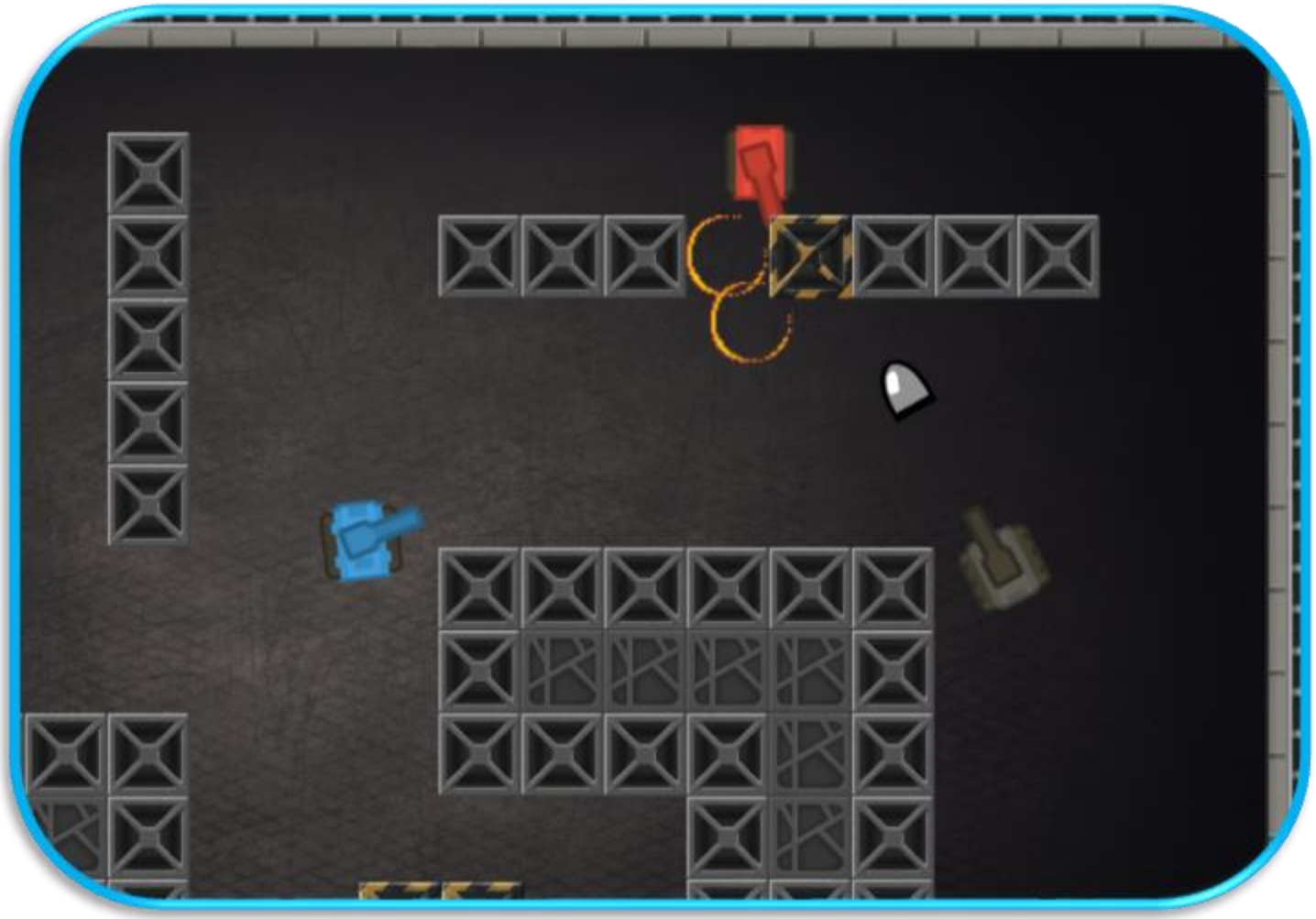
CMP303
Presentation



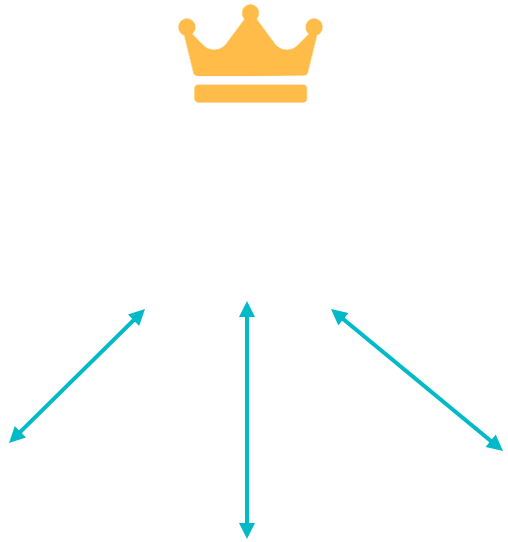
The Game

- Mini Tank Shooter
- 2-4 Players
- Keyboard and Mouse Input
- Destructible Blocks
- Tank Shells ricochet
- First to 3 Round Wins
- Round is complete when only one tank remains

Network Architecture

Hybrid Client/Server

- Client can act as a server allowing for other clients to connect.
- The acting server relays packets to all the clients.



Why Hybrid Client/Server?

- Type of game to play with friends, no need for dedicated servers.
- The users will create servers when needed. Demand will always be met.
- Easier to keep clients synced.

Transport-Layer Protocol(s)

Lobby

Transmission Control Protocol (TCP)

Gameplay

User Datagram Protocol (UDP)

Lobby - TCP

Why use TCP for the lobby?

- Speed of packets being sent doesn't matter.
- Reliability is more important.
- Don't need to worry about packet not being received.

Gameplay - UDP

Why use UDP for the gameplay?

- Latency is more important during gameplay.
- The sacrifice of reliability for speed is worth it.
- Can work around reliability issues.

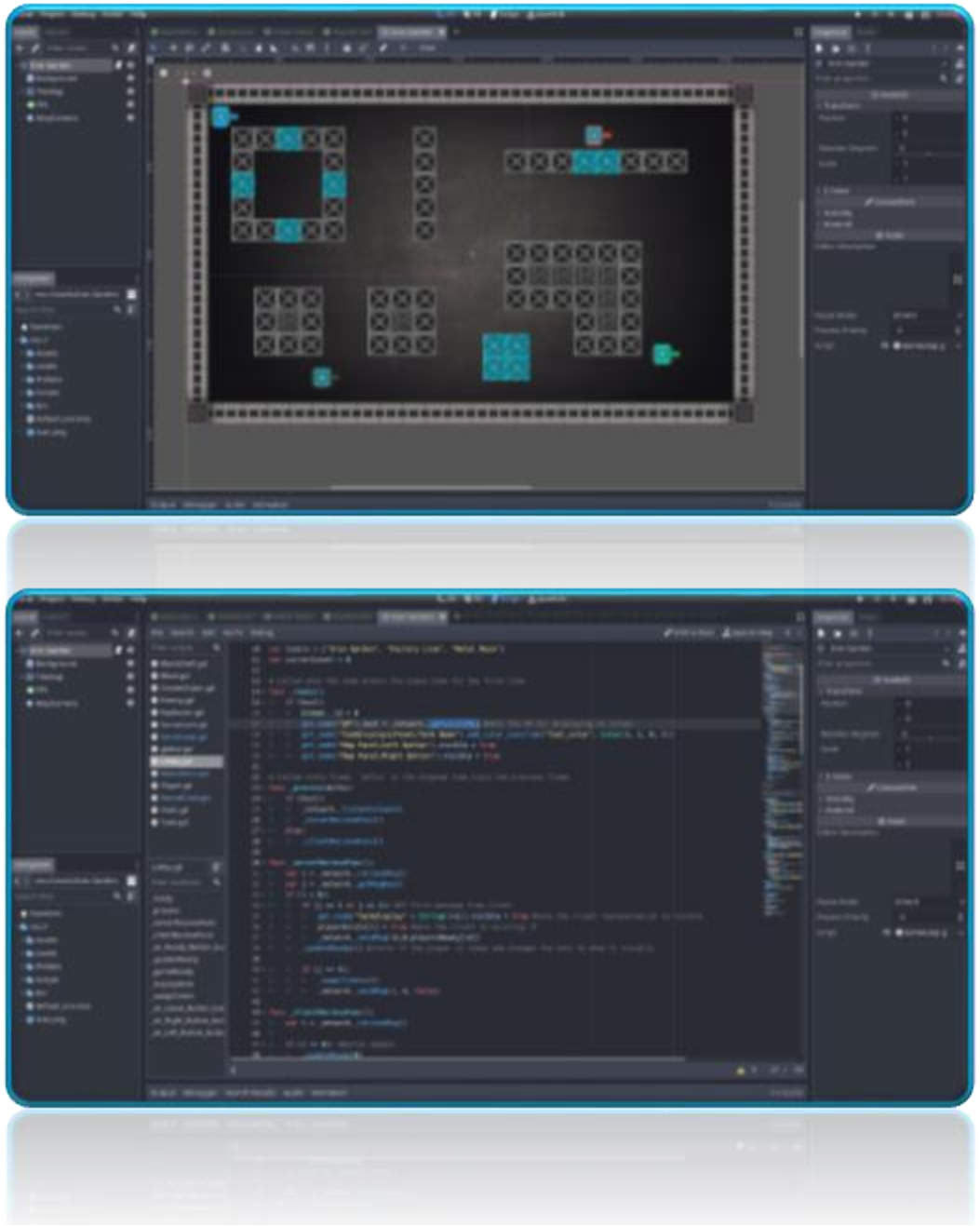
Network API

- WinSock


Networking
Code

Godot & GDNative

- The game was made in the Godot Game Engine.
- Godot uses its own programming language GDScript.
- However, the networking code is implemented in C++ using GDNative.



```
func _on_CreateButton_pressed(): #Sets the client to host and loads the
    Global._lobbyNetwork = load("res://Scripts/LobbyServer.gdns").new()
func _on_JoinButton_pressed(): #Sets the client not to host
    Global._lobbyNetwork = load("res://Scripts/LobbyClient.gdns").new()
```

```
_network._sendMsg(id,0,playersReady[id])
```

Integrating Networking Code

```
void LobbyServer::_sendMsg(int id, int key, bool pReady)
{
    LobbyMessage message;
    message.playerID = id;
    message.key = key;
    message.playerReady = pReady;

    // send to each client
    for (auto it = clientSockets.begin(); it != clientSockets.end(); )  // note no ++it here
    {
        SOCKET conn = *it;
        int count = send(conn, (const char*)&message, sizeof(LobbyMessage), 0);
        if (count != sizeof(LobbyMessage))
        {
            std::cout << "sendto failed\n";
        }
        ++it;
    }
}
```

- Both the server and client use the same program, they just load different scripts.
- This method allows for calling networking functions that use C++

Lobby Server Set Up

- Creates and bind a TCP socket for listening for connections.

```
// Create a TCP socket used for listening for connections
sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
```

- WSAEvents are used to tell if events occur.

```
//Create Listen event
ListenEvent = WSACreateEvent();
WSAEventSelect(sock, ListenEvent, FD_ACCEPT | FD_CLOSE);
```

- Accept and Close events are used on the listen socket.

Lobby Server Listen

- Upon a new connection:
  - If there is less than 3 clients
    - Add client socket to array
    - Create events for closing, reading, and writing

```
returnVal = WSAWaitForMultipleEvents(1, &ListenEvent, false, 0, false);

if ((returnVal != WSA_WAIT_TIMEOUT) && (returnVal != WSA_WAIT_FAILED)) {
```

```
if (NetworkEvents.lNetworkEvents & FD_ACCEPT)
```

```
SOCKET clientSock = accept(sock, NULL, NULL);
if (clientSockets.size() == 3)
{
    closesocket(clientSock);
    return;
}
else
{
    clientSockets.push_back(clientSock);
```

```
clientEvents[clientSockets.size()-1] = WSACreateEvent();
WSAEventSelect(clientSockets[clientSockets.size() -1],
    clientEvents[clientSockets.size()-1], FD_CLOSE | FD_READ | FD_WRITE);
```

Lobby Client Set Up

1. Creates TCP socket
2. Connect the socket to the server's address.

```
// Create a TCP socket.
sock = socket(AF_INET, SOCK_STREAM, 0);
```

```
// Connect the socket to the server.
connect(sock, (const sockaddr*)&serverAddr, sizeof serverAddr);
```

Game Server Set up

1. Creates UDP socket
2. Bind Socket to same address lobby socket was

```
// Create a UDP socket.
sock = socket(AF_INET, SOCK_DGRAM, 0);
```

```
void GameServer::_bindSocket(String address, int cCount)
{
    clientCount = cCount;
    serverIP = address;

    sockaddr_in serverAddr;
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_addr.s_addr = inet_addr(address.alloc_c_string());
    serverAddr.sin_port = htons(SERVERPORT);

    if (bind(sock, (const sockaddr*)&serverAddr, sizeof(serverAddr)) != 0)
```

Game Client Set up

1. Creates UDP socket
2. Connect to Server socket with same IP as lobby socket was.

```
// Create a UDP socket.
sock = socket(AF_INET, SOCK_DGRAM, 0);
```

```
void GameClient::_setServerIP(String address)
{
    serverIP = address;
    servAddr.sin_family = AF_INET;
    servAddr.sin_port = htons(SERVERPORT);
    servAddr.sin_addr.s_addr = inet_addr(address.alloc_c_string());
```

How messages affect game

Switch statements and If statements are used to produce different outcomes.

In the lobby, ID and Key are used to differentiate different messages.

```
#ID 0 is server ready state update
#ID 1-3 is
#ID 4 is Host Disconnect
#ID 5 is Game Start
#ID 6 is Map Update

#--- Key 1 is a new client has connect (send a new message saying you exist to the new client)
#--- Key 2 is you are the new client and are told your id
#--- Key 3 same as 1, but doesn't send a message (this is to stop infinite sending back and fort
#--- Key 4 is client has disconnected so removed them from screen.
```

The game uses a similar method, without the keys*

Tick Rate and sending messages

- If the interval between messages (0.1 seconds/10 tps) has elapsed
- And the tank is alive

A message is sent with the tanks own id, position, body rotation, barrel rotation, and local client time.

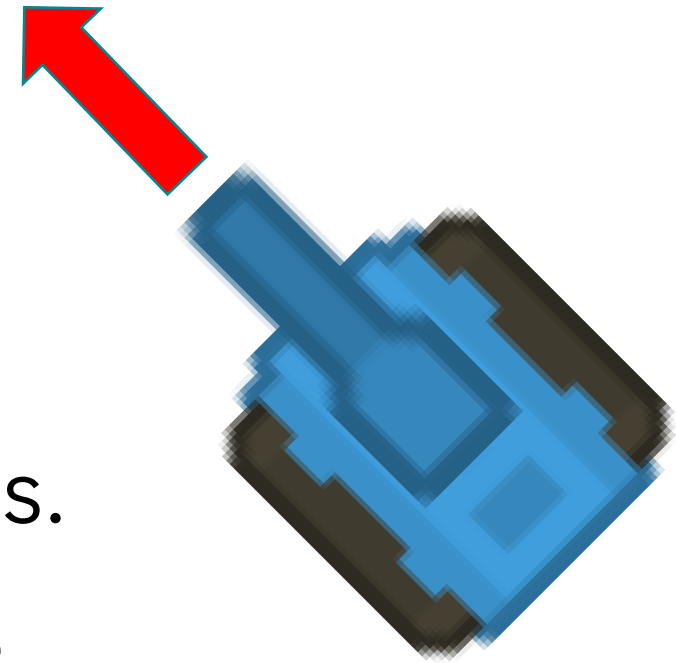
```
if (_interval < _timeElapsed and _playerAlive[id]):
    _network._sendMsg(id, tank[id].position, tank[id].get_BodyRot(), tank[id].get_BarrelRot(), false, _timeElapsed)
    _interval += 0.1
```

In the lobby messages and sent when needed*

Prediction Techniques Used

Tank Prediction

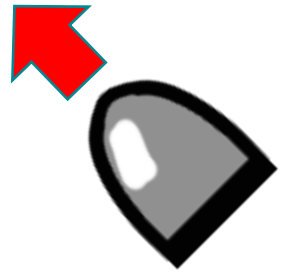
- The tank can only move forward and rotate.
- There will be fewer sharp movements.
- Linear prediction will be appropriate.



```
var speedl = (_messagePos[2] - _messagePos[1]) / (_messageTime[2] - _messageTime[1])
var displacement = speedl * (_timeElapsed-_messageTime[2])
var _predictedPos = _messagePos[2] + displacement
```

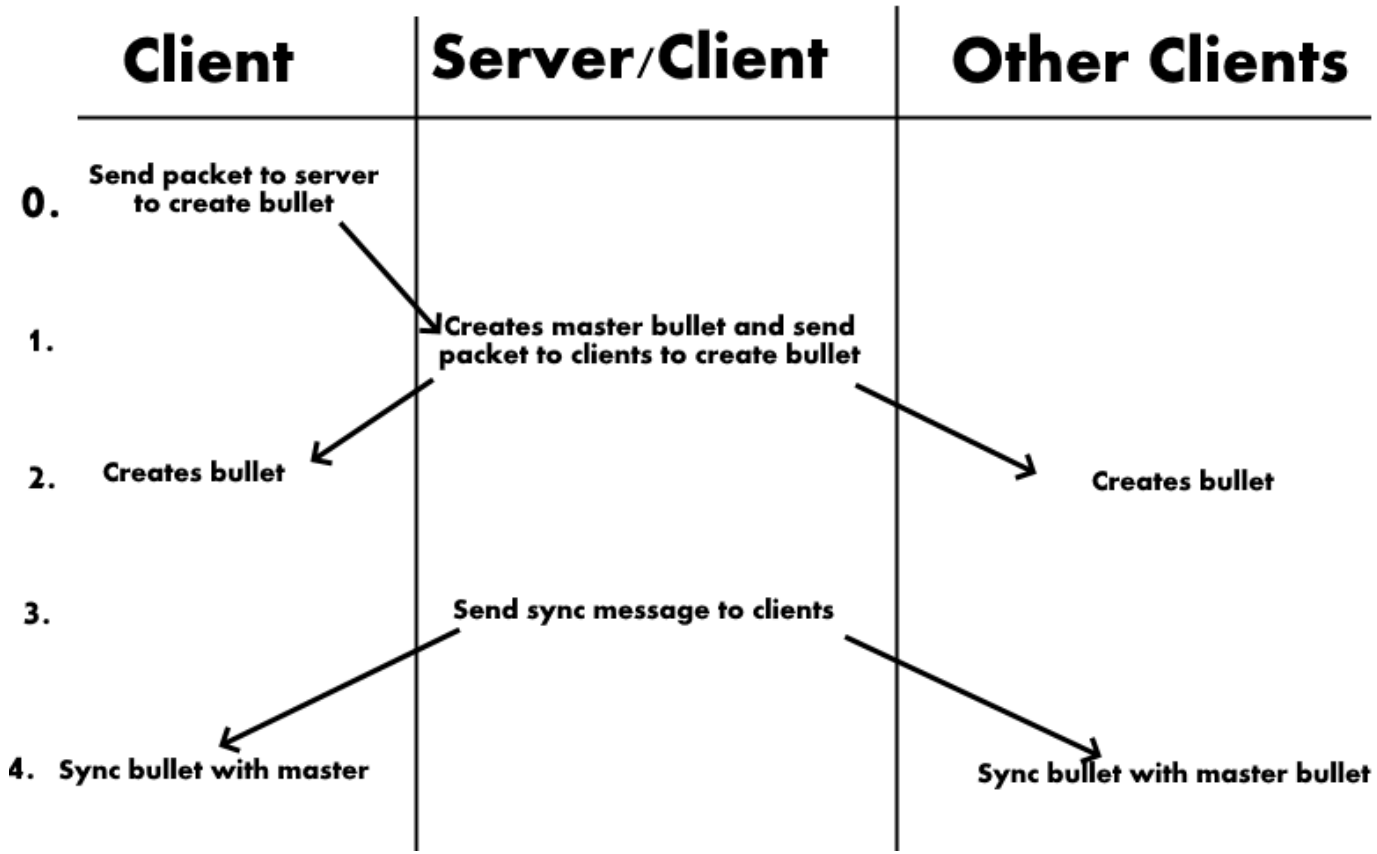
Bullet Prediction

- Bullets only move in one direction, and bounce.
- The client will simulate the bullets movement.
- Server will send update ever couple seconds.
- Since velocity is constant, we can calculate the position.



$$\text{Bullet Position} = \text{Message Position} + \left(\text{Velocity} \times (\text{Client Time} - \text{Message Time})\right)$$

Other Design or Development Decisions that relate to networking



Shell Simulation

- The server creates the master shell
- Even if a client wants creates the shell, they wait for the server to tell them to tell them to create it.

Packet Format

- *Lobby has a playerID*
- *Key - to determine the action to be taken on the player*
- *Player Ready state*

```
struct LobbyMessage
{
    int playerID;
    int key;
    bool playerReady;
};
```

- *Game Message has a playerID*
- *The position of the player*
- *The rotation of the body*
- *Rotation of the barrel*
- *State changes*
- *Time of packet being sent*

```
struct GameMessage
{
    int playerID;
    Position pos;
    float bodyRot;
    float barrelRot;
    bool stateChange;
    float time;
};
```

A Critical Discussion of effectiveness of solution

Handling Disconnection

- Tracks the time between packets being sent
- If the time exceeds 10 seconds

Client Disconnection

- Removes client
- Shifts over clients down the list
- Positive: Game Continues

Host Disconnection

- Closes socket
- Returns to the Create or Join Screen
- Negative: Game Ends
- Solution: Host Migration

Toggle State Objects

Bullet Created

- Packet sent saying to create a bullet.
- Packet contains position and direction data
- Negative: Duplicate or dropped packets could cause issues.
- Solution: Giving bullet ID's

Destroyable Blocks

- Packet sent saying to destroy block.
- Packet contains the block ID.
- Negative: Duplicate packets could cause array to go out of index.
- Solution: Make array constant size
- Negative: Dropped packets could cause blocks to remain when destroyed
- Solution: Sending multiple updates

Performance Under Bad Conditions

Differences compared to when no effects are being applied*

	Lag (120ms)	Dropped (25%)	Duplicate (25%)	Out of Order (25%)
Enemy Tank 	Visually jumps back a little while turning	Little to no effect	Little to no effect	No effect
Bullet (non-master) 	No effect	Chance of bullet not spawning on clients	Chance of two bullets spawning	No effect