# Rock Paper Scissors: A Multiplayer Realtime Strategy Game

Thanik Sitthichoksakulchai
ID: 1804397

## Introduction

Rock Paper Scissors is a 2D multiplayer real-time strategy game based on Rock Paper Scissors with a strategy aspect. The game has maximum players of four. A player can train, upgrade the units and use them to fight with other players. There are three types of units which are rock, paper, and scissors. The game was built using Unity engine with C# Socket for the networking part.

## Game Mechanic

At the start of the game, each player has 3 buildings that can train a unit and upgrade itself. Each building has its own type which is rock, paper, and scissors. Each building has 1000 health points and each unit has 100 health points. The player who can kill all the enemy's units and destroy all enemy's buildings will win. Each player has a limited number of units which is 50.

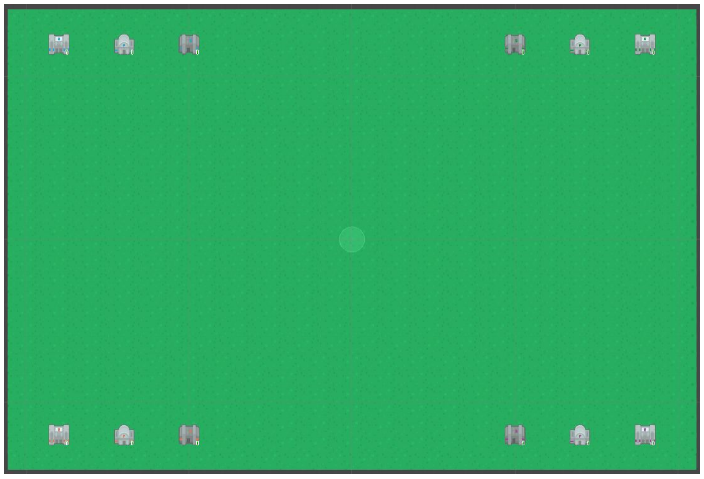
*Figure 1 Overall map of the game*

*Figure 2 Picture of rock, paper, scissors units*

### Unit Attack

Units can make damage to any enemy's units in attack range. The damage depends on the type of unit that attacks to and attacks from.

| Attack from \ Attack to | Rock | Paper | Scissors |
|---|---|---|---|
| Rock | 10 | 5 | 35 |
| Paper | 35 | 10 | 5 |
| Scissors | 5 | 35 | 10 |

### Unit Attack Cooldown

Units have cooldown time on each attack and it depends on the level of units.

| **Level** | **Cooldown in seconds** |
|---|---|
| 1 | 1 |
| 2 | 0.75 |
| 3 | 0.5 |
| 4 | 0.25 |
| 5 | 0.15 |

## Networking Architecture

The game uses Server-Client architecture for networking. The server controls all the game logic and keeps sending game state to clients. The client sends the player input to the server then the server will send the result back as the updated game state. Because the client can only display the game state from the server, the client can't manipulate the game state directly. This can prevent client-side cheating.

## Protocols

The game uses UDP for the transport-layer protocol. As the game objects' state keeps changing over time, older game states might be outdated when it arrived at the client. With this constraint, TCP is not practical because useless old states packets would be resent if those packets were lost in transmitting. While playing, the server will send the game state changes 20 times per second.

## Network Message Data Structure

The network message consists of two segments: an enum value indicating the type of message and a payload of data. The message is serialized in binary format using MessagePack library.

### Type of network message and its payload

- JOIN -- used for a client connecting to server
  - Player's nickname
- WELCOME -- used for the server sending assigned client's ID back after receiving joining message from a client
  - Client's ID
- ERROR -- used for the server sending an error message to a client
  - Error message
- LOBBYDATA -- used for the server sending game lobby data to a client
  - All clients' data
  - Game start time
- READY -- used for a client sending ready state while the in-game lobby
- UNREADY -- used for a client sending unready state while the in-game lobby
- DISCONNECT -- used for a client disconnecting from the server
- SHUTDOWN -- used for the server telling clients that it's going to shut down
- CLIENTTIME -- used for a client sending its current game time to the server
  - Current game time
- SERVERTIME -- used for the server sending its current game time back after receiving client's game time
  - Client's game time that received
  - Server's current game time
- SYNCTIME -- used for the server sending its current game time for client's game time synchronization
  - Server's current game time
- UPDATE -- used for the server sending all changes in units and buildings data
  - Game time
  - All changes from the last sent update
  - All players' stats like a number of units for each player.
  - Max object ID
- FULLUPDATE -- used for the server sending all units and buildings state data in case of desynchronization
  - Game time
  - All units and buildings state data
  - All players' stats like a number of units for each player.
  - Max object ID
- UNITSACTIONS -- used for the client sending player's action on units and buildings
- GAMEOVER -- used for the server sending the game result to clients

- Client ID of winning player

## API

The game uses C# Asynchronous Socket API. This API is more suitable because the networking code should not block the game engine thread (Unity).

## Integration

Networking-related code for server and client is in a separate class which is NetworkServerManager, and NetworkClientManager.

### Server-side code

Every fixed 20 times per second timestep, the game will gather all game objects' states and compare them with last sent states. Then it calls the socket API to send these data to all clients.

### Client-side code

Because game logic functions can't be called within the network callback function, the client needs a buffer for storing data from the server. Then process them in the next game frame using boolean to trigger processing.

## Prediction

The game uses linear interpolation for predicting the position of units as the units can move only in the X and Y axis. The predicted position can be calculated by using 2 latest unit's positions and timestamps.

$$v = \frac{p_1 - p_0}{t_1 - t_0}$$

$$p = p_0 + (v \times (t_c - t_0))$$

$v$ is the velocity of the unit.
$p_0$ is the latest position of the unit.
$p_1$ is the older latest position of the unit.
$t_0$ is a timestamp of the latest position of the unit.
$t_1$ is a timestamp of the older latest position of the unit.
$p$ is predicted position.
$t_c$ is current game time.

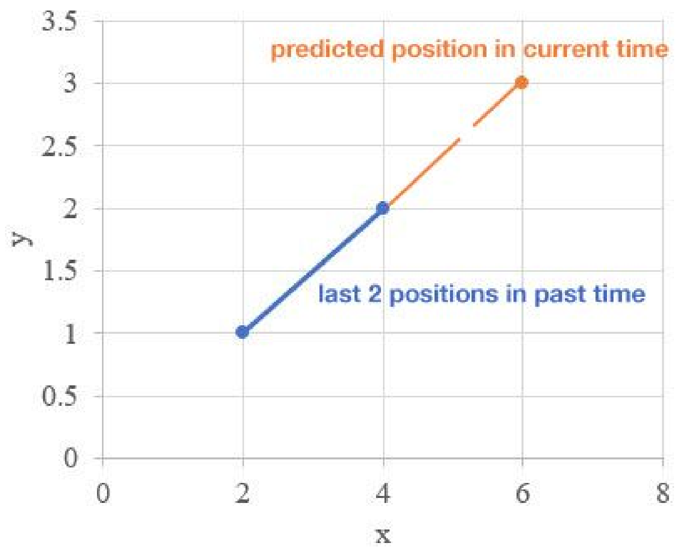

*Figure 3 A graph showing the position of a game object in different time*

## Testing

The game was tested on virtual machine and host machine and Clumsy (a tool to simulate a bad networking) was also used in testing. Tests were conducted using these settings.

- Best network connection
    - Clumsy was turned off.
- Typical internet network connection
    - Latency: 50ms
    - Drop: 0%
- Cross-country internet network connection
    - Latency: 150ms
    - Drop: 0%
- Poor network connection
    - Latency: 350ms
    - Drop: 10%

### Observations and problems from testing

- Lagging network can cause a delayed result in the player making actions on game objects. For the real-time strategy game, it’s acceptable if actions are received properly on the server-side.
- Dropped packets can cause game state desynchronization and some actions from player didn’t arrive to the server. The player needs to repeat his action again.
- The units wiggle around when they are moving to an empty space. This problem is related to Unity’s pathfinding library as they are trying to fight each other to get to that position and it can make units’ position prediction looks bad.

- Time synchronization between server and clients fails sometimes. This can cause units' position jump around the game map.
- Disconnect/Shutdown message isn't sent when the game was closing sometimes.

### Further improvements

- In bad network scenario (150ms+ latency), the delay result from player's action becomes noticeable. Using animation and sound can hide this latency for example, playing acknowledge sound after player makes an action on unit, making unit moves before the action arrives to the server then correct game state result later.
- Some network packets like player's action should be resent if the server didn't get.
- Find a better solution for multiple objects pathfinding.

## Conclusion

To summarize, the game can demonstrate on how simple Server-Client networking for game works and how to handle latency in real time networking application. But it still needs some improvements and polishing.