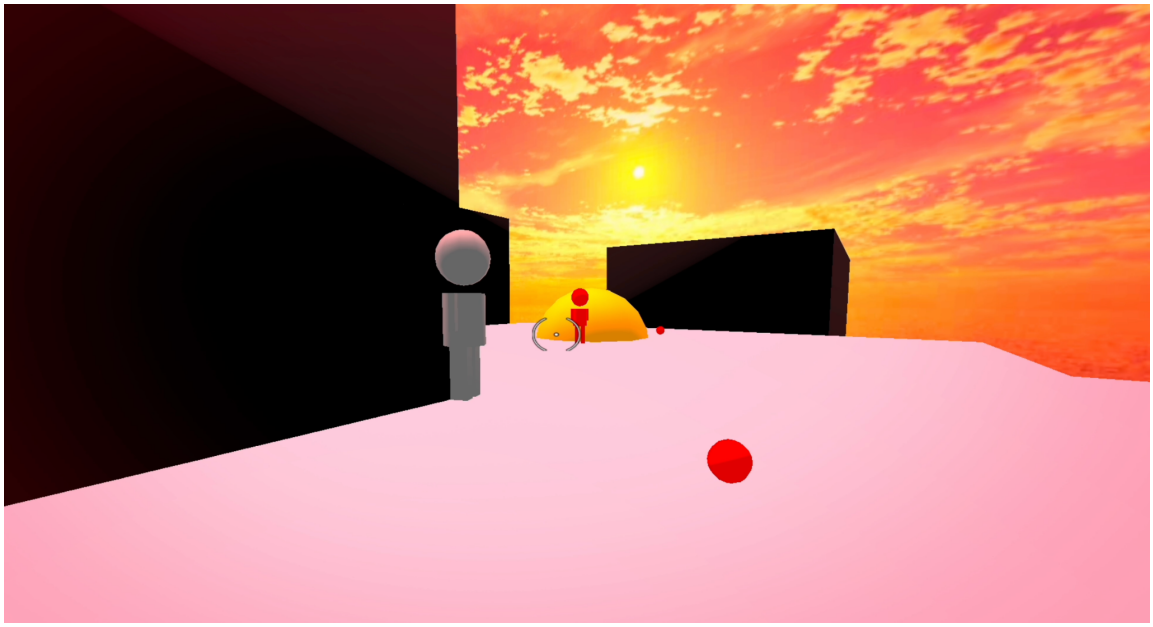
CSCI 1680 Final Project

Networks: Evan Mickelson and Matthew McQuistion
Graphics: Evan Mickelson, Jakob Wismar, and Luke Riley

Github Repo Demo Video

- **Introduction**: What were your overall project goals? What (briefly) did you achieve?

The overall goal of this project was to design and make a multiplayer FPS game. This would provide challenges for synchronizing physics and game state across many clients. We ended up achieving a playable multiplayer deathmatch game.



- **Design/Implementation**: What did you build, and how does it work? For this part, give an overview of the major components of your system design and how they work, similar to what you might write in a readme.

The project's core was the “Entity Component System,” or “ECS” for short. This design enabled us to synchronize game state across different clients more easily than other game architectures.

As an overview, the ECS consists of entities (32 bit integers with each bit corresponding to a component), components (structs of data associated with different entities), and systems (functions that apply to all entities with given components).

As a user of our ECS, you register components and systems beforehand. Those get stored in continuous arrays where the index into the array is the entity’s id. Another continuous array is stored that maps entity ids to their corresponding component flags. Then, you register “systems”

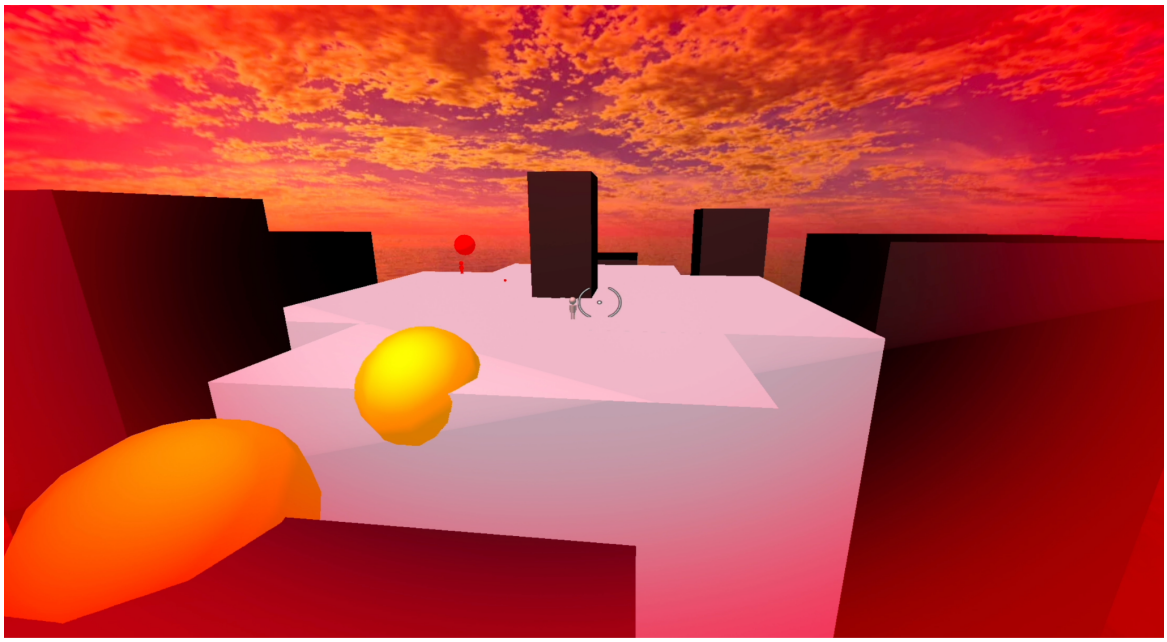
which are functions that have the ecs, entity, and delta time as parameters along with a bitmask of required components.

Once registered, each system is called on all entities with corresponding bitmasks every frame.

We opted to use arrays in the backend because we had 1 byte corresponding to each component's ID, meaning there would be 256 elements for each component. While there could be optimizations here to make this expandable, having it fixed means we get ease of use from the network side, and other efficiencies (such as cache hits). It also reduced the complexity of our code, and given this project's scope was large between needing both graphics and networking capabilities, that was helpful. In addition, the scale of our project is pretty small (only a handful of players shooting no more than ~5 projectiles at a time), and thus it was extremely unlikely we would need more than 256 entities at a time.

To prevent jitter and allow for similar designs between the server and clients, we have a system of “authority.” A client will have authority over anything it creates, and the server initially sends the corresponding player to kick it off. The server has authority over everything else. Any network client with authority over an object has authority over all of the client's state; incoming packets won’t override their version of its state, and they will send out packets with entities they have authority over.

One issue with this is that there needs to be more local security. An attacker could modify the client to position their player or projectiles anywhere on the map without checks. On the other hand, it also ensures that malicious actors cannot edit clients.



Because ECS is a data-driven system, designing the actual networking was relatively straightforward. No TCP was used, only UDP sockets. The server starts up, a connection is established using UDP and getting IP information from the sockets, then every game tick the state is broadcasted.

To broadcast state, the server (or client with all entities it has authority over) iterates through all the data in the ECS and puts any alive and registered entities and data onto UDP packets, and sends them off. Because state updates are sent so frequently and because speed is of the essence, reliability and packet order are not necessary. However, each packet header has information about what local tick it was sent on, which is then used to compare with the last received packets to know what or what not to deserialize. The data is serialized by registering object, flag, then the component data, and deserialized by copying component data based on the inputted object and flags (component order is the same order as on the flags), and "null-terminated" when the next object and flags are both 0.

```
void Network::deserializeAllDataIntoECS() {

    // Iterate through all connections, pop once per tick
    std::lock_guard<std::mutex> lock(m_connectionMutex);

    while (!buffer.empty()) {

        TickData& nxt = buffer.front();
        if (m_connectionMap.size() > 0 && nxt.data == nullptr) {
            return;
        }

        char buff[FULL_PACKET];

        // Ignore authority for first established tick
        if (next.tick < 0) {
            m_ecs->deserializeIntoData(nxt.data, nxt.data_size, true);
        } else
            m_ecs->deserializeIntoData(nxt.data, nxt.data_size, false);

        delete[] nxt.data;
        nxt.data = nullptr;
        buffer.pop();
    }

    if (next.tick < 0)
        next.tick = 0;
}
```

```
void ECS::deserializeIntoData(char* serialized_data, size_t max_size, bool ignore_auth) {
    // IMPORTANT: what happens to used data size when a new object is deserialized in?
    // ALSO, ensure that tip of data isn't full if flags are empty/object is destroyed
//    std::cout << "Deserializing into data (ECS)" << std::endl;

    if (serialized_data == nullptr) {
//        std::cout << "Null Serialized Data" << std::endl;
        return;
    }
    size_t ob_ptr = 0;

    entity_t ent;
    flags_t flags;

    while (ob_ptr < max_size) {

        // get initial entity data
        memcpy(&ent, &serialized_data[ob_ptr], sizeof(entity_t));
        ob_ptr += sizeof(entity_t);
        memcpy(&flags, &serialized_data[ob_ptr], sizeof(flags_t));
        ob_ptr += sizeof(flags_t);

        // end early if empty entity / end of data
        if (ent == 0 && flags == 0) {
            std::cout << "breaking"  <<std::endl;
            break;
        }

        bool cpy = (m_entities[ent] == 0);

        // Remove data if deleted
        if (m_entities[ent] != 0 && flags == 0) {
            for (int com = 0; com < MAX_COMPONENTS; com++) {
                bool has_flag = m_entities[ent] & (1 << com);
                if (!m_component_registered[com] || !has_flag) {
                    continue;
                }
                m_usedDataSize -= m_component_num_to_size[com];
            }
        }
        // Copy over all component data
        // IMPORTANT: what if entity doesn't exist before?
        m_entities[ent] = flags;
        for (int com = 0; com < MAX_COMPONENTS; com++) {
            bool has_flag = m_entities[ent] & (1 << com);
            if (!m_component_registered[com] || !has_flag) {
                continue;
            }

            // Only copy if authority allows for it
            if (!m_hasAuthority[ent] || ignore_auth)
                memcpy(getComponentData(ent, com), (serialized_data + ob_ptr), m_component_num_to_size[com]);

            ob_ptr += m_component_num_to_size[com];
            if (cpy)
                m_usedDataSize += m_component_num_to_size[com];

        }
    }

    resetNextAlloc();
}
```

To ensure synchronization, the client and server register the same systems and components (except the render system because that is only client-sided). They are running identical systems and constantly synchronizing their data, so everything should be relatively in sync.

All in all, the process goes something like this:

- The server starts up
- A client sends a request to the server on a pre-agreed-upon port and IP
- The server gets this request, creates a “connection” struct representing that client, and sends back a welcome message with the player entity corresponding with the client
- The client receives the welcome message gets its player entity, and also sets up a connection struct representing the server
- The game loop begins
- At the beginning of each frame, the server and client check a queue of game state updates.
    - If this queue is not empty, copy its data onto the local version of the ECS for all objects that they do not have authority over
        - For the first set of packets received, authority doesn’t matter. They will just receive the state from the server.
- Run and render the simulation (physics, rendering, etc.)
    - Any objects created by objects this client or server has authority over will also have authority by the local client
- Broadcast the game state
    - For the client, broadcast all objects they have authority over to the server
    - For the server, broadcast the whole game state to all clients (when deserialized, if they have authority, it will be ignored)

- **Discussion/Results**: Describe any results you have, what you have learned, and any challenges you faced along the way. For this part, please include any relevant logs/screenshots of your program operating (and/or reference your demo video).

It took a while for us to get the server working consistently. We also had to do a lot of wrangling and meetups to align the graphics and server teams. Debugging was much easier as we had new tools and could visualize issues. Most of the progress was slow-moving until the very end, where much of the game logic came together in the last day or two.

- **Conclusions/Future work**: Overall, what have you learned? How did you feel about this project overall? If you could keep working on this project, what would you do next?

We felt that the unique nature of this project being split across two very different classes offered one of the most interesting takes on not only how network programming is actually used in the context of a real “product” but also how to balance trade-offs between both camps to ensure that networking is smooth and reliable with minimal compromise to graphics quality or game features. Although ECS was extremely well-suited to help us find this balance, the need to

communicate between graphics programmers and network designers could still cause a bit of friction, given that only one person in the group was in both classes and had to act as a liaison between the two graphics/rendering-centered team members and the other student in networks.

The next significant thing we want to do on this project is graphics interpolation. Right now, because the entire state is synched up and the same transform data used for components is used for physics, there is a lot of jitter for the graphics from other players. To solve this problem, we would need a system to interpolate the graphics' transformation based on physics data obtained from previous ticks to render the graphics more smoothly.

Retransmission. Our design approach was incredibly nice because many seemingly hard synchronization problems were solved for us. How do you combat jitter? Give clients control over their player and its data. How about projectiles getting destroyed? All the clients are running the same simulation, so having projectiles destroyed on a timer or hitting a wall will sort itself out. One problem we faced, though, is that once objects are destroyed locally, their data will no longer be broadcast. You could broadcast that they have died by sending empty flags, but it could get dropped if you only do it for one packet, and doing it repeatedly forever is wasteful. If an in-between, how many to send? Having retransmission and acknowledgment to create an implementation of “reliable” (but unordered) UDP (or even interspersing TCP packets) to reliably send over things that either won’t be sorted out by the simulation or are controlled randomly (such as by player commands) is a clear next step.

A better solution to prevent jitter and control than the current authority system. Ideally, the server should authenticate the state it receives from clients. If it strays too much from the servers or in an unusual way, that player can be kicked or banned.

Github Repository: https://github.com/evan-mick/quake-clone
Demo Video: https://drive.google.com/file/d/1oyknDe_eDyTXVMn1sffx-3-eQqOfbj4V/view?usp=sharing