DyeWars Development Session Summary

What We Built

This session added a **player facing/direction system** and **improved packet architecture** to the multiplayer game.

Core Concepts Covered

1. Smart Pointers and Memory Management

We explored C++ memory management patterns:

Type	Purpose	When to Use
`std::unique_ptr`	Single owner, auto-cleanup	Object has one clear owner (e.g., `Player` inside `GameSession`)
`std::shared_ptr`	Shared ownership, reference counted	Multiple owners or outlives creating scope (e.g., `GameSession`)
Raw pointer (`*`)	Observing, no ownership	Pointing to something guaranteed to outlive you (e.g., `GameServer*` in `GameSession`)

Key insight: `make_shared` and `make_unique` allocate on the heap and wrap in a smart pointer. When the smart pointer dies, memory is automatically freed.

```cpp
// Heap allocation with auto-cleanup
auto session = std::make_shared<GameSession>(...);
auto player = std::make_unique<Player>(id, 0, 0);

// Raw pointer - just observing, no cleanup responsibility
GameServer* server_;  // Safe because GameServer outlives GameSession
```

---

2. Threading Model

The server runs 4 threads:

Thread	Created Where	Responsibility
Main (io_context)	`main()` → `io_context.run()`	All network I/O (async accept, read, write)
Game Loop	`GameServer` constructor	20 ticks/sec, processes updates, calls Lua
Console	`StartConsole()`	Keyboard commands ('r', 'q')
File Watcher	`LuaGameEngine`	Monitors Lua scripts for hot-reload

Key insight: Async I/O means one thread handles thousands of connections. No new thread per client.

```cpp
// async_read doesn't block - it registers a callback and returns immediately
asio::async_read(socket_, buffer, [this, self](std::error_code ec, std::size_t) {
    // This runs LATER when data arrives
});
// Code here runs IMMEDIATELY
```

---

3. Mutex and Thread Safety

When multiple threads access shared data, use a mutex:

```cpp
std::mutex sessions_mutex_;

// Lock just long enough to access shared data
{
    std::lock_guard<std::mutex> lock(sessions_mutex_);
    // Access sessions_ safely
}  // Lock automatically released

// Do slow work OUTSIDE the lock
```

Key insight: `lock_guard` uses RAII pattern—automatically unlocks when it goes out of scope, even on exceptions or early returns.

---

4. The `shared_from_this()` Pattern

For async callbacks that might fire after an object could be deleted:

```cpp
```

```cpp
void GameSession::ReadPacketHeader() {
    auto self(shared_from_this());  // Keep this session alive
    asio::async_read(socket_, buffer, [this, self](...) {
        // Even if session removed from map, 'self' keeps it alive until callback completes
    });
}
```

Features Implemented

Player Facing System

Server (C++):

Added `facing_` member to `Player` class:

```cpp
class Player {
    uint32_t id_;
    int x_;
    int y_;
    uint8_t facing_ = 2;  // 0=up, 1=right, 2=down, 3=left

public:
    void SetFacing(uint8_t direction);
    uint8_t GetFacing() const { return facing_; }
};
```

Client (C#):

Added facing state and turn logic:

```csharp
public int MyFacing { get; private set; } = 2;
public Dictionary<uint, int> OtherPlayerFacings { get; private set; } = new();
public System.Action<int> OnMyFacingUpdated;

private void HandleDirectionInput(int direction) {
    if (MyFacing == direction)
        SendMoveCommand(direction);    // Already facing, move
    else
        SendTurnCommand(direction);    // Turn first
}
```

Packet Helper System

Created reusable functions for reading/writing packet data.

Server (C++):

```cpp
namespace PacketWriter {
    inline void WriteU8(std::vector<uint8_t>& buffer, uint8_t value);
    inline void WriteU16(std::vector<uint8_t>& buffer, uint16_t value);
    inline void WriteU32(std::vector<uint8_t>& buffer, uint32_t value);
}

namespace PacketReader {
    inline uint8_t ReadU8(const std::vector<uint8_t>& buffer, size_t& offset);
    inline uint16_t ReadU16(const std::vector<uint8_t>& buffer, size_t& offset);
    inline uint32_t ReadU32(const std::vector<uint8_t>& buffer, size_t& offset);
}
```

Client (C#):

```csharp
private byte ReadU8(byte[] payload, ref int offset) {
    return payload[offset++];
}

private ushort ReadU16(byte[] payload, ref int offset) {
    ushort value = (ushort)((payload[offset] << 8) | payload[offset + 1]);
    offset += 2;
    return value;
}

private uint ReadU32(byte[] payload, ref int offset) {
    uint value = (uint)((payload[offset] << 24) | (payload[offset + 1] << 16) |
                        (payload[offset + 2] << 8) | payload[offset + 3]);
    offset += 4;
    return value;
}
```

Updated Packet Protocol

Changed coordinates from 1 byte (max 255) to 2 bytes (max 65535) for future map sizes.

Opcode	Name	New Payload	Direction
`0x01`	Move	`[direction:1][facing:1]`	C → S
`0x04`	Turn	`[direction:1]`	C → S
`0x10`	Your Position	`[x:2][y:2][facing:1]`	S → C
`0x12`	Other Player Update	`[id:4][x:2][y:2][facing:1]`	S → C
`0x13`	Your Player ID	`[id:4]`	S → C
`0x14`	Player Left	`[id:4]`	S → C
`0x15`	Facing Update	`[facing:1]`	S → C
`0x20`	Batch Update	`[count:1][id:4,x:2,y:2,facing:1]...`	S → C

Sprite Direction Rendering

Unity `GridRenderer` now swaps sprites based on facing:

```csharp
[SerializeField] private Sprite[] directionSprites; // 0=up, 1=right, 2=down, 3=left

private void OnMyFacingUpdated(int facing) {
    if (localPlayerInstance != null && facing < directionSprites.Length) {
        SpriteRenderer sr = localPlayerInstance.GetComponent<SpriteRenderer>();
        if (sr != null) {
            sr.sprite = directionSprites[facing];
        }
    }
}
```

Key Debugging Lesson

When events aren't firing, trace the full flow:

1. Is the sender calling the event? → Add `Debug.Log` before invoke
2. Are there subscribers? → Check `event?.GetInvocationList()?.Length`
3. Is the subscriber registering? → Check subscription code

Our bug: `OnMyFacingUpdated` subscription was missing—we had a duplicate `OnMyPositionUpdated` instead.

Architecture Decisions Discussed

Decision	Choice	Reasoning
Where to call Lua events	Game Loop thread	Keeps network thread fast, non-blocking
Lock duration	As short as possible	Grab data, release lock, then do slow work
Raw pointer for `GameServer*`	Yes	Server guaranteed to outlive sessions
`uint16_t` for coordinates	Yes	Supports maps up to 65535x65535
Queue outgoing packets	Future improvement	Centralizes all sends, easier debugging

---

Files Modified

Server:

- `include/server/Player.h` — Added `facing_`
- `src/server/Player.cpp` — Added `SetFacing()`
- `include/server/Common.h` — Added `facing` to `PlayerData`
- `include/server/PacketHelpers.h` — New file with read/write helpers
- `src/server/GameSession.cpp` — Updated packet handling
- `src/server/GameServer.cpp` — Updated batch packet

Client:

- `Assets/code/NetworkManager.cs` — Added facing, turn command, packet helpers
- `Assets/code/GridRenderer.cs` — Added sprite swapping for facing