Let’s Make an Multiplayer Game Without an Engine

… WHAT?

Premise

- We will learn how to make a simple multiplayer game
- We will use C++20 coroutines and ASIO library for basic networking objects
- We will NOT make production ready code
- We will make a functional and understandable PROTOTYPE

Speaker

- Tomáš Janoušek
- Work Experience: Arma 4, Mafia: The Old Country, TopSpin 2K25
- Networking Experience: Master's thesis
- You don't need any of it to start

Recommended Tools

- VS Code
- C++20
- CMake
- Git

Hello, server!

First we will show an example, how to connect to a server with ASIO library.

Connecting to a Server 1/2

```cpp
#include <iostream>
#include <asio.hpp>
using asio::ip::tcp;

void connect(tcp::socket& socket, const tcp::resolver::results_type& endpoints);

int main(int argc, char* argv[]) {
  asio::io_context context;  // executes asynchronous tasks
  tcp::socket socket(context);  // connection between computers
  tcp::resolver resolver(context);  // address finder
  auto endpoints = resolver.resolve("127.0.0.1", "8080");
  connect(socket, endpoints); // connect socket to address, next slide
  context.run(); // run all queued tasks
  return 0;
}
```

Connecting to a Server 2/2

```cpp
void connect(tcp::socket& socket, const tcp::resolver::results_type& endpoints) {
  asio::async_connect(socket, endpoints, [](std::error_code ec, tcp::endpoint) {
    if (ec) {
      std::cout << "Could not connect: " << ec.message() << "\n";
      return;
    }
    std::cout << "Connected.";
  });
}
```

Introspection

- We connected a client to already running server
- We created an io context to run asynchronous tasks
- We used a callback
- If we would need to send message to server after connection is established, we would need a callback inside a callback or handle state in some flags.

Coroutines

Now we are aware of downsides of callbacks, so let's explore another approach.

Coroutines: What are they?

- Better functions™
- They run and return a value, however, they can be paused (suspended) to wait
- Program can do things in the meantime (draw graphics, handle input, etc.)
- Not specific to networking, but very useful for networking

ASIO Coroutines

- Function signature has return type of `asio::awaitable<Type>`
- Spawned using `asio::co_spawn(executor, awaitable_fn, exception_handler)`
- Suspended using `co_await awaitable_fn` (C++20 keyword)
- Values returned using `co_return` (C++20 keyword)

Wait for Hello world!

```cpp
#include <iostream>
#include <asio.hpp>

asio::awaitable<int> greeter(asio::io_context& context) {
    asio::steady_timer timer(context, asio::chrono::seconds(3));
    std::cout << "Wait for it... ";
    co_await timer.async_wait(asio::use_awaitable);
    std::cout << "Hello world!";
    co_return 0;
}

int main() {
    asio::io_context context;
    asio::co_spawn(context, greeter(context), asio::detached);
    context.run();
    return 0;
}
```

Client Server Connection

Once again, we will connect a client to server. Then we will create a server that will wait for any number of connections.

Client Connection

```cpp
asio::awaitable<void> connect(tcp::socket& socket,
    const tcp::resolver::results_type& endpoints) {
  auto [error_code, _endpoint] =
    co_await asio::async_connect(socket, endpoints, asio::as_tuple);
  if (error_code) {
    std::cout << "Could not connect: " << error_code.message() << "\n";
    co_return;
  }
  std::cout << "Connected!";
}

int main(int argc, char* argv[]) {
  // ...
  co_spawn(context, connect(socket, endpoints), asio::detached);
  // ...
}
```

Server Connection 1/2

```cpp
#include <iostream>
#include <asio.hpp>
using asio::ip::tcp;

asio::awaitable<void> listener();

int main(int argc, char* argv[]) {
  asio::io_context context;
  asio::signal_set signals(context, SIGINT /*interrupt*/, SIGTERM /*terminate*/);
  signals.async_wait([&](asio::error_code _ec, int _signal){ context.stop(); });
  co_spawn(context, listener(), asio::detached);
  context.run();
  return 0;
}
```

Server Connection 2/2

```
asio::awaitable<void> listener() {
  auto executor = co_await asio::this_coro::executor;
  std::cout << "Staring a server on port 8080...";
  tcp::acceptor acceptor(executor, {tcp::v4(), 8080});
  while (true) {
    auto [ec, socket] = co_await acceptor.async_accept(asio::as_tuple);
    if (ec) {
      std::cout << "Could not accept a new connection: " << ec.message() << "\n";
      co_return;
    }
    std::cout << "There is a new connection!\n";
    // ... use socket variable to send and receive messages
  }
}
```

Test, test, test!

Before making the fully-fledged game, we need to make sure our basic functionality is working as intended. We will try that we can connect notebook to PC. PC will use fixed and wired internet, notebook will use internet from a mobile hotspot from a different ISP.

Why we can't run just server...

- Private IP from ISP
- Firewall blocking external packets
- Router not forwarding the port
- Antivirus blocking packets

How to circumfluent internet protections

- Cloud server (AWS, Azure, Google Cloud): 100s€/month
- Custom Server & Public IP: 10€/month + Server Cost
- Virtual Private Network (Hamachi, Zero Tier, RadminVPN): various free plans

Virtual Private Network

- All users need to download the same program.
- One player sets up a new private network.
- Other players then connect to that network.
- Then, all can communicate freely as they are on the same network, meaning one player can run the server and others join it.

Recap

- We can create a client server connection
- We can use coroutines (think of JS `async` / `await`)
- We can test our program over the real internet
- We still want to make a multiplayer game

Folder structure

```
project
  |-- client
    |-- main.cpp
    |-- game.hpp, game.cpp
    |-- network.hpp, network.cpp
    |-- renderer.hpp, renderer.cpp
  |-- server
    |-- main.cpp
    |-- server.hpp, server.cpp
  |-- shared
    |-- player.hpp, player.cpp
    |-- players.hpp, players.cpp
```

Big picture

- Client executable is actual game you send to players with graphics, keyboard handling
- Server executable is run only on dedicated server, without graphics and without interaction
- Shared files contain files that can both client and server use

Shared files

- Player structure that contains player information
- Players structure that contains multiple players in a hashmap

Player attributes

- constant speed
- current position
- target position
- current avatar

Player structure

```cpp
struct Player {
  static constexpr float speed = 0.1f;
  float target_x;
  float target_y;
  float current_x;
  float current_y;
  int avatar;
}
```

Player methods

- Constructor with and without parameters
- Serialization, deserialization
- Position update (moves to target)
- Usage: TODO

Player methods

```cpp
struct Player {
// ...
  Player();
  Player(float x, float y, int avatar);
  std::string serialize() const;
  void deserialize(std::string serialized);
  void update_position(int delta_ms);
};
```

Players

- wrapper over hashmap for player_id -> player
- serialization, deserialization
- usage: TODO

Players

```
struct Players {
  std::unordered_map<int, Player> data;

public:
  std::string serialize() const;
  void deserialize(std::string serialized);
};
```

Client

- Main creates the executable, offloads most of the work to independent modules
- **Game** module handles only game logic and serialization
- **Network** module handles only networking logic
- **Renderer** module handles only graphics

Main

- Creates the application
- Handles key presses
- Updates graphics and receives network updates
- Destroys the application when closed

Main

```c
#define SDL_MAIN_USE_CALLBACKS 1
#include <SDL3/SDL.h>
#include <SDL3/SDL_main.h>

SDL_AppResult SDL_AppInit(void **appstate, int argc, char *argv[]);
SDL_AppResult SDL_AppEvent(void *appstate, SDL_Event *event);
SDL_AppResult SDL_AppIterate(void *appstate);
void SDL_AppQuit(void *appstate, SDL_AppResult result);
```

Game

- Moves my player across the screen
- Updates all players based on serialized string
- Updates my player when key is pressed
- Updates my player when key is released

Game

```
class Game {
  // ...
public:
  int get_player_id();
  Players const& get_players();
  Player const& get_player();
  void update_state_locally();
  std::string serialize();
  void deserialize(std::string serialized);
  void key_press(SDL_Scancode scancode, int avatars_count);
  void key_release(SDL_Scancode scancode);
}
```

Renderer

- Loads all textures to memory
- Draws the textures (sprites) on players positions
- Unloads all textures when closing the game

Renderer

```
class Renderer {
// ...
public:
  int get_avatars_count();
  SDL_AppResult init();
  void update(int player_id,
    Player const& player, Players const& players);
  void quit();
};
```

Network public methods

- connects to server address and server port
- polls server updates

```cpp
using SerializeFn = std::function<std::string()>;
using DeserializeFn = std::function<void(std::string)>;

class Network {
// ...
public:
  Network();
  void connect_to_server(
    const char* server_address,
    const char* server_port,
    SerializeFn serialize,
    DeserializeFn deserialize);
  SDL_AppResult receive_updates();
};
```

Network private methods

- connects to resolved endpoints
- writes updates to server
- reads server updates

Network private methods

```cpp
class Network {
  static constexpr int client_update_ticks = 1; // 1 update per second
  asio::io_context context;
  asio::ip::tcp::socket client_socket;
  SerializeFn serialize;
  DeserializeFn deserialize;
  bool should_exit = false;

private:
  asio::awaitable<void> write_to_server();
  asio::awaitable<void> read_from_server();
  asio::awaitable<void> connect_to_endpoints(
    const asio::ip::tcp::resolver::results_type endpoints);
}
```

Server

- Main file that is used to create the executable
- Server file which contains networking logic

Server public methods

- start listening
- update game state

Server public methods

```cpp
class Server {
// ...
public:
  asio::awaitable<void> start_listening_on(asio::ip::port_type port);
  asio::awaitable<void> update_game_state();
};
```

Server private methods

- connecting a new socket
- writing an update to player
- receiving an update from player

Server private methods

```cpp
class Server {
  static constexpr int server_update_ticks = 10; // 10 updates per second
  Players players;
  int last_used_id = 0;
  std::unordered_map<int, asio::ip::tcp::socket> sockets;

private:
  asio::awaitable<void> write_to_player(int player_id);
  asio::awaitable<void> read_from_player(int player_id);
  asio::awaitable<void> connect_new_socket(asio::ip::tcp::socket socket);
}
```

Server messages

```
...
Wrote to player 3: 3 0 2 144 442 1 3 632 521 2 0 240 135 3 0 330 318
Wrote to player 1: 1 0 2 144 442 1 3 632 521 2 0 240 135 3 0 330 318
Wrote to player 2: 2 0 2 144 442 1 3 632 521 2 0 240 135 3 0 330 318
Wrote to player 0: 0 0 2 144 442 1 3 632 521 2 0 240 135 3 0 330 318
Wrote to player 3: 3 0 2 144 442 1 3 632 521 2 0 240 135 3 0 330 318
Wrote to player 1: 1 0 2 144 442 1 3 632 521 2 0 240 135 3 0 330 318
Wrote to player 2: 2 0 2 144 442 1 3 632 521 2 0 240 135 3 0 330 318
Wrote to player 0: 0 0 2 144 442 1 3 632 521 2 0 240 135 3 0 330 318
Wrote to player 3: 3 0 2 144 442 1 3 632 521 2 0 240 135 3 0 330 318
Wrote to player 1: 1 0 2 144 442 1 3 632 521 2 0 240 135 3 0 330 318
...
```

Server update string

```
Wrote to player 3: 2 0 4 144 442 1 5 632 521 2 2 240 135 3 1 330 318
Wrote to player 3: your_id (player_id player_avatar player_x player_y)+
```

- your_id: `3`
- player 0: `4 144 442`
- player 1: `5 632 521`
- player 2: `2 240 135`
- player 3: `1 330 318`