to the game, enhancing the graphics and deploying the game to be accessible publicly.

2 Networking

Making a real-time game proposes quite a difficult challenge concerning networking. Low latency is crucial otherwise the game becomes unplayable. The communication between the client and server is performed through a TCP socket which is slower than UDP but provides an easier implementation for us as we minimize the size of the packets to a bare minimum. These choices might be reconsidered in the future when the game gets deployed publicly.

We designed a simple communication API with a JSON-like format. The structure of a packet is as follows:

{"time":datetime.datetime.now(),
"sender":"server/client", "flag":flag, "data":[data]}

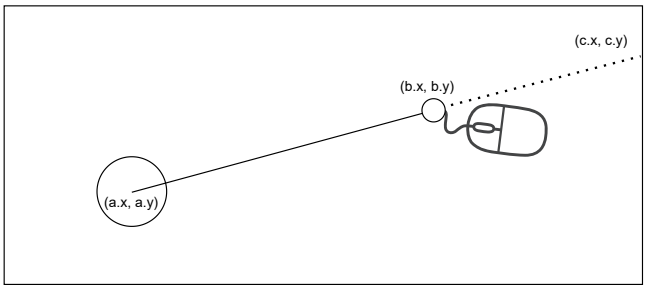
The packets are then decoded according to their type specified in the "flag"parameter by the receiver. Another aspect is that the server only acts as a responder meaning that it cannot send data to the clients without them asking for it this means that if the clients want to receive a permanent stream of data from the server they need to send their requests.

Our main idea was to prevent cheating. The game needs to be validated or entirely played by the server. We decided to take the latter approach which means that the whole match loop is handled by a single thread on the server. Threading is an essential part of the design of the server. There are two types of threads: threaded client and threaded match the reason for that is that we need the communication with the client and each separate match to be handled at the same time to increase scalability. While the match loop starts the clients in the match get notified that they are in a game and start their loop in which they are sending their keyboard and mouse inputs to the server and get a representation of the game state in response which is then drawn on their screen. This ensures that the state of the game is the same for all players and that players cannot send corrupted info as the server updates the game only, when certain inputs on the keyboards are sent. After the game all clients receive statistics from the game and the server deletes the game instance afterwards. A simple networking diagram can be seen at the end.

3 Methods

There were two interesting challenges in the design of the game loop. The first one was ensuring that throughout the existence of the player class, it stays inside the border of the game this is implemented as an invariant method that ensures that any movement is disallowed when the future state means breaking the invariant rule.

The other was designing the hook to make it the most realistic possible. We aimed to send the hook in the direction of the mouse but when calculating the speed vector as the difference between point B and A and normalizing it the hook would always



Obrázek 1: Vector rectangle intersection

stop at the location of the mouse (the location at the time of casting). That is why we employed a different approach and calculated the speed vector as the normalized difference between C and A. But to calculate point C one needs to use parametric equations of a line to find the point where the vector intersects with the rectangle. Let's set point $TL(0,0)$ meaning the top-left part of the screen and point $BR(1280,720)$ meaning the bottom-right which is our screen resolution. We also know that the coordinates of the point $C(c.x, c.y)$ satisfy from the parametric equation the following:

$$C(a.x + t(b.x - a.x), a.y + t(b.y - a.y))$$

Now we just need to find the correct t which has to be positive. We follow with a simple approach.

$$\begin{aligned} a.x + t_1(b.x - a.x) &= TL.x \\ a.x + t_2(b.x - a.x) &= BR.x \\ a.y + t_3(b.y - a.y) &= TL.y \\ a.y + t_4(b.y - a.y) &= BR.y \end{aligned}$$

As there is always just one unknown we can solve all of the four equations separately for t.

$$\begin{aligned} t_1 &= \frac{TL.x - a.x}{(b.x - a.x)} \\ t_2 &= \frac{BR.x - a.x}{(b.x - a.x)} \\ t_3 &= \frac{TL.y - a.y}{(b.y - a.y)} \\ t_4 &= \frac{BR.y - a.y}{(b.y - a.y)} \end{aligned}$$

After plugging in the values of TL and BR we get:

$$\begin{aligned} t_1 &= \frac{-a.x}{(b.x - a.x)} \\ t_2 &= \frac{1280 - a.x}{(b.x - a.x)} \\ t_3 &= \frac{-a.y}{(b.y - a.y)} \\ t_4 &= \frac{720 - a.y}{(b.y - a.y)} \end{aligned}$$

Now we just take the smallest positive t and from the equation above find the point C. This turned out to be very effective and made the animation of the hook as realistic as we wanted. The rope is still tied to the player but the front point of the harpoon is flying in the direction of the cast.

4 Results

From the standpoint of implementation, we believe we have managed most of what we aimed for. The only two things we failed to do in time were **2v2 matches** and **matchmaking system** based on the elo. The first could be added in a matter of days as it is no different from 1v1 but we did not have enough time to do it. The matchmaking system based on Elo would need many tests and simulations where we could generate a bunch of players and queue them on some mock server that would simulate the matchmaking. We decided to stick to completely intuitive matchmaking that connects players whenever there is enough of them to create a game and the elo system is designed to reflect even massive differences between their skill levels (represented by elo). But the networking and playability of the game are working better than expected and our approach turned out to be working well. The overall design turned out pretty nicely except for the plain colors of the players which could be in the future changed to some custom skins. The settings menu is already prepared to do this in the future.

5 Conclusion

Overall we believe that we have created quite an interesting project, especially with the aspect of real-time networking which is not that commonly implemented in the community cause of its problematic nature. After some final polishes and with a custom soundtrack the game could be in the future deployed publicly and the project officially published to present an approach to how real-time networking can be done in pygame. Publishing would need to be accompanied by some security measures as hashing passwords or preventing SQL injections in the login screen.

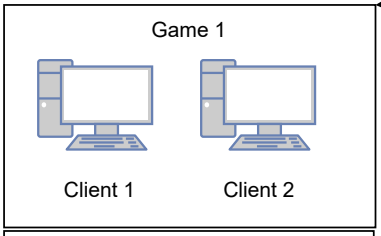
Reference

[1] unknown. Intersection point of a vector or its extension with surrounding rectangle. online, 2022. [cit. 2023–01–06] `https://stackoverflow.com/questions/74021571/intersection-point-of-a-vector-or-its-extension-with-surrounding-rectangle`.

Database query

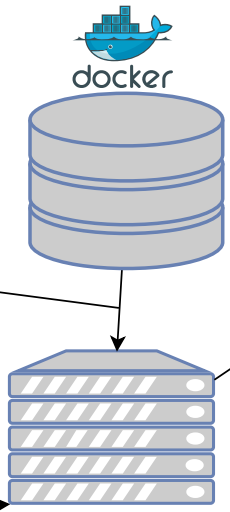
The clients send a requests to the server whenever they need to **log-in**, **see top ranked players**, or **see their match history** and the server queries the database and returns the result inside a packet.

Similar logic applies to game queue, client notifies the server that he wants to play and the server adds him to the set of queued players. Whenever enough players are ready the server sends those players a notification and conencts them into a game. After the game is finished clients receive match statistics and the result.



Threading

The communication with the clients is performed inside seperate threads and each game also runs inside a different thread. This design enables better scalability instead of iterating over the games sequentially.



Game state drawing - clients

The only work that is done by clients is streaming the inputs to the server and in response receiving representation of the game state. The clients than only draw the state of the game from the information provided by the server on each screen.

Game state networking - server

All the game logic is handled by the server and is completely independent from the requests sent by clients. Each game is played in a single thread.

The clients stream permanently the state of their mouse and keyboard and provide inputs to the game. These states are assigned to a variable from which the server takes the inputs at each tick. This design prevents the game speed to be dependent on the number of packets sent by clients.

Online players		
Client 3	Client 4	Client 5
Client 6		

Solo lobby		
Client 7	Client 8	Client 9
Client 10		