player.cpp

```
#include "headers.hpp"
#include "player.hpp"
#include "tools.hpp"
#include "client.hpp"
#include <time.h>

Player::Player(int x, int y, sf::RenderWindow& window, Toaster&
toaster)
    : toaster(toaster),  map(20, window), position(x, y),
    window(window)
{

    loadTextures();

    loadSFX();

    srand(time(NULL));

}



void Player::setFocus(bool focus)
{
    window.setMouseCursorVisible(!focus);
    window_focused = focus;
    if (focus)
        sf::Mouse::setPosition({ WIDTH / 2, HEIGHT / 2 }, window);
}

#define KEY_PRESSED(key)
sf::Keyboard::isKeyPressed(sf::Keyboard::Key::key)
void Player::handleKeys(float dt)
{
    if (!window_focused)
        return;

    if (KEY_PRESSED(Escape))
        quitGame();




    // move
    v2f movement(0, 0);
    if (KEY_PRESSED(W))
        movement.y += +1;
    if (KEY_PRESSED(A))
        movement.x += +1;
    if (KEY_PRESSED(S))
        movement.y += -1;
    if (KEY_PRESSED(D))
        movement.x += -1;

    if (dead)
```

```
        movement = v2f(0, 0);

    moving = (movement.x != 0 || movement.y != 0);

    if (moving)
        move(atan2(movement.x, movement.y), dt);


    moving_forward = movement.y > 0;

    // run
    if (KEY_PRESSED(LShift))
        running = true;
    else
        running = false;

    if (KEY_PRESSED(LControl))
        crouching = true;
    else
        crouching = false;



}

void Player::quitGame()
{
    has_quit = true;



    updateServer();

    window.close();
}


void Player::rotateHead(int delta_x, int delta_y, float dt)
{
    // ignore enourmous rotation requests
    float move_size = mag(v2f(delta_x, delta_y)) * mouse_sensitivity *
    dt;
    if (dt != -1 && move_size > 0.6f)
    {
        cout << "size " << move_size << " movement suppressed" << "\n";
        return;
    }



    // horizontal
    rotation_x += delta_x * mouse_sensitivity * dt;
    map.shiftSky(delta_x * mouse_sensitivity * dt); // shift sky
    // vertical
    rotation_y += delta_y * -mouse_sensitivity * dt;


    // cap y rotation
    if (rotation_y > -0.15f)
        rotation_y = -0.15f;

    if (rotation_y < -1.05f)
        rotation_y = -1.05f;

    map.floor_level = HEIGHT / 2 * (1 + tan(rotation_y)) / tan(fov_y /
    2);

}

void Player::move(float angle_offset, float dt)
{

    float current_speed = speed;
    if (running)
        current_speed *= run_multiplier;
    else if (crouching)
        current_speed *= crouch_multiplier;

    v2f potential_position;
    potential_position.x = position.x + current_speed * run_multiplier
    * dt * cos(rotation_x - angle_offset);
    potential_position.y = position.y + current_speed * run_multiplier
    * dt * sin(rotation_x - angle_offset);

    v2i ones(1, 1);
    v2i predicted_cell = v2i(floor(potential_position.x),
    floor(potential_position.y));

    v2i area_tl = min((v2i)position, predicted_cell);
    area_tl = max({ 0, 0 }, area_tl - ones); // clamp


    v2i area_br = max((v2i)position, predicted_cell);
    area_br = min({ map.width, map.height }, area_br + ones); // clamp


    for (int cell_x = area_tl.x; cell_x <= area_br.x; cell_x++)
    {
        for (int cell_y = area_tl.y; cell_y <= area_br.y; cell_y++)
        {

            // is wall
            if (map.getCell(cell_x, cell_y) == 1)
            {
                v2f nearest_point;
                nearest_point.x = std::max(float(cell_x),
                std::min(potential_position.x, float(cell_x + 1)));
                nearest_point.y = std::max(float(cell_y),
                std::min(potential_position.y, float(cell_y + 1)));

                v2f ray_to_nearest = nearest_point -
                potential_position;

                float overlap = body_radius - mag(ray_to_nearest);

                if (std::isnan(overlap)) overlap = 0;

                if (overlap > 0)
                {
                    potential_position -= norm(ray_to_nearest) *
                    overlap;
                }
            }

        }
    }

    position = potential_position;
}

// returns distance in that direction
Player::HitInfo Player::shootRay(float angle_offset)
{
    //angle_offset = angle_offset * (tan(angle_offset));

    v2f ray_dir(cos(rotation_x + angle_offset), sin(rotation_x +
    angle_offset));

    v2f ray_unit_step_size;
    ray_unit_step_size.x = sqrt(1 + (ray_dir.y / ray_dir.x) *
    (ray_dir.y / ray_dir.x));
    ray_unit_step_size.y = sqrt(1 + (ray_dir.x / ray_dir.y) *
    (ray_dir.x / ray_dir.y));

    v2i current_cell(position);

    v2f ray_length;
    v2i cell_step;

    if (ray_dir.x < 0)
    {
        cell_step.x = -1;
        ray_length.x = (position.x - (float)current_cell.x) *
        ray_unit_step_size.x;
    }
    else
```

```
{
    cell_step.x = 1;
    ray_length.x = (float(current_cell.x + 1) - position.x) *
    ray_unit_step_size.x;
}

if (ray_dir.y < 0)
{
    cell_step.y = -1;
    ray_length.y = (position.y - (float)current_cell.y) *
    ray_unit_step_size.y;
}
else
{
    cell_step.y = 1;
    ray_length.y = (float(current_cell.y + 1) - position.y) *
    ray_unit_step_size.y;
}

float distance = 0;
bool tile_found = false;
bool latest_hit_on_x;
while (!tile_found)
{
    if (ray_length.x < ray_length.y)
    {
        current_cell.x += cell_step.x;
        distance = ray_length.x;
        latest_hit_on_x = false;
        ray_length.x += ray_unit_step_size.x;
    }
    else
    {
        current_cell.y += cell_step.y;
        distance = ray_length.y;
        latest_hit_on_x = true;
        ray_length.y += ray_unit_step_size.y;

    }

    if (current_cell.x < 0 || current_cell.y < 0 || current_cell.x
    >= map.width || current_cell.y >= map.height)
        return { -1 };

    //hit wall
    if (map.getCell(current_cell.x, current_cell.y) == 1)
    {
        v2f hit_position = position + ray_dir * distance;
        //getting the x offset on the texture
        float texture_x;

        if (latest_hit_on_x)
        {
                texture_x = hit_position.x - floor(hit_position.x);
                //if (sin(rotation_x) > 0) texture_x = 1 - texture_x;
            }
            else
            {
                texture_x = hit_position.y - floor(hit_position.y);
                //if (cos(rotation_x) < 0) texture_x = 1 - texture_x;
            }


            return { distance, latest_hit_on_x, texture_x };
        }

    }


}


void Player::drawObject(Object& object, float dt)
{
    float projection_distance = 0.5f / tan(fov_y / 2);


    // object's direction relative to ours.
    float angle_to_object = atan2(position.y - object.position.y,
    object.position.x - position.x) + rotation_x;

    float projection_position = 0.5f * tan(angle_to_object) / tan(fov_x
    / 2);

    float screen_x = WIDTH * (0.5f - projection_position);

    float object_distance = sqrt(pow(position.x - object.position.x, 2)
    + pow(position.y - object.position.y, 2));
    float screen_size = HEIGHT * projection_distance / (object_distance
    * cos(angle_to_object));

    screen_size *= object.scale_by;

    if (screen_size < 0) // behind us
        return;

    v2f player2object = position - object.position;
    v2f object_direction = object.position +
    v2f(cos(object.rotation_x), sin(object.rotation_x));
    //float direction_offset = angleBetweenVectors(object_direction,
    player2object) * TO_DEGREES;// + 22.5f;
    float direction_offset = 360 + 90 + TO_DEGREES * (object.rotation_x
    + atan2(object.position.x - position.x, object.position.y - position.y));

    object.direction_index = ((int)round(direction_offset / 45) % 8 +
    8) % 8;
    object.animate(dt);

    object.sprite.setScale(screen_size * object.shrink_by, screen_size
    * object.shrink_by);
    object.sprite.setPosition(
        screen_x - 0.5f * object.sprite.getTextureRect().height *
        object.sprite.getScale().x,
        map.floor_level - 0.5f * object.sprite.getTextureRect().height
        * (object.sprite.getScale().y - screen_size * (1 - object.shrink_by)));


    window.draw(object.sprite);


    if (object.dead) // no nametag on dead guys
        return;


    // nametag
    sf::Text nametag(object.username, nametag_font, 16);
    nametag.setFillColor(sf::Color::White);
    nametag.setPosition(screen_x - 0.25f *
    nametag.getLocalBounds().width * object.sprite.getScale().x,
        map.floor_level - 0.17f * object.sprite.getTextureRect().height
        * object.sprite.getScale().y);
    nametag.setScale(screen_size, screen_size);
    window.draw(nametag);

}


void Player::shootRays(Player::HitInfo*& hits)
{

    // For each column of pixels
    for (int x = 0; x < WIDTH; x++)
    {
        float y = x / float(WIDTH - 1) - 0.5f;
        float angle = atan(2 * y * tan(fov_x / 2));
        HitInfo hit_info = shootRay(angle);


        hits[x] = hit_info;
        hits[x].perceived_distance = hit_info.distance * cos(angle);


    }

}

void Player::drawWorld(HitInfo*& hits, float dt)
{
    //Sort objects and get the distances
    bool column_drawn[WIDTH];
    for (int x = 0; x < WIDTH; x++)
        column_drawn[x] = false;

    // sort from farthest to closest (descending distance)
    int i;
    for (i = 0; i < sorted_objects.size(); i++)
    {
        bool perfect = true;

        for (int j = 0; j < sorted_objects.size() - 1; j++)
        {
            if (sorted_objects[j]->distFrom(position) <
            sorted_objects[j + 1]->distFrom(position))
            {
                perfect = false;

                Object* temp = sorted_objects[j];
                sorted_objects[j] = sorted_objects[j + 1];
                sorted_objects[j + 1] = temp;
            }
        }

        if (perfect)
            break;
    }


    /*cout << "Objects: ";
    for (int i = 0; i < objects.size(); i++)
    {
        cout << sorted_objects[i]->player_id << " ";
    }
    cout << "enough of objects\n";*/

    for (Object* object : sorted_objects)
    {
        float object_distance = object->distFrom(position);
        for (int x = 0; x < WIDTH; x++)
        {
            if (!column_drawn[x] && hits[x].distance > object_distance)
            {
                drawColumn(x, hits[x]);
                column_drawn[x] = true;
            }
        }

        drawObject(*object, dt);
    }


    for (int x = 0; x < WIDTH; x++)
```

```
        if (!column_drawn[x])
            drawColumn(x, hits[x]);

}

void Player::drawColumn(int x, const Player::HitInfo& hit_info)
{
    float len = 1000 / hit_info.perceived_distance;

    if (hit_info.on_x_axis)
        wall_sprite.setTexture(wall_texs[1]);
    else
        wall_sprite.setTexture(wall_texs[0]);


    // drawing
    wall_sprite.setTextureRect(sf::IntRect(
        v2i(hit_info.texture_x * wall_texs[0].getSize().x, 0), v2i(1,
        wall_texs[0].getSize().y)
    ));
    wall_sprite.setScale(1, len / wall_texs[0].getSize().y);
    wall_sprite.setPosition(x, map.floor_level - len / 2);
    window.draw(wall_sprite);
}

void Player::drawCrosshair(float dt)
{
    if (!dead)
    {
        sf::RectangleShape rect(v2f(4, 12));
        rect.setFillColor(sf::Color::Cyan);

        rect.setPosition(v2f(WIDTH / 2 - 2, HEIGHT / 2 - 16));
        window.draw(rect);

        rect.setPosition(v2f(WIDTH / 2 - 2, HEIGHT / 2 + 4));
        window.draw(rect);


        rect.setSize(v2f(12, 4));
        rect.setPosition(v2f(WIDTH / 2 - 16, HEIGHT / 2 - 2));
        window.draw(rect);

        rect.setPosition(v2f(WIDTH / 2 + 4, HEIGHT / 2 - 2));
        window.draw(rect);
    }


    // damage direction indicator

    for (int i = 0; i < hit_direction_timers.size(); i++)
    {
        hit_direction_timers[i] += dt;
        if (hit_direction_timers[i] > 0.4f) // timer done
        {
            hit_direction_timers.erase(hit_direction_timers.begin() +
            i);
            hit_direction_angles.erase(hit_direction_angles.begin() +
            i);

            i--;
        }
        else
        {
            hit_indicator_sprite.setRotation(180 + (+
            hit_direction_angles[i] - rotation_x) * TO_DEGREES);
            window.draw(hit_indicator_sprite);
        }
    }


    // reticle
    if (reticle_timer >= 0)
    {
        reticle_timer += dt;

        if (reticle_timer > 0.06f)
            reticle_timer = -1;
        else
            window.draw(reticle_sprite);
    }

    if (current_damage_opacity > 0)
    {
        damage_overlay_sprite.setColor(sf::Color(255, 255, 255,
        (int)current_damage_opacity));
        window.draw(damage_overlay_sprite, sf::BlendAlpha);

        if(!dead)
            current_damage_opacity -= dt * 100;
    }

}


void Player::loadTextures()
{
    // walls
    wall_texs = new sf::Texture[2];

    sf::Texture wall;
    wall.loadFromFile("sprites/1L.png");
    wall_texs[0] = wall;
    wall.loadFromFile("sprites/1D.png");
    wall_texs[1] = wall;
```

```
// gun
gun_texs = new sf::Texture[5]();

for (int i = 0; i < 5; i++)
{
    string path = "sprites/gun_animation/gun_X.png";
    path[path.find('X')] = i + '0';
    gun_texs[i].loadFromFile(path);
}

gun_sprite.setTexture(gun_texs[0]);
gun_sprite.setScale(0.8, 0.8);
gun_position = { WIDTH / 2 - gun_texs[0].getSize().x *
gun_sprite.getScale().x / 2 + 25,
HEIGHT - gun_texs[0].getSize().y * gun_sprite.getScale().x + 30 };
gun_sprite.setPosition(gun_position);

gun_animation_timer = 0;
gun_movement_stopwatch = 0;

gun_animation_duration = new float[5];
gun_animation_duration[1] = 0.08f;
gun_animation_duration[2] = 0.12f;
gun_animation_duration[3] = gun_animation_duration[4] = 0.2f;

enemy_tex.loadFromFile("sprites/spritesheet2.png");

indicator_texture.loadFromFile("sprites/indicator.png");
hit_indicator_sprite = sf::Sprite(indicator_texture);

hit_indicator_sprite.setScale(0.7f, 0.7f);

hit_indicator_sprite.setPosition(
    WIDTH / 2,
    HEIGHT / 2);

hit_indicator_sprite.setOrigin(
    hit_indicator_sprite.getLocalBounds().width / 2,
    hit_indicator_sprite.getLocalBounds().height / 2);

reticle_texture.loadFromFile("sprites/hit_marker2.png");
reticle_sprite = sf::Sprite(reticle_texture);
reticle_sprite.setPosition(
    WIDTH / 2,
    HEIGHT / 2);

reticle_sprite.setOrigin(
    reticle_sprite.getLocalBounds().width / 2,
    reticle_sprite.getLocalBounds().height / 2);

reticle_sprite.setScale(0.6f, 0.6f);
```

```
    damage_overlay_tex.loadFromFile("sprites/getting-hit-overlay.png");
    damage_overlay_sprite.setTexture(damage_overlay_tex);
    damage_overlay_sprite.setScale(1.3, 1.3);
    damage_overlay_sprite.setOrigin(WIDTH / 2, HEIGHT / 2);
    damage_overlay_sprite.setPosition(WIDTH / 2, HEIGHT / 2);

    //font
    if (!nametag_font.loadFromFile("Fonts/Roboto-Regular.ttf"))
    {
        cout << "Couldn't Find Nametag Font\n";
    }
    if (!deathscreen_font.loadFromFile("Fonts/Roboto-Light.ttf"))
    {
        cout << "Couldn't Find deathscreen Font\n";
    }
    if (!bold_font.loadFromFile("Fonts/Roboto-Medium.ttf"))
    {
        cout << "Couldn't Find bold Font\n";
    }
}

void Player::loadSFX()
{
    gunshot_buffer.loadFromFile("sfx/9mm-pistol.wav");
    gun_sound.setBuffer(gunshot_buffer);
    gun_sound.setVolume(10);

    gunclick_buffer.loadFromFile("sfx/handgun-release.wav");
    click_sound.setBuffer(gunclick_buffer);
    click_sound.setVolume(25);
}



// draws the hand holding the gun at the bottom of the screen
// dt - deltaTime, the time in seconds since the beginning of the last
frame
void Player::drawGun(float dt)
{
    gun_animation_timer += dt;
    gun_movement_stopwatch += dt * 8;

    if (moving)
        hand_move_range = lerp(hand_move_range, max_hand_range, 0.14f);
    else
        hand_move_range = lerp(hand_move_range, 0, 0.06f);


    float hand_x = sin(gun_movement_stopwatch) * hand_move_range;
    float hand_y = 0.2f * cos(gun_movement_stopwatch) *
    cos(gun_movement_stopwatch) * hand_move_range;

    if (dead)
    {
        gun_offset_y = lerp(gun_offset_y, 300, 0.2f);
    }
    else
        gun_offset_y = lerp(gun_offset_y, hand_y, 0.1f);

    gun_offset = { hand_x , gun_offset_y };

    gun_sprite.setPosition(gun_position + gun_offset);


    //cout << "\n";
    window.draw(gun_sprite);

    // if frame bigger than zero, we animating
    if (gun_animation_frame && gun_animation_timer >=
    gun_animation_duration[gun_animation_frame])
    {
        gun_animation_frame = (gun_animation_frame + 1) % 5;
        gun_animation_timer = 0;
        gun_sprite.setTexture(gun_texs[gun_animation_frame]);
    }


}

void Player::drawDeathScreen(float dt)
{
    if (!dead) return;

    string main_string = killer_name + " killed you";
    sf::Text main(main_string, bold_font, 80);
    main.setOrigin(main.getLocalBounds().width / 2,
    main.getLocalBounds().height / 2);
    main.setPosition(WIDTH / 2, HEIGHT / 2 - 30);
    main.setOutlineColor(sf::Color::Black);
    main.setOutlineThickness(2);
    main.setFillColor(sf::Color(200, 30, 20));
    window.draw(main);

    string sub_string = "press space to respawn";
    sf::Text sub(sub_string, deathscreen_font, 30);
    sub.setOrigin(sub.getLocalBounds().width / 2,
    sub.getLocalBounds().height / 2);
    sub.setPosition(WIDTH / 2, HEIGHT / 2 + 60 );
    window.draw(sub);

}

void Player::shootGun(bool left_click)
{
    if (dead)
        return;
    //if right click or gun is animating, don't shoot
    if (!left_click || gun_animation_frame)
    {
        click_sound.play();
        return;
    }

    gun_shot = true;

    gun_sound.play();

    gun_animation_frame = 1;
    gun_sprite.setTexture(gun_texs[gun_animation_frame]);
    gun_animation_timer = 0;


}


void Player::updateServer()
{

    std::lock_guard<std::mutex> lock(mtx);

    Client::PlayerInfo player_info = getPlayerInfo();


    void* buffer = malloc(sizeof(player_info) + received_events_size);
    int buffer_size = sizeof(player_info) + received_events_size;
    memcpy(buffer, &player_info, sizeof(player_info));

    if (received_events_size)
    {
        memcpy((char*)buffer + sizeof(player_info), received_events,
        received_events_size);

        received_events_size = 0;
    }

    string error;
    if (!client.sendEncryptedUDP(buffer, buffer_size, error))
    {
        cout << "couldn't send udp because: " << error << '\n';
        free(buffer);
        return;
    }


    free(buffer);

}
```

player.cpp

```
void Player::listenToServer()
{

    while (!has_quit)
    {

        string error;
        void* buffer;
        int buffer_size;
        if (!client.recvEncryptedUDP(buffer, buffer_size, error))
        {
            continue;
        }

        //cout << "Got This UDP (with size " << buffer_size << "): ";
        //printBytes(buffer, buffer_size);

        std::lock_guard<std::mutex> lock(mtx);

        char player_count = *(char*)buffer;


        int other_players_count = player_count - 1;
        if (other_players_count != objects.size())
        {

            objects.resize(other_players_count);
            sorted_objects.resize(other_players_count);

            for (int i = 0; i < objects.size(); i++)
            {
                objects[i] = Object(-10, -10, enemy_tex);
                sorted_objects[i] = &objects[i];
            }

        }

        // 1 byte of player_count and then PlayerInfo structs one after
        the other

        int events_byte_count = buffer_size - 1 - player_count *
        sizeof(Client::PlayerInfo);



        // update all others
        int object_index = 0;
        Client::PlayerInfo* current_info_buffer =
        (Client::PlayerInfo*)((char*)buffer + 1);
        for (int i = 0; i < player_count; i++, current_info_buffer++)
        //point to next buffer
        {
            int current_player_id = current_info_buffer->player_id;

            addToLeaderboard(current_player_id,
            current_info_buffer->score, current_info_buffer->username);

            if (current_player_id == client.player_id)
                continue;

            objects[object_index].loadPlayerInfo(*current_info_buffer);

            object_index++;
        }

        updateLeaderboard(-1);

        if (events_byte_count > 0)
            handleEvents((char*)buffer + 1 + player_count *
            sizeof(Client::PlayerInfo), events_byte_count);


        free(buffer);
    }


}
void Player::handleEvents(char* events, int events_size)
{
    bool all_events_intelligible = true;

    int event_size = 0;
    for (int index = 0; index < events_size; index += event_size)
    {
        event_size = *(events + index);

        //cout << "Event: ";
        //printBytes(events + index, event_size);

        int event_type = *(events + index + 1);

        if (event_type == 1) // shooting happened
        {
            int shooter_id = *(char*)(events + index + 2);
            int victim_id = *(char*)(events + index + 3);
            handle_shooting_victim(victim_id, shooter_id);

            if (shooter_id == client.player_id) // i shot
            {
                reticle_timer = 0; // start reticle
            }

        }
        else if (event_type == 2) // death happened
        {
            int killer_id = *(char*)(events + index + 2);
            int victim_id = *(char*)(events + index + 3);
            handle_killing(killer_id, victim_id);

        }
        else if (event_type == 3) // new person joined
        {
            continue;

            int new_guy_id = *(char*)(events + index + 2);
            char* new_guy_username = events + index + 3;

            addToLeaderboard(new_guy_id, 0, new_guy_username);


            if (new_guy_id == client.player_id) continue;



            Object* object = getObject(new_guy_id);
            if (object == nullptr)
            {
                cout << "player with id " << new_guy_id << " not
                found\n";
                all_events_intelligible = false;
                continue;
            }

            object->username = new_guy_username;

            toaster.toast(object->username + " has joined the lobby");


        }
        else if (event_type == 4) // username of someone
        {
            continue;

            int player_id = *(char*)(events + index + 2);
            int score = *(char*)(events + index + 3);
            char* username = events + index + 4;

            addToLeaderboard(player_id, score, username);

            if (player_id == client.player_id) continue;


            cout << username << " is already here.\n";


            Object* object = getObject(player_id);
            if (object == nullptr)
            {
```

```
            cout << "player with id "<<  player_id << " not
            found\n";
            all_events_intelligible = false;
            continue;
        }

        //set the object to the new player
        //object->player_id = player_id;
        object->username = username;
    }
    else if (event_type == 5) // someone left
    {
        continue;
        int player_id = *(events + index + 2);
        cout << "event 5\n";
        //remove from leaderboard
        for(int i = 0; i < leaderboard.size(); i++)
            if (leaderboard[i].player_id == player_id)
            {
                cout << "erasing\n";
                leaderboard.erase(leaderboard.begin() + i);
                break;
            }


        if (player_id == client.player_id) continue;

        Object* object = getObject(player_id);
        if (object == nullptr)
        {
            cout << "leaving player not found\n";
            all_events_intelligible = false;
            continue;
        }

        //toaster.toast(object->username + " has left the lobby");
    }
    else
    {
        cout << "Unrecognized Event: ";
        printBytes(events + index, event_size);
        all_events_intelligible = false;
    }


}

if (all_events_intelligible)
{
    received_events_size = events_size;
    memcpy(received_events, events, events_size);
}
```

```
}

void Player::getKilled(const string& killer_name)
{
    cout << "You got Killed\n";
    dead = true;
    this->killer_name = killer_name;
    current_damage_opacity = max_damage_opacity;
}

void Player::handle_killing(int killer_id, int victim_id)
{
    int verb_index = rand() % 9;
    string killer_name = getUsername(killer_id);
    toaster.toast(killer_name + " " + verbs[verb_index] + " " +
    getUsername(victim_id));

    if (client.player_id == killer_id)
        score++;

    updateLeaderboard(killer_id);

    if (victim_id == client.player_id)
    {
        getKilled(killer_name);
        return;
    }

    //someone else died
    Object* victim = getObject(victim_id);
    if (victim == nullptr)
    {
        cout << "victim not found\n";
        return;
    }

    victim->gotKilled();




}

void Player::handle_shooting_victim(int victim_id, int shooter_id)
{
    // you got shot
    if (victim_id == client.player_id)
    {
        getShot(shooter_id);
        return;
    }

    //someone else got shot
    Object* victim = getObject(victim_id);
    if (victim == nullptr)
    {
        cout << "victim not found\n";
        return;
    }

    victim->gotShot();
}

void Player::getShot(int shooter_id)
{
    Object* shooter = getObject(shooter_id);
    if (shooter == nullptr)
    {
        cout << "shooter not found\n";
        return;
    }

    // turn on hit direction
    float relative_angle = atan2(position.y - shooter->position.y,
    position.x - shooter->position.x);
    hit_direction_timers.push_back(0);
    hit_direction_angles.push_back(relative_angle);

    // turn on damage overlay
    current_damage_opacity = max_damage_opacity;
}

Object* Player::getObject(int id)
{
    for (int i = 0; i < objects.size(); i++)
    {
        if (objects[i].player_id == id)
            return &objects[i];
    }
    return nullptr;
}

Object* Player::getAnyObject()
{
    for (int i = 0; i < objects.size(); i++)
    {
        if (objects[i].player_id == -1)
            return &objects[i];
    }
    return nullptr;
}

Client::PlayerInfo Player::getPlayerInfo()
{
    Client::PlayerInfo info = {
        client.player_id,
        shootRay(0).distance,
        position.x, position.y, rotation_x, rotation_y,
        0,
        score
    };

    info.flags = 0;
    info.flags |= moving * Client::PlayerInfo::moving;
    info.flags |= moving_forward * Client::PlayerInfo::forward;
    info.flags |= gun_shot * Client::PlayerInfo::gun_shot;
    info.flags |= has_quit * Client::PlayerInfo::quit;
    info.flags |= dead * Client::PlayerInfo::dead;

    if (client.username.size() > 15)
        cout << "username too long oh nooooooo\n";

    strcpy_s(info.username, client.username.c_str());

    // reset gun shot flag
    gun_shot = false;

    return info;

}

void Player::respawn()
{
    if (!dead)
        return;

    dead = false;

    float x, y;
    while (true)
    {
        x = (float)rand() / RAND_MAX * map.width;
        y = (float)rand() / RAND_MAX * map.height;
        if (map.getCell(x, y) == 0)
            break;
    }
    rotation_x = (float)rand() / RAND_MAX * 2 * PI;

    position = { x, y };

}

string Player::getUsername(int id)
{
    if (id == client.player_id)
    {
        return client.username;
    }
    Object* victim = getObject(id);
    if (victim == nullptr)
    {
        cout << "Username not found\n";
        return "MISSING USERNAME 2";
    }

    return victim->username;

}

void Player::addToLeaderboard(int player_id, int score, const string&
username)
{
    for (const auto& board : leaderboard)
        if (board.player_id == player_id)
            return;

    cout << "adding player " << player_id << " to leaderboard.\n";
    leaderboard.emplace_back(player_id, score, username);
}

void Player::updateLeaderboard(int killer_id)
{
    for (int i = 0; i < leaderboard.size(); i++)
    {
        if (leaderboard[i].player_id == killer_id)
        {
            leaderboard[i].score++;
            break;
        }
    }

    for (int i = leaderboard.size() - 1 - 1; i >= 0; i--)
    {
        if (leaderboard[i].score < leaderboard[i + 1].score)
        {
            Toaster::LeaderboardEntry temp = leaderboard[i];
            leaderboard[i] = leaderboard[i + 1];
            leaderboard[i + 1] = temp;
        }
    }
}

void Player::debug()
{
    cout << position.x << ": X\n"
        << position.y << ": Y\n"
        << map.floor_level << ": map floor level\n\n";

    debug_mode ^= 1;
}
```