Games Engineering Report

Xiangyi Chen
u5752303

University of Warwick

February 8, 2026

1 Part1 Optimising a Rasterizer

1.1 Introduction

This section implements various optimizations to an initial rasterizer, including optimizations to basic arithmetic operations, using SIMD to process multiple data simultaneously, improving the flow algorithm, and using multithreading for optimization. Ultimately, this results in a significant optimization of the rasterizer.

1.2 Scene3 Design

This scene is designed for multi-threading optimization. It renders six spheres, with the top three spheres constantly rotating. Each sphere has many vertices and triangles, which allows it to effectively leverage the advantages of multi-threading.Figure 1.

```
void scene3(bool if_trans_first = false, bool if_AVX = false){
    // ...
    // 1. Create high-poly spheres
    int LAT = 40;
    int LON = 80;
    for (int i = 0; i < 6; ++i){
        Mesh* s = new Mesh();
        *s = Mesh::makeSphere(2.5f, LAT, LON);

        float x = -6.0f + (i % 3) * 6.0f;
        float y = (i < 3) ? 3.5f : -3.5f;

        s->world = matrix::makeTranslation(x, y, -10.f);
        scene.push_back(s);
    }
    // ...
    while (running)
    {
        // ...
        // Animate only top row
        for (int i = 0; i < 3; ++i)
        {
            scene[i]->world =
                scene[i]->world *
                    matrix::makeRotateXYZ(0.01f, 0.02f, 0.015f);
        }
        //...
        }}
```

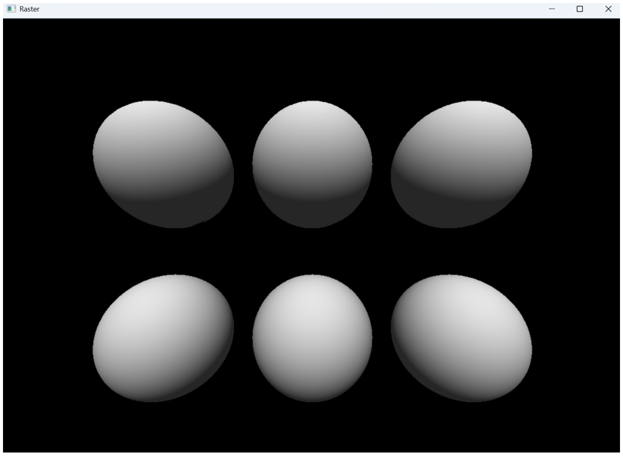


Figure 1: *Scene3*

1.3 Single Thread Optimization

1.3.1 Basic Caculations Optimization

First, I decided to optimize only the basic matrix and vector operations in the existing render process, using SIMD to process the four data of both matrices and vectors at the same time.

However, in my tests, not all operations were suitable for SIMD. When using SIMD multiplication on vectors, the processing speed actually slowed down. This is because the original computational load was small, and the program was already very efficient after compiler optimization. The overhead of SIMD's load and store operations outweighed the time savings. Therefore, in the vector algorithm, I only optimized the normalize function, pre-calculating the division variable to reduce the original three divisions to one division and three multiplications.I also performed similar optimizations on other similar parts that could be optimized in this way.

```
void normalise() {
    float length = std::sqrt(x * x + y * y + z * z);
    float invlength = 1 / length;
    x *= invlength;
    y *= invlength;
    z *= invlength;
}
```

Using SIMD in matrix multiplication can indeed make the operation faster.
The SIMD logic for matrix multiplication works as follows: First, store the information of each row of the right matrix. Since the first row of the final matrix is obtained by multiplying the first row of the left matrix by each column of the right matrix, the process iterates through all four rows of the final matrix. In each iteration, each element of the first row of the left matrix is stored in a ‘m128‘ variable.
For example, the first row of the final matrix stores the first element ‘a0‘ of the first row of the left matrix. Multiplying this by the first row of the right matrix is equivalent to multiplying all elements of the first row of the final matrix in the original calculation. Once the calculation of multiplying the first row of the left matrix by each column of the right matrix is complete, the next step is to multiply the data such as a1, b1..., one by one and sum them up to obtain the data of the first row.
The matrix-vector multiplication method uses the SSE dp command.

```
// SSE matrix multiply: out = A * B
static inline void mul(MatData& out, const MatData& A, const MatData& B) {
    __m128 b0 = _mm_load_ps(&B.a[0]);
    __m128 b1 = _mm_load_ps(&B.a[4]);
    __m128 b2 = _mm_load_ps(&B.a[8]);
    __m128 b3 = _mm_load_ps(&B.a[12]);

    for (int r = 0; r < 4; ++r) {
        __m128 a0 = _mm_set1_ps(A.a[r * 4 + 0]);
        __m128 a1 = _mm_set1_ps(A.a[r * 4 + 1]);
        __m128 a2 = _mm_set1_ps(A.a[r * 4 + 2]);
        __m128 a3 = _mm_set1_ps(A.a[r * 4 + 3]);

        __m128 res0 = _mm_add_ps(_mm_add_ps(_mm_mul_ps(a0, b0),
                    _mm_mul_ps(a1, b1)),
                    _mm_add_ps(_mm_mul_ps(a2, b2),
                        _mm_mul_ps(a3, b3)));

        _mm_store_ps(&out.a[r * 4], res0);
    }
}

// SSE matrix * vector using _mm_dp_ps
static inline void mul(vec4& out, const MatData& A, const vec4& v) {
    __m128 vec = _mm_load_ps(&v[0]);

    for (int row = 0; row < 4; ++row) {
        __m128 matRow = _mm_load_ps(&A.a[row * 4]); // load row

        __m128 dp = _mm_dp_ps(matRow, vec, 0xF1); // mask 0xF1: multiply all, store in lowest
        out[row] = _mm_cvtss_f32(dp); // extract lowest float
    }
}
```

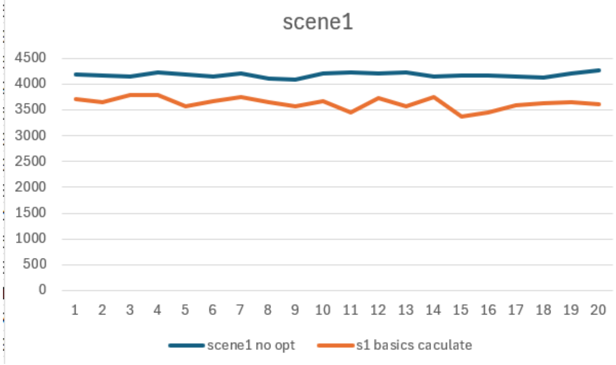


Figure 2: *Scene1 Caculations Optimization*

Secondly, SIMD can also be used in the zbuffer’s clear function to process multiple values at once. This optimization is very significant because zbuffer clear needs to be performed in every render loop.

```
void clear() {
    // could also use fill_n
    __m256 onef = _mm256_set1_ps(1.0f);
    for (unsigned int i = 0; i < width * height; i += 8) {
        _mm256_store_ps(&buffer[i], onef);
    }
}
```

Since the original color class only stores three floats (rgb), I added padding for better memory alignment.

```
class alignas(16) colour_opt {
public:
    union {
        struct {
            float r, g, b; // Red, Green, and Blue components of the colour
            float padding;
        };
        float rgb[4];      // Array representation of the RGB components};};
```

After implementing the above optimizations, the overall performance has been significantly improved. I used data from 20 cycles to make this judgment.(Figure 2.Figure 3.Figure 4.)

1.3.2 Algorithm Optimization

First, there’s the lighting normalization. Since the lighting didn’t change in this case, and normalizing the lighting every draw is unreasonable, I changed the lighting calculation to once per frame.

Secondly, there’s the transformation of the mesh vertices. The algorithm that transforms each triangle’s three vertices sequentially repeatedly calculates

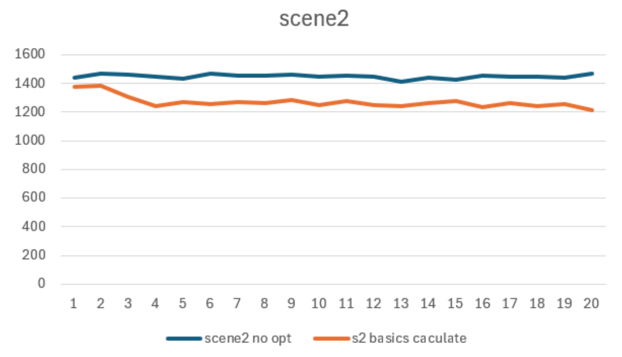


Figure 3: *Scene2 Caculations Optimization*

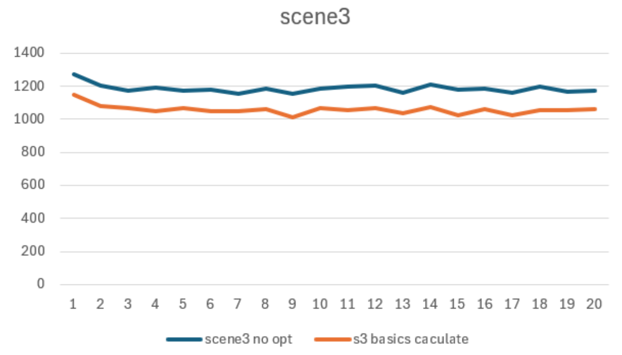


Figure 4: *Scene3 Caculations Optimization*

the common vertices of two triangles, it cause some waste.A better approach is to transform all the vertices in the mesh first, and then draw each triangle one by one.

```
for (int i = 0; i < vCount; ++i){
    Vertex& out = vsCache[i];
    const Vertex& in = mesh->vertices[i];

    // view pos
    view_positions[i] = view * in.p;

    // clip space
    out.p = p * in.p;
    out.p.divideW();

    // viewport transform
    out.p[0] = (out.p[0] + 1.f) * 0.5f * renderer.canvas.getWidth();
    out.p[1] = (out.p[1] + 1.f) * 0.5f * renderer.canvas.getHeight();
    out.p[1] = renderer.canvas.getHeight() - out.p[1];

    // normal (world space)
    out.normal = mesh->world * in.normal;
    out.normal.normalise();

    out.rgb = in.rgb;
}
```

To avoid further waste, I also added a backf culling algorithm. By calculating the normal value of the triangle's plane, this algorithm culls triangles that are away from the camera, thus reducing a significant number of draw calls.

```
for (unsigned int triIdx = 0; triIdx < triCount; ++triIdx) {
    triIndices& ind = mesh->triangles[triIdx];

    vec4& v0_view = view_positions[ind.v[0]];
    vec4& v1_view = view_positions[ind.v[1]];
    vec4& v2_view = view_positions[ind.v[2]];
    vec4 e1 = v1_view - v0_view;
    vec4 e2 = v2_view - v0_view;
    vec4 view_normal = vec4::cross(e1, e2);
    if (vec4::dot(view_normal, -v0_view) >= 0.0f)
        continue;
```

Finally, and most significantly, the improvement is in the optimization of the draw algorithm. The original draw algorithm sequentially calculates whether each pixel is within the triangle's region. If it is, it sequentially calculates the color, normal, and depth before finally calling draw. This process involves numerous multiplications and divisions, resulting in very low efficiency.

A better approach is to use LEE (Linear Expression Evaluation). The principle is that when faced with the same triangle, the functions of its three sides are fixed,

$$E(x,y) = A*x + B*y + C \tag{1}$$

When you substitute a coordinate into this equation, the sign of the result will differ depending on whether the coordinate lies on either side of the line.

Therefore, when you increase the value of x, the result of the entire equation only increases by the size of A.And A is a constant value of the function.

$$E(x,y) = A*(x+1) + B*y + C \tag{2}$$

$$= A*x + A + B*y + C \tag{3}$$

Based on this algorithm, after calculating the value of the top-left corner point to be evaluated, we can use the simplest addition to obtain the alpha, beta, and gamma values of any other point. This will greatly improve the efficiency of the algorithm. Furthermore, based on this algorithm, we can use SIMD AVX2 to process 8 pixels on the screen at once.

To maximize the efficiency of this function, I precalculated many fixed values, thus using space to improve time. This also resulted in a very long function, so I will use a portion of the function for illustration.

The entire draw process involves first calculating each increment in the x and y directions in advance and setting them as the corresponding AVX2 variables.

```
eg.
 const float invArea = 1.0f / area;

 const float dz_dx = (v[0].p[2] * e0.x + v[1].p[2] * e1.x + v[2].p[2] * e2.x) * invArea;
 const float dz_dy = (v[0].p[2] * e0.y + v[1].p[2] * e1.y + v[2].p[2] * e2.y) * invArea;

 const vec4 dn_dx = (v[0].normal * e0.x + v[1].normal * e1.x + v[2].normal * e2.x) * invArea;
const vec4 dn_dy = (v[0].normal * e0.y + v[1].normal * e1.y + v[2].normal * e2.y) * invArea;

const colour dc_dx = (v[0].rgb * e0.x + v[1].rgb * e1.x + v[2].rgb * e2.x) * invArea;
const colour dc_dy = (v[0].rgb * e0.y + v[1].rgb * e1.y + v[2].rgb * e2.y) * invArea;

// --- SIMD constants ---
const __m256 zero = _mm256_setzero_ps();
//const __m256 lane = _mm256_set_ps(7, 6, 5, 4, 3, 2, 1, 0);
const __m256 lane = _mm256_setr_ps(0.0f, 1.0f, 2.0f, 3.0f, 4.0f, 5.0f, 6.0f, 7.0f);
```

After setting the initial value, loop through x and y, and increment the corresponding initial value after each single loop.

```
for (int y = minY; y <= maxY; ++y){
    int x = minX;
    for (; x <= maxX - 7; x += 8) {
    ...
    w0v = _mm256_add_ps(w0v, w0_step8);
    w1v = _mm256_add_ps(w1v, w1_step8);
    w2v = _mm256_add_ps(w2v, w2_step8);
    zv = _mm256_add_ps(zv, z_step8);

    nx = _mm256_add_ps(nx, n_step8x);
    ny = _mm256_add_ps(ny, n_step8y);
    nz = _mm256_add_ps(nz, n_step8z);

    cr = _mm256_add_ps(cr, c_step8r);
    cg = _mm256_add_ps(cg, c_step8g);
    cb = _mm256_add_ps(cb, c_step8b);
    }
w0_row += w0_dy;
w1_row += w1_dy;
w2_row += w2_dy;
z_row += dz_dy;
n_row += dn_dy;
c_row += dc_dy;
}
```

Using "mm256 and ps" command can get the valid mask of 8 values.This will helpful for later caculate.

```
__m256 w0v_zero = _mm256_cmp_ps(w0v, zero, _CMP_GE_OQ);
__m256 w1v_zero = _mm256_cmp_ps(w1v, zero, _CMP_GE_OQ);
__m256 w2v_zero = _mm256_cmp_ps(w2v, zero, _CMP_GE_OQ);
__m256 inside = _mm256_and_ps(w0v_zero, w1v_zero);
inside = _mm256_and_ps(inside, w2v_zero);
int mask = _mm256_movemask_ps(inside);
if(mask == 0)
{
    w0v = _mm256_add_ps(w0v, w0_step8);
    w1v = _mm256_add_ps(w1v, w1_step8);
    w2v = _mm256_add_ps(w2v, w2_step8);
    zv = _mm256_add_ps(zv, z_step8);

    nx = _mm256_add_ps(nx, n_step8x);
    ny = _mm256_add_ps(ny, n_step8y);
    nz = _mm256_add_ps(nz, n_step8z);

    cr = _mm256_add_ps(cr, c_step8r);
    cg = _mm256_add_ps(cg, c_step8g);
    cb = _mm256_add_ps(cb, c_step8b);
    continue;
}
```

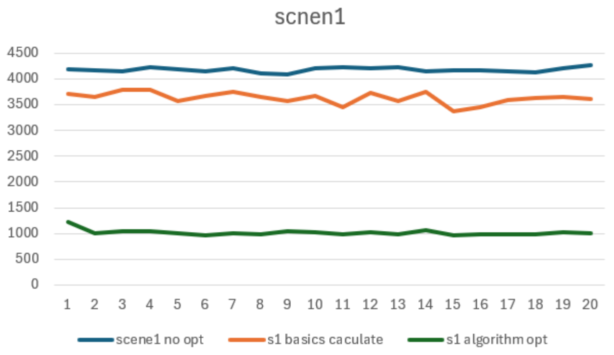


Figure 5: *Scene1 Algorithm Optimization*

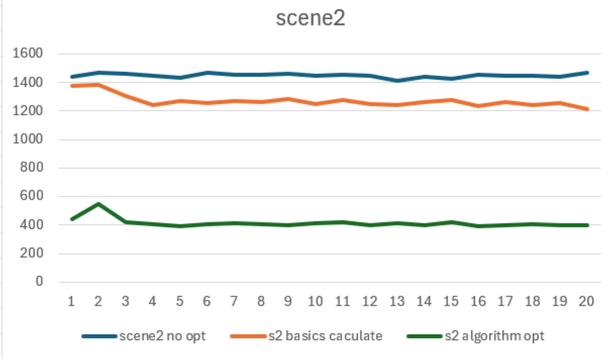


Figure 6: *Scene2 Algorithm Optimization*

When processing eight values where not all lanes are active, the algorithm iterates over the final mask by repeatedly extracting the index of the least significant set bit, processing the corresponding pixel, and then clearing that bit from the mask. This continues until the mask becomes zero, ensuring that only draw valid pixels.

```
int m = final_mask;
while (m)
{
    // count trailing zeros
    int i = _tzcnt_u32(m);
    m &= m - 1;
    renderer.canvas.draw(
        x + i, y,
        (unsigned char)(rr[i]),
        (unsigned char)(gg[i]),
        (unsigned char)(bb[i])
    );
    renderer.zbuffer(x + i, y) = zz[i];
}
```

After implementing the above optimizations, efficiency has been significantly improved. The following data is based on the optimizations in the previous section, using all the optimizations from this section.(Figure 5.Figure 6.Figure 7.)

1.3.3 Data Structures Optimization

When transforming vertices, using SIMD to batch process vertex data can significantly improve efficiency.

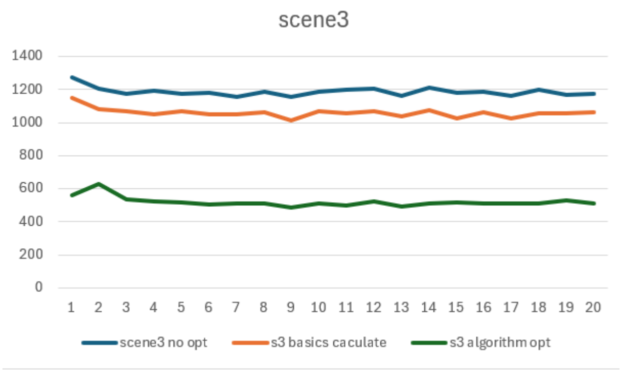


Figure 7: *Scene3 Algorithm Optimization*

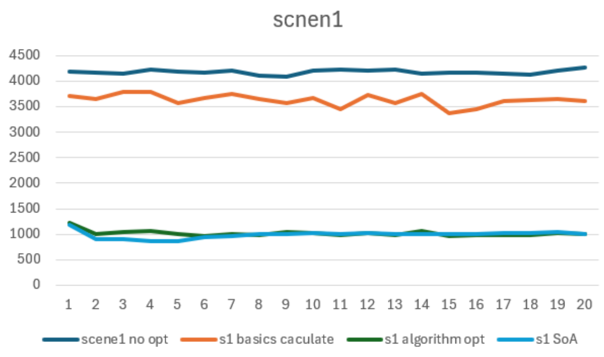


Figure 8: *Scene1 Data Structures Optimization*

Therefore, I redesigned the original AoS mesh structure into a SoA structure.

```
class Mesh_SoA{
public:
colour col;
float kd;
float ka;
matrix world;

alignas(32) std::vector<float> positions_x, positions_y, positions_z, positions_w;
alignas(32) std::vector<float> normals_x, normals_y, normals_z;
alignas(32) std::vector<float> colors_r, colors_g, colors_b;

std::vector<triIndices> triangles;  // List of triangles in the mesh
}
```

This data storage structure allows SIMD to be used to transform vertices to 8 in one go during the vertex transform phase.

```
int i = 0;
for (; i + 7 < vCount; i += 8) {
    // view positon caculate

    // cam position
    __m256 outX = _mm256_add_ps(
        _mm256_add_ps(_mm256_mul_ps(p00, vx),
            _mm256_mul_ps(p01, vy)),
        _mm256_add_ps(_mm256_mul_ps(p02, vz),
            _mm256_mul_ps(p03, vw))
    );
    // y z w

    __m256 invW = _mm256_rcp_ps(outW); // fast approx
    // Newton-Raphson refine for better precision
    invW = _mm256_mul_ps(invW, _mm256_sub_ps(_mm256_set1_ps(2.0f), _mm256_mul_ps(outW, invW)));

    outX = _mm256_mul_ps(outX, invW);
    outY = _mm256_mul_ps(outY, invW);
    outZ = _mm256_mul_ps(outZ, invW);

    // normal matrix transformation
    __m256 nx = _mm256_load_ps(&mesh->normals_x[i]);
    __m256 ny = _mm256_load_ps(&mesh->normals_y[i]);
    __m256 nz = _mm256_load_ps(&mesh->normals_z[i]);

    __m256 outNX = _mm256_add_ps(
        _mm256_add_ps(_mm256_mul_ps(w00, nx),
            _mm256_mul_ps(w01, ny)),
        _mm256_mul_ps(w02, nz)
    );
    // normal y z

    // normalize
    __m256 lenSq = _mm256_add_ps(_mm256_add_ps(_mm256_mul_ps(outNX, outNX),
        _mm256_mul_ps(outNY, outNY)),
        _mm256_mul_ps(outNZ, outNZ));
    __m256 invLen = _mm256_rsqrt_ps(lenSq);
    invLen = _mm256_mul_ps(invLen, _mm256_sub_ps(_mm256_set1_ps(1.5f),
        _mm256_mul_ps(_mm256_set1_ps(0.5f),
            _mm256_mul_ps(lenSq, _mm256_mul_ps(invLen, invLen)))));
    outNX = _mm256_mul_ps(outNX, invLen);
    outNY = _mm256_mul_ps(outNY, invLen);
    outNZ = _mm256_mul_ps(outNZ, invLen);
    // store in vertex cache
    for (int j = 0; j < 8; ++j) {
        Vertex& out = vsCache[i + j];
        out.p = vec4(((float*)&outX)[j], ((float*)&outY)[j], ((float*)&outZ)[j], 1.f);
        out.normal = vec4(((float*)&outNX)[j], ((float*)&outNY)[j], ((float*)&outNZ)[j], 0.f);
        out.rgb.set(mesh->colors_r[i + j], mesh->colors_g[i + j], mesh->colors_b[i + j]);
    }
```

This optimization is not significant in scenes 1 and 2 because the cube has too few vertices. However, for a sphere with a large number of vertices, SIMD's performance improvement is significant, and this can also be seen in the data from scene 3.(Figure 8.Figure 9.Figure 10.)

1.4 Multithreading Strategies

1.4.1 Thread Pool Design

For multithreading optimization, I designed a thread pool to handle tasks. Each task is a Task, and a tasks queue is used to store all unprocessed tasks. The queue and the complete mutex are used to lock the state when adding and completing tasks. 'condition' and

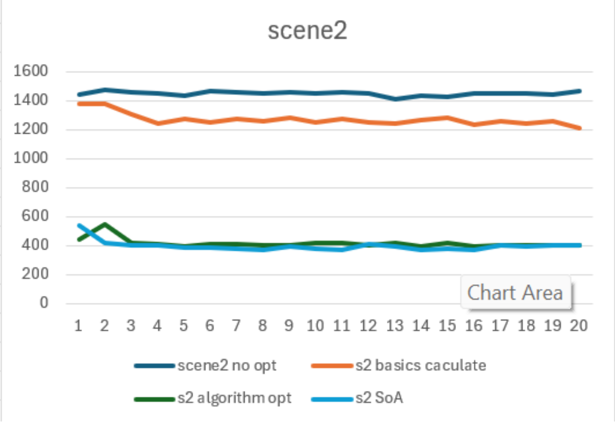


Figure 9: *Scene2 Data Structures Optimization*

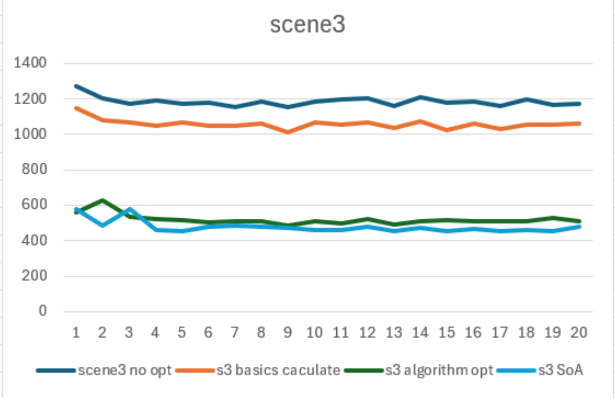


Figure 10: *Scene3 Data Structures Optimization*

'completecv' are two condition variables used to sleep and wake up threads. 'working' and 'tasknum' store the number of currently working tasks and the number of unfinished tasks.

```
class ThreadPool {
private:
    std::vector<std::thread> workers;

    // task queue
    std::queue<std::shared_ptr<Task>> tasks;

    // lock
    mutable std::mutex queue_mutex;
    std::mutex complete_mutex;
    std::condition_variable condition;
    std::condition_variable complete_cv;

    // status flags
    std::atomic<bool> stop;
    std::atomic<int> working;
    std::atomic<int> task_num;
};
```

Th eenqueue() function adds tasks to the task queue. It first locks the queue mutex, then passes the task function to the task queue via an rvalue move, and finally wakes up a thread using notify one.The waitAll() function is waiting all tasks had finished.

```
    // submit task
    void enqueue(std::function<void()> job) {
        {
            std::unique_lock<std::mutex> lock(queue_mutex);
            if (stop) {
                throw std::runtime_error("Cannot enqueue on stopped ThreadPool");
            }
            tasks.emplace(std::make_shared<StdFunctionTask>(std::move(job)));
            ++task_num;
        }
        condition.notify_one();
    }

    // wait for all tasks complete
    void waitAll() {
        std::unique_lock<std::mutex> lock(complete_mutex);
        complete_cv.wait(lock, [this]() {
            return tasks.empty() && working == 0 && task_num == 0;
            });
    }
```

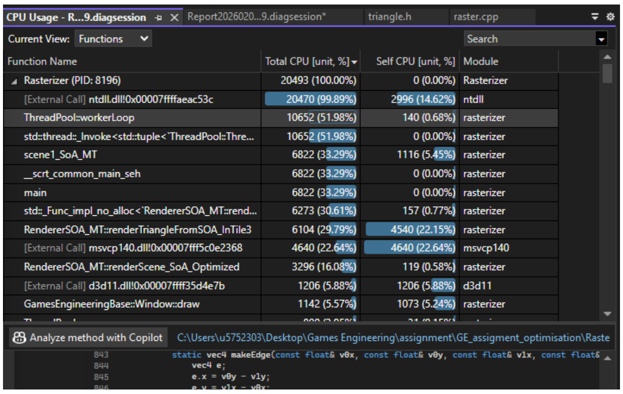
The workerLoop function runs continuously in each worker thread.First, the worker tries to get a task from the task queue.It locks the queue mutex and waits a condition variable if the queue is empty.While waiting, the mutex is released, so other threads can add tasks to the queue. When the worker is woken up, it checks whether the thread pool is stopping and the queue is empty.If it is, the worker exits the loop.If a task is available, the worker removes one task from the queue, increases the number of working tasks, and then releases the lock. After the task has finished, the worker loop locks the queue again, updates the working and task counters, and checks whether all tasks have been completed.If there are no remaining tasks and no task is be working, it notifies the threads that are waiting for all tasks to finish.

```
void workerLoop(int thread_id) {
    while (true) {
        std::shared_ptr<Task> task;
        // get task{
            std::unique_lock<std::mutex> lock(queue_mutex);

            // wait for task
            condition.wait(lock, [this]() {
                return stop || (!tasks.empty());
                });

            if (stop && tasks.empty()) {
                return;
            }

            // get task
            task = std::move(tasks.front());
            tasks.pop();
            working++;
        }
        // do task
        task->execute(); {
            std::unique_lock<std::mutex> lock(queue_mutex);
            working--;
            task_num--;

            // notify all
            if (tasks.empty() && working == 0) {
                std::lock_guard<std::mutex> lock(complete_mutex);
                complete_cv.notify_all();
            }      } }}
```

1.4.2 Solution

To optimize for multithreading, the most suitable place to use multithreading is the final drawing stage. Therefore, in order to divide the task for multiple threads to process, the screen pixels need to be divided into multiple regions, and each thread only calculates the pixels of its own region and draws them. Regarding the tile division logic, my tests show that dividing it into n*n tiles is slower than dividing it into 1*n tiles. That is, the screen is divided into many rows based on the number of threads, with each row assigned to a single thread. Ideally, the number of rows should correspond to the number of threads, thus avoiding excessive thread pool scheduling overhead. When running scene1, the Profiler shows that when using a thread pool, thread invokes consume a significant amount of resources.Figure 11.

In the initial design of the tiles, I planned for each tile to manage its own zbuffer. However, in actual operation, the time consumed by repeatedly clearing the zbuffer of each tile was far greater than the time to clear the zbuffer of the entire canvas. Moreover, modifying the zbuffer data at a specific position using array indexing did not cause data conflicts. Therefore, I did not use the zbuffer of this tile, but instead used the overall zbuffer.



CPU Usage - R...9.diagsession
Report2026020...9.diagsession*
triangle.h
raster.cpp
Current View: Functions
Search
Analyze method with Copilot

Function Name	Total CPU [unit, %]	Self CPU [unit, %]	Module
Rasterizer (PID: 8196)	20493 (100.00%)	0 (0.00%)	Rasterizer
[External Call] ntdll.dll!0x00007ffffaeac53c	20470 (99.89%)	2996 (14.62%)	ntdll
ThreadPool::workerLoop	10652 (51.98%)	140 (0.68%)	rasterizer
std::thread::_Invoke<std::tuple<`ThreadPool::Thre...	10652 (51.98%)	0 (0.00%)	rasterizer
scene1_SoA_MT	6822 (33.29%)	1116 (5.45%)	rasterizer
__scrt_common_main_seh	6822 (33.29%)	0 (0.00%)	rasterizer
main	6822 (33.29%)	0 (0.00%)	rasterizer
std::_Func_impl_no_alloc<`RendererSOA_MT::rend...	6273 (30.61%)	157 (0.77%)	rasterizer
RendererSOA_MT::renderTriangleFromSOA_InTile3	6104 (29.79%)	4540 (22.15%)	rasterizer
[External Call] msvcp140.dll!0x00007fff5c0e2368	4640 (22.64%)	4640 (22.64%)	msvcp140
RendererSOA_MT::renderScene_SoA_Optimized	3296 (16.08%)	119 (0.58%)	rasterizer
[External Call] d3d11.dll!0x00007ffff35d4e7b	1206 (5.88%)	1206 (5.88%)	d3d11
GamesEngineeringBase::Window::draw	1142 (5.57%)	1073 (5.24%)	rasterizer

C:\Users\u5752303\Desktop\Games Engineering\assignment\GE_assigment_optimisation\Raste

```
static vec4 makeEdge(const float& v0x, const float& v0y, const float& v1x, const float&
    vec4 e;
    e.x = v0y - v1y;
```

Figure 11: *Scene1 Multithread Profiler*

```
class Tile {
public:
    int x, y;           // top left coordinates
    int width, height;  // size
    Zbuffer<float> zbuffer;
    std::vector<unsigned char> colors;

    Tile() : x(0), y(0), width(0), height(0) {}

    Tile(int _x, int _y, int _w, int _h) : x(_x), y(_y), width(_w), height(_h) {
        //zbuffer.resize(width * height, std::numeric_limits<float>::max());
        zbuffer.create(width, height);
        colors.resize(width * height * 3, 0);
    }
};

class RendererSOA_MT {
private:
    int TILE_WIDTH;
    int TILE_HEIGHT;
    std::vector<Tile> tiles;
    int tiles_x, tiles_y;
RendererSOA_MT(int canvas_width, int canvas_height, ThreadPool& pool)
: TILE_WIDTH(1024), TILE_HEIGHT(96){
    TILE_HEIGHT = canvas_height / pool.getThreadCount();

    // caculate tiles count
    tiles_x = (canvas_width + TILE_WIDTH - 1) / TILE_WIDTH;
    tiles_y = (canvas_height + TILE_HEIGHT - 1) / TILE_HEIGHT;

    tiles.reserve(tiles_x * tiles_y);
    // create tiles
    for (int ty = 0; ty < tiles_y; ++ty) {
        for (int tx = 0; tx < tiles_x; ++tx) {
            int tile_x = tx * TILE_WIDTH;
            int tile_y = ty * TILE_HEIGHT;
            int tile_w = std::min(TILE_WIDTH, canvas_width - tile_x);
            int tile_h = std::min(TILE_HEIGHT, canvas_height - tile_y);

            tiles.emplace_back(tile_x, tile_y, tile_w, tile_h);
        }
}}};
```

To avoid identifying triangles not within a tile when checking pixels within that tile, a triangle index vector needs to be created for each tile before using multi-threaded drawing. This way, each tile only needs to check for triangles that appear within it.

The principle is to first iterate through all triangles using a for loop to obtain their bounding box coordinates. By dividing by the tile size, we can determine which tile each triangle belongs to in terms of x and y coordinates. Then, we push the triangle's index into the corresponding tile.

```
// triangle binning to tiles
const int triCount = (int)transformed_mesh.triangles.size();
std::vector<std::vector<int>> triangle_buckets(tileCount);

{const float* pos_x = transformed_mesh.transformed_positions_x.data();
const float* pos_y = transformed_mesh.transformed_positions_y.data();
float min_x = std::min({ pos_x[idx.v[0]], pos_x[idx.v[1]], pos_x[idx.v[2]] });
float max_x = std::max({ pos_x[idx.v[0]], pos_x[idx.v[1]], pos_x[idx.v[2]] });
float min_y = std::min({ pos_y[idx.v[0]], pos_y[idx.v[1]], pos_y[idx.v[2]] });
float max_y = std::max({ pos_y[idx.v[0]], pos_y[idx.v[1]], pos_y[idx.v[2]] });

min_x = std::max(0.0f, min_x);
max_x = std::min((float)renderer.canvas.getWidth() - 1, max_x);
min_y = std::max(0.0f, min_y);
max_y = std::min((float)renderer.canvas.getHeight() - 1, max_y);
if (min_x > max_x || min_y > max_y) continue;

int min_tx = int(std::floor(min_x)) / TILE_WIDTH;
int max_tx = int(std::ceil(max_x)) / TILE_WIDTH;
int min_ty = int(std::floor(min_y)) / TILE_HEIGHT;
int max_ty = int(std::ceil(max_y)) / TILE_HEIGHT;

min_tx = std::max(0, min_tx);
max_tx = std::min(tiles_x - 1, max_tx);
min_ty = std::max(0, min_ty);
max_ty = std::min(tiles_y - 1, max_ty);

for (int ty = min_ty; ty <= max_ty; ++ty) {
    for (int tx = min_tx; tx <= max_tx; ++tx) {
        int tile_idx = ty * tiles_x + tx;
        triangle_buckets[tile_idx].push_back(t);
    }
}}
```

Once these operations are complete, each tile can be sent as a task to the thread.

```
// draw
for (int tile_idx = 0; tile_idx < tileCount; ++tile_idx) {
    if (!triangle_buckets[tile_idx].empty()) {
        pool.enqueue([&, tile_idx]() {
            Tile& tile = tiles[tile_idx];
            // tile.zbuffer.clear();
            // std::fill(tile.colors.begin(), tile.colors.end(), 0);
            for (int tri_idx : triangle_buckets[tile_idx]) {
                const triIndices& ind = transformed_mesh.triangles[tri_idx];
                // clip out of the screen
                float z0 = transformed_mesh.transformed_positions_z[ind.v[0]];
                float z1 = transformed_mesh.transformed_positions_z[ind.v[1]];
                float z2 = transformed_mesh.transformed_positions_z[ind.v[2]];
                if (!(fabs(z0) > 1.0f || fabs(z1) > 1.0f || fabs(z2) > 1.0f)) {
                    renderTriangleFromSOA_InTile3( renderer, transformed_mesh, light, ka, kd, tri_idx, tile );
                }       }       }); }
}
pool.waitAll();
```

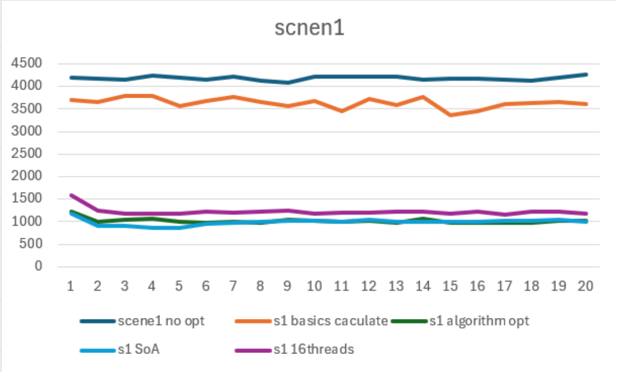


Figure 12: *Scene1 Mutithreading Optimization*

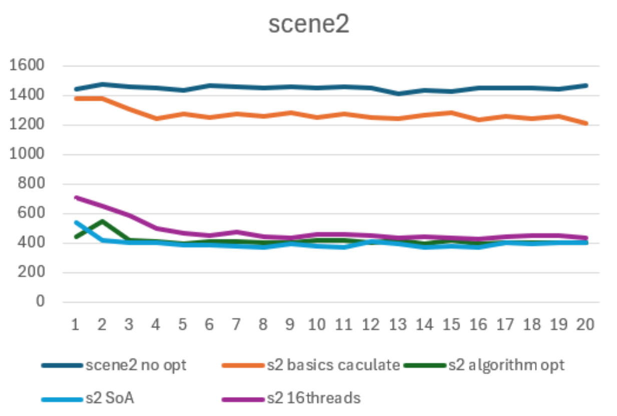


Figure 13: *Scene2 Mutithreading Optimization*

1.4.3 Outcomes

For the multi-threaded version, scenes 1 and 2 are slower than the single-threaded version. My analysis is that the number of vertices and triangles in scenes 1 and 2 is too small, resulting in a small amount of computation required for each frame. Therefore, the thread scheduling overhead of dividing them into multiple tasks and assigning them to a thread pool is greater than the benefits of multi-threading. (Figure 12.Figure 13.)

Moreover, the performance improvement brought about by using different numbers of threads is minimal.Figure 14.

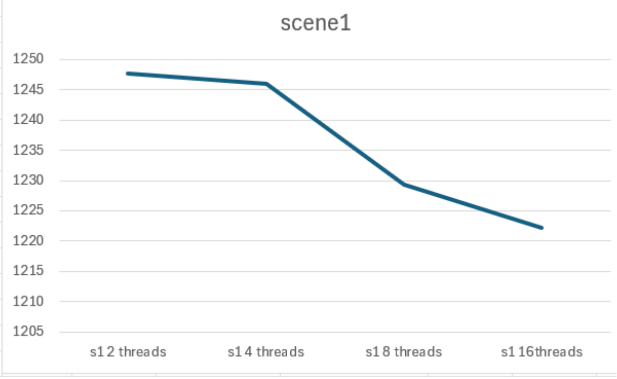


Figure 14: *Scene1 Using different Number of Thread*

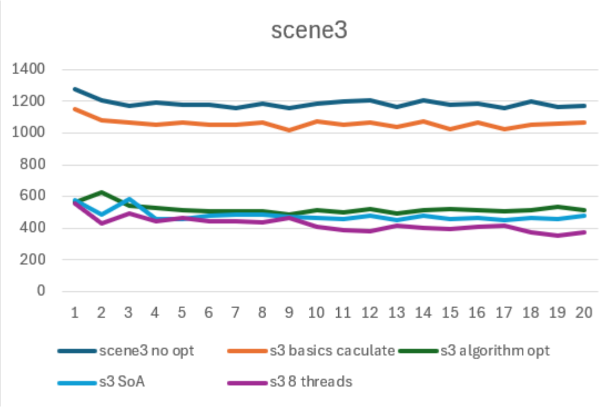


Figure 15: *Scene3 Mutithreading Optimization*

However, in my own design of scene 3, because of the large number of vertices and triangles, multithreading can also be used to handle the vertex transformation and triangle binning stages.

Furthermore, the performance improvement from multithreading is significant.(Figure 15.Figure 17.) The settings only change the "Enabel Enhanced Instruction Set" to "Advanced Vector Extensions 2 (X86/X64) (/arch:AVX2)". Figure 18.

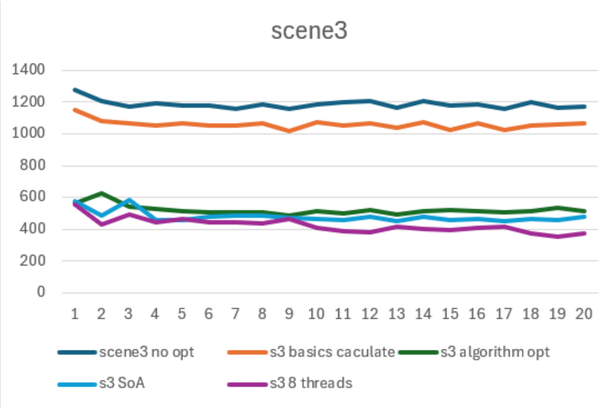
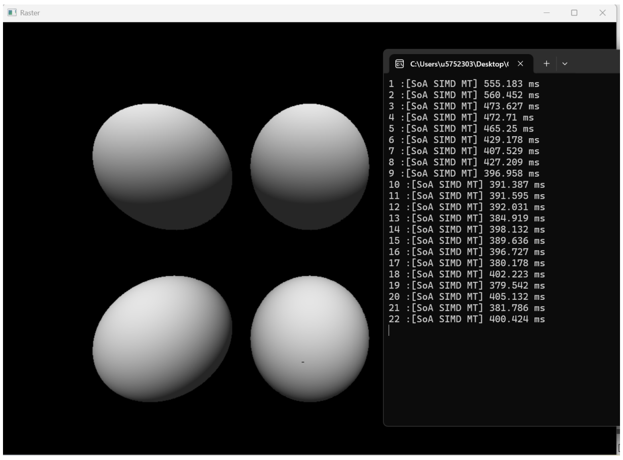
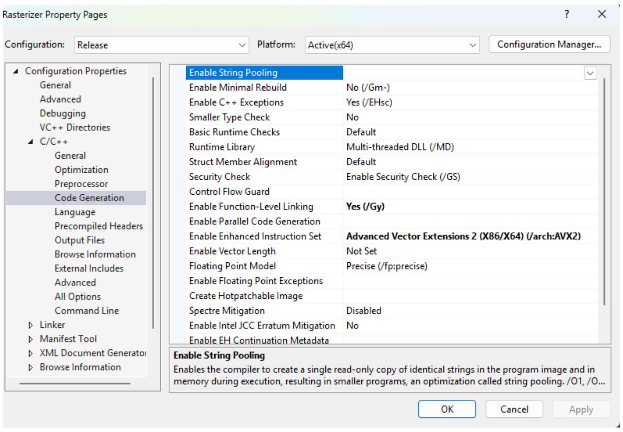


Figure 16: *Scene3 Using different Number of Thread*



```
1 :[SoA SIMD MT] 555.183 ms
2 :[SoA SIMD MT] 560.452 ms
3 :[SoA SIMD MT] 473.627 ms
4 :[SoA SIMD MT] 472.71 ms
5 :[SoA SIMD MT] 465.25 ms
6 :[SoA SIMD MT] 429.178 ms
7 :[SoA SIMD MT] 407.529 ms
8 :[SoA SIMD MT] 427.209 ms
9 :[SoA SIMD MT] 396.958 ms
10 :[SoA SIMD MT] 391.387 ms
11 :[SoA SIMD MT] 391.595 ms
12 :[SoA SIMD MT] 392.031 ms
13 :[SoA SIMD MT] 384.919 ms
14 :[SoA SIMD MT] 398.132 ms
15 :[SoA SIMD MT] 389.636 ms
16 :[SoA SIMD MT] 396.727 ms
17 :[SoA SIMD MT] 380.178 ms
18 :[SoA SIMD MT] 402.223 ms
19 :[SoA SIMD MT] 379.542 ms
20 :[SoA SIMD MT] 405.132 ms
21 :[SoA SIMD MT] 381.786 ms
22 :[SoA SIMD MT] 400.424 ms
```

Figure 17: *Scene3 Mutithreading Working Output*



Configuration: Release

Platform: Active(x64)

Enable String Pooling	
Enable Minimal Rebuild	No (/Gm-)
Enable C++ Exceptions	Yes (/EHsc)
Smaller Type Check	No
Basic Runtime Checks	Default
Runtime Library	Multi-threaded DLL (/MD)
Struct Member Alignment	Default
Security Check	Enable Security Check (/GS)
Control Flow Guard	
Enable Function-Level Linking	**Yes (/Gy)**
Enable Parallel Code Generation	
Enable Enhanced Instruction Set	**Advanced Vector Extensions 2 (X86/X64) (/arch:AVX2)**
Enable Vector Length	Not Set
Floating Point Model	Precise (/fp:precise)
Enable Floating Point Exceptions	
Create Hotpatchable Image	
Spectre Mitigation	Disabled
Enable Intel JCC Erratum Mitigation	No
Enable EH Continuation Metadata	

Enable String Pooling
Enables the compiler to create a single read-only copy of identical strings in the program image and in memory during execution, resulting in smaller programs, an optimization called string pooling. /O1, /O...

Figure 18: *Settings*

1.5 Conclusion

In this rasterizer optimization, the most significant improvement was in the algorithm. A good algorithm can fundamentally improve operational efficiency. The second most important factor was the design of data storage and retrieval logic. However, considering multithreading, these designs become even more complex.

In this project, I implemented some extreme optimizations, but these optimizations are almost exclusively applicable to this specific scenario. For multi-threaded optimization strategies, there are actually better SoA storage methods, such as dividing every 8 vertices into a block. This allows for more efficient to create tasks when processing multiple threads, improving memory storage logic and thus increasing cache hit rate. Given more time, I might explore better data structures and algorithms, also a better multithreading task solution.

2 Part2 Chat Room

2.1 Introduction

This section implements a simple online chat room. Once the server program is running, it can accept connection requests from clients. After a successful connection, it can receive messages sent by the client and forward them to other users. The client program includes a user interface, allowing users to connect to the server and send messages to other users via a fixed IP address and port. It includes a public chat room and a private chat room for individual users.

2.2 Chat Room Server

2.2.1 Network Protocol

In order for the server to correctly parse the type of information sent by the client, both parties need to customize some rules to distinguish different information types. Before sending a message, a message header needs to be sent to inform the recipient what type of message it is. The recipient then processes the message based on its type.

```
enum class MessageType {
    // connect to the server
    CLIENT_CONNECT = 1,
    CLIENT_DISCONNECT = 2,
    // public and private message
    PUBLIC_MESSAGE = 3,
    PRIVATE_MESSAGE = 4,
    // userlist message for update the online user
    USER_LIST_UPDATE = 5,
};
struct MessageHeader {
    MessageType type;

    MessageHeader() : type(MessageType::CLIENT_CONNECT) {}
    MessageHeader(MessageType t, unsigned int s) : type(t) {}
};
```

Different message types require different message formats. Client-connect messages only require the client to pass its own username to the server. Public messages require sending the sender's name and content. Private messages require sending both sender and receiver information. Userlist messages require sending the usernames and number of all users.

```
struct ClientConnectMessage {
    char username[32];
};
struct PublicMessage {
    char sender[32];
    char content[256];
};
struct PrivateMessage {
    char sender[32];
    char target[32];
    char content[256];
};
struct UserListMessage {
    int user_count;
    char users[32][32];
};
```

2.2.2 Server Design

The "serversocket" store the server's socket to linsten the connect from client.The "clients" store the sockets for each user and its name.The "clientsmutex" is used to lock the clients map to prevent other threads from modifying the clients map and causing conflicts. The "running" is used to control the life of the server, when it is false, all while loop will stop and the thread will be released.

The "init" function will create the server socket for listening the client's connection request.First initialize the WinSock, then create socket, bind the ip address and port, then set the socket listen the connect request. After these, create a thread to handle client connections ,and use detach() to make this thread run at background, when close the socket, the while loop in thread will break, thread will be release.

```
class ChatServer {
public:
    SOCKET server_socket;
    std::unordered_map<SOCKET, std::string> clients;
    std::mutex clients_mutex;
    bool running;

    bool init(int port) {
        // Step 1: Initialize WinSock
        WSADATA wsaData;
        if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
            std::cerr << "WSAStartup failed with error: " << WSAGetLastError() << std::endl;
            return false;
        }
        // create socket
        server_socket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
        if (server_socket == INVALID_SOCKET) {
            std::cerr << "Socket creation failed " << std::endl;
            WSACleanup();
            return false;
        }
        // bind port and address
        sockaddr_in serverAddr;
        serverAddr.sin_family = AF_INET;
        serverAddr.sin_port = htons(port);
        serverAddr.sin_addr.s_addr = INADDR_ANY;
        if (bind(server_socket, (sockaddr*)&serverAddr, sizeof(serverAddr)) == SOCKET_ERROR) {
            std::cerr << "Bind failed" << std::endl;
            closesocket(server_socket);
            WSACleanup();
            return false;
        }
        if (listen(server_socket, SOMAXCONN) == SOCKET_ERROR)  {
            std::cerr << "Listen failed" << std::endl;
            closesocket(server_socket);
            WSACleanup();
            return false;
        }
        running = true;
        std::cout << "Chat Server started on port " << port << std::endl;

        std::thread
        // detach thread, when close the server, the while loop
        // in thread will break, thread will be release
        acceptThread(&ChatServer::accept_client, this);
        acceptThread.detach();
        return true;
    }
};
```

The accept client function is a while loop to accept the client's connection request, if accept failed, it will continue to next loop.After connecting the client, create a new thread to handle this client's messages.

```
void accept_client() {
    while (running) {
        sockaddr_in client_address;
        int len = sizeof(client_address);
        SOCKET client_socket = accept(server_socket, (sockaddr*)&client_address, &len);
        if (client_socket == INVALID_SOCKET)  {
            if (running)
                std::cerr << "Accept failed" << std::endl;
            continue;
        }
        char clientIP[INET_ADDRSTRLEN];
        inet_ntop(AF_INET, &client_address.sin_addr, clientIP, INET_ADDRSTRLEN);
        std::cout << "Accept connection from: " << clientIP << ":" << ntohs(client_address.sin_port) << std::endl;
        // create client thread
        std::thread clientThread(&ChatServer::handle_client, this, client_socket);
        clientThread.detach();
    }}
```

The handle client function is used to parse the message content sent by the client and perform correspond operations. After the server and client build the connections, the client needs to send a connection message to tell the server what it's username. First, the server will receive the size of header's size and check if it is a connect message header, if it is, then receive the connect message to get the username, the server will use this client's socket and username to store them in the clients map. After this, the server will send a userlist message to this client and all clients, and send a public message to tell all clients that a new client had joined the chat room.

```
int receive_message(SOCKET socket, char* buffer, int size){
    int totalReceived = 0;
    while (totalReceived < size) {
        int received = recv(socket, buffer + totalReceived, size - totalReceived, 0);
        if (received <= 0) {
            return received;}
        totalReceived += received;}
    return totalReceived;
}
void handle_client(SOCKET client_socket) {
    std::string username;
    MessageHeader header;
    if (receive_message(client_socket, (char*)&header, sizeof(header)) != sizeof(header)) {
        std::cout << "Failed to receive header" << std::endl;
        close_client(client_socket, username);
        return;}
    if (header.type != MessageType::CLIENT_CONNECT)  {
        close_client(client_socket, username);
        return; }

    ClientConnectMessage connect_message;
    if (receive_message(client_socket, (char*)&connect_message, sizeof(connect_message)) != sizeof(connect_message)) {
        std::cout << "Failed to receive connect message" << std::endl;
        close_client(client_socket, username);
        return; }

    username = connect_message.username;
    // create client
    {
        std::lock_guard<std::mutex> lock(clients_mutex);
        clients[client_socket] = username;
    }

    std::cout << "User " << username << " joined the room" << std::endl;

    // send to new user
    send_userlist(client_socket);
    // send a public message to all user
    PublicMessage message("System", username + " joined the chat");
    MessageHeader send_header(MessageType::PUBLIC_MESSAGE, sizeof(PublicMessage));

    {
        std::lock_guard<std::mutex> lock(clients_mutex);
        for (const auto& client : clients) {
            // not send to myself
            if (client.second != username) {
                send(client.first, (char*)&send_header, sizeof(send_header), 0);
                send(client.first, (char*)&message, sizeof(message), 0);
            }
        }
    }
    broadcast_userlist();

    while (running) {
    // information processing logic
  }
    close_client(client_socket, username);}
```

Then using a while loop to process the message from this client. First, receive the header, if it is a public message type, create a new public message struct and receive the content from client and store in this struct, then for loop all clients to send the this public message. If it is a private message, the server will search if the target client in the clients map, if the user is found, a message is sent only to that user; otherwise, the loop continues. If the message is disconnect message, break the loop and close the client.

```
while (running) {
    if (receive_message(client_socket, (char*)&header, sizeof(header)) != sizeof(header)) {
        std::cout << "Client '" << username << "' disconnected" << std::endl;
        break;
    }

    if (header.type == MessageType::PUBLIC_MESSAGE) {
        PublicMessage message;

        if (receive_message(client_socket, (char*)&message, sizeof(message)) != sizeof(message)) {
            std::cout << "Failed to receive public message from " << username << std::endl;
            break;
        }

        std::cout << "Public message from " << message.sender << ": " << message.content << std::endl;

        MessageHeader header(MessageType::PUBLIC_MESSAGE, sizeof(PublicMessage));

        for (const auto& client : clients) {
            send(client.first, (char*)&header, sizeof(header), 0);
            send(client.first, (char*)&message, sizeof(message), 0);
        } }
    else if (header.type == MessageType::PRIVATE_MESSAGE) {
        PrivateMessage message;

        if (receive_message(client_socket, (char*)&message, sizeof(message)) != sizeof(message)) {
            std::cout << "Failed to receive private message from " << username << std::endl;
            break;
        }

        std::cout << "Private message from " << message.sender << " to " << message.target << std::endl;

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

        // search target
        SOCKET targetSocket = INVALID_SOCKET;
        for (const auto& client : clients) {
            if (client.second == message.target) {
                targetSocket = client.first;
                break;
            }
        }

        if (targetSocket != INVALID_SOCKET) {
            MessageHeader header(MessageType::PRIVATE_MESSAGE, sizeof(PrivateMessage));
            send(targetSocket, (char*)&header, sizeof(header), 0);
            send(targetSocket, (char*)&message, sizeof(message), 0);
        }}
    else if (header.type == MessageType::CLIENT_DISCONNECT) {
        std::cout << "Client " << username << " requested disconnect" << std::endl;
        break;
    }
    else {
        std::cout << " unknown " << username << std::endl;
    }}
```

The send userlist function uses for loop collect all clients' name and storee in the UserListMessage. Then send to all clients or one client.

```
void send_userlist(SOCKET target) {
    std::lock_guard<std::mutex> lock(clients_mutex);

    UserListMessage list;
    list.user_count = 0;

    // collect all username
    for (const auto& client : clients) {
        strncpy_s(list.users[list.user_count], client.second.c_str(), sizeof(list.users[list.user_count]) - 1);
        list.users[list.user_count][sizeof(list.users[list.user_count]) - 1] = '\0';
        list.user_count++;
    }
    MessageHeader header(MessageType::USER_LIST_UPDATE, sizeof(UserListMessage));

    send(target, (char*)&header, sizeof(header), 0);
    send(target, (char*)&list, sizeof(list), 0);
}
```

The close function set running = false, then all threads' while loop will stop and release.Close all sockets.

```
void close() {
    running = false;

    // close  linsten socket
    if (server_socket != INVALID_SOCKET) {
        closesocket(server_socket);
        server_socket = INVALID_SOCKET;
    }

    // close all clients {
        std::lock_guard<std::mutex> lock(clients_mutex);
        for (auto& client : clients)
        {
            closesocket(client.first);
        }
        clients.clear();
    }
    WSACleanup();
    std::cout << "Server stopp" << std::endl;
}
```

2.3 Chat Room Client

2.3.1 Client Logic Design

The client interacts with the server through an event handling mechanism. When the server sends a message to the client, the client parses the message type and creates an event of that type. During the main loop's while loop, the ‘process event‘ function is executed in each loop iteration to handle these events, allowing message reception and UI rendering to operate separately.

The “DISCONNECTED" type used to handle disconnection events sent by the server. The ” PUBLICMESSAGE" and "PRIVATEMESSAGE" are used to handle public and private messages. The USER_LIST_UPDATE is used to handle when new client join the chat room, update the online userlist.

```
enum class NetworkEventType {
    CONNECTED = 0, // didn't use
    DISCONNECTED,
    PUBLIC_MESSAGE,
    PRIVATE_MESSAGE,
    USER_LIST_UPDATE
};
struct NetworkEvent {
    NetworkEventType type;
    std::string sender;
    std::string text;
    std::string target;
    std::vector<std::string> users;

    NetworkEvent() : type(NetworkEventType::CONNECTED) {}
    NetworkEvent(NetworkEventType t) : type(t) {}
};
```

The whole chat room use a ChatWindow class, the username store the client's username, users online store the all online users' name.

The "public message" and "private chat" store the message content from this and other clients. They use the ChatMessage struct to store one message,then the UI can render different color for other clients. "public input" and "private input" are used to store messages sent from the chat box. "client socket" is the socket to connect the server,"recieve thread" is used to receive the message from server, and store them as a event in the "event queue"."event mutex" is used to lock the queue when processing and adding event. "system" is used for play sound.

```
struct ChatMessage {
    std::string sender;
    std::string target;
    std::string text;
    bool isPrivate;
    ChatMessage(const std::string& s = "", const std::string& t = "", bool priv = false, const std::string& tar = "")
    : sender(s), text(t), isPrivate(priv), target(tar) {}
};
class ChatWindow {
public:
    std::string username;
    std::vector<std::string> users_online;
    std::vector<ChatMessage> public_message;
    std::map<std::string, std::vector<ChatMessage>> private_chat;

    // input buffer
    char public_input[200];
    std::map<std::string, std::array<char, 200>> private_input;

    bool connected;

    // own socket
    SOCKET client_socket;
    std::thread recieve_thread;
    std::atomic<bool> running;
    std::queue<NetworkEvent> event_queue;
    std::mutex event_mutex;

    FMOD::System* system;};
```

The connect server function will connect to the server using the IP address and port passed as function parameters.After connected succeeds, send a connect message to server to tell the server client's username. Then start a thread to receive the message form server.

```
bool ChatWindow::connect_server(const std::string& ip, int port, const std::string& user_name) {
    // Step 1: Initialize WinSock
    WSADATA wsaData;
    if (WSAStartup(MAKEWORD(2, 2), &wsaData) != 0) {
        std::cerr << "WSAStartup failed with error: " << WSAGetLastError() << std::endl;
        return false;
    }

    // Step 2: Create a socket
    client_socket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if (client_socket == INVALID_SOCKET) {
        std::cerr << "Socket creation failed with error: " << WSAGetLastError() << std::endl;
        WSACleanup();
        return false;
    }
    // set server address
    sockaddr_in serverAddr;
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(port);
    if (inet_pton(AF_INET, ip.c_str(), &serverAddr.sin_addr) <= 0) {
        std::cout << "Invalid address" << std::endl;
        if (client_socket != INVALID_SOCKET) {
            closesocket(client_socket);
            client_socket = INVALID_SOCKET;
        }
        WSACleanup();
        return false;
    }
    // connect server
    if (connect(client_socket, (sockaddr*)&serverAddr, sizeof(serverAddr)) == SOCKET_ERROR)  {
        std::cout << "connect failed" << std::endl;
        if (client_socket != INVALID_SOCKET) {
            closesocket(client_socket);
            client_socket = INVALID_SOCKET;
        }
        WSACleanup();
        return false;
    }

    ClientConnectMessage connect_message(user_name);
    if (!send_message_toserver(MessageType::CLIENT_CONNECT, &connect_message, sizeof(connect_message))){
        std::cerr << "Failed to send connect message" << std::endl;
        if (client_socket != INVALID_SOCKET) {
            closesocket(client_socket);
            client_socket = INVALID_SOCKET;
        }
        WSACleanup();
        return false;
    }

    username = user_name;
    connected = true;
    running = true;

    // receive thread
    recieve_thread = std::thread(&ChatWindow::recive_message, this);

    public_message.push_back(ChatMessage("System", "Connect to chat server"));
    return true;
}
```

The receive message function use a for loop to process the message from server, it will receive the header first, Then create an event to handle this message,after pushed the event into the queue, the corresponding notification tone is played based on the sender's type.

```
void ChatWindow::recive_message() {
    while (running && client_socket != INVALID_SOCKET) {
        MessageHeader header;
        int received = recv(client_socket, (char*)&header, sizeof(header), 0);
        if (received <= 0) {
            NetworkEvent event(NetworkEventType::DISCONNECTED);

            event_queue.push(event);
            break;
        }

        NetworkEvent event;
        switch (header.type) {
        case MessageType::PUBLIC_MESSAGE:  {
            PublicMessage message;
            recv(client_socket, (char*)&message, sizeof(message), 0);

            event.type = NetworkEventType::PUBLIC_MESSAGE;
            event.sender = message.sender;
            event.text = message.content;
            break;
        }
        case MessageType::PRIVATE_MESSAGE: {
            PrivateMessage private_message;
            recv(client_socket, (char*)&private_message, sizeof(private_message), 0);

            event.type = NetworkEventType::PRIVATE_MESSAGE;
            event.sender = private_message.sender;
            event.target = private_message.target;
            event.text = private_message.content;
            break;
        }
        case MessageType::USER_LIST_UPDATE: {
            UserListMessage userlist;
            recv(client_socket, (char*)&userlist, sizeof(userlist), 0);

            event.type = NetworkEventType::USER_LIST_UPDATE;
            for (int i = 0; i < userlist.user_count; i++) {
                event.users.push_back(userlist.users[i]);
            }
            break;
        }
        default:
            continue;
        }
        {
            std::lock_guard<std::mutex> lock(event_mutex);
            event_queue.push(event);
        }
        // if not my message,paly sound
        if (event.sender != username)
            ChatWindow::play_music(event.type);
    }}
```

The process event function will loop all events, before get an event, lock the queue lock at first. If the type is DISCONNECTED, close the socket and clear all online users, and push a public message at the chat to show connect lost. If the type is "PUBLIC MESSAGE", just push the message into the public message vector. If the type is "PRIVATE MESSAGE", it will judge if the sender is this client's name to avoid creating your own chat box. Then find the sender's name in private chat map, if didn't found, then create one, and push the message into this chat's message vector.It will also create a new chat input buffer. If the type is "USER LIST UPDATE", it will update the local online userlist.

```
void ChatWindow::process_event() {
    while (!event_queue.empty()) {
        // get event
        NetworkEvent event;
        {
            std::lock_guard<std::mutex> lock(event_mutex);
            event = event_queue.front();
            event_queue.pop();
        }

        switch (event.type) {
        case NetworkEventType::DISCONNECTED:
            connected = false;
            close_connect();
            public_message.push_back(ChatMessage("System", "Connect lost"));
            // clear all users
            users_online.clear();
            if (!username.empty()) {
                users_online.push_back(username);
            }
            break;

        case NetworkEventType::PUBLIC_MESSAGE:
            if (event.sender != username)
            public_message.push_back(ChatMessage(event.sender, event.text));
            break;

        case NetworkEventType::PRIVATE_MESSAGE:{
            std::string name = (event.sender == username) ? event.target : event.sender;

            // check private chat map
            if (private_chat.find(name) == private_chat.end()){
                private_chat[name] = std::vector<ChatMessage>();
            }
            // add chat message
            ChatMessage mess(event.sender, event.text, true, event.target);
            private_chat[name].push_back(mess);

            if (private_input.find(name) == private_input.end()){
                private_input[name].fill('\0');
            }
        }
            break;
        case NetworkEventType::USER_LIST_UPDATE:
            // when someone link the server
            update_userlist(event.users);
            break;
        default:
            break;
        } }}
```

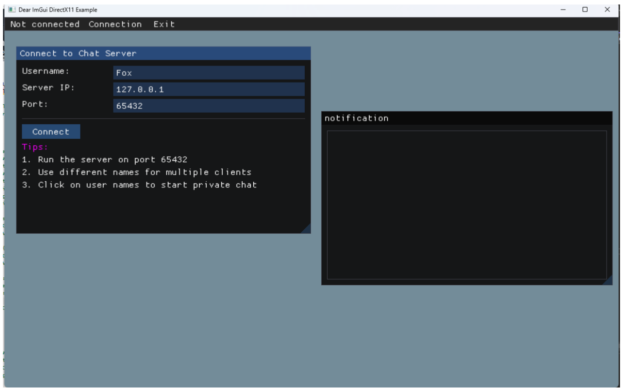
2.3.2 Client UI Design

The main loop will call renderall function to render all window. The notification window will show all local public messages even if the client lost the connection from the server. If not connected, render the login window, if connected, draw the main chat room windows.

```
void ChatWindow::render_all() {
    process_event();
    if (!connected) {
        // login
        login_win();
        // Display some public messages
        ImGui::SetNextWindowSize(ImVec2(600, 400), ImGuiCond_FirstUseEver);
        ImGui::SetNextWindowPos(ImVec2(600, 350), ImGuiCond_FirstUseEver);
        if (ImGui::Begin("notification", nullptr, ImGuiWindowFlags_NoCollapse)) {
            ImGui::BeginChild("Messages", ImVec2(0, 0), true);
            for (const auto& message : public_message) {
                if (message.sender == "System") {
                    ImGui::PushStyleColor(ImGuiCol_Text, ImVec4(1.0f, 0.5f, 0.0f, 1.0f));
                }
                else {
                    ImGui::PushStyleColor(ImGuiCol_Text, ImVec4(0.5f, 0.5f, 0.5f, 1.0f));}

                ImGui::Text("%s: %s", message.sender.c_str(), message.text.c_str());
                ImGui::PopStyleColor();
            }
            ImGui::EndChild();
        }
        ImGui::End();
    }
    else {
        // main
        ImGui::SetNextWindowSize(ImVec2(800, 600), ImGuiCond_FirstUseEver);
        ImGui::Begin("Chat Room", nullptr, ImGuiWindowFlags_NoCollapse);

        // user
        user_win();
        ImGui::SameLine();

        // chat
        ImGui::BeginGroup();
        chat_win();
        ImGui::EndGroup();

        ImGui::End();

        // private
        private_win();
    } }
```

The login window sets the initial values to connect the server. However, if you want to use a different username, please change the username before logging in. The input box supports pressing Enter to connect. Clicking "connect" will call the connectserver function to connect to the server.Figure 19 . If the connection fails, the failure message will be pushed to the public message and displayed in the notification window.Figure 20.

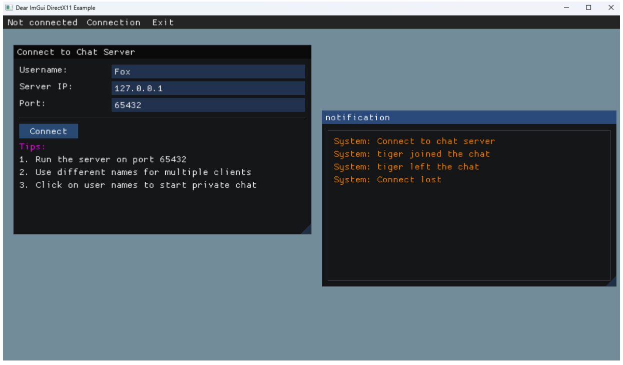
```
void ChatWindow::login_win() {
    ImGui::SetNextWindowPos(ImVec2(100, 100), ImGuiCond_FirstUseEver);
    ImGui::SetNextWindowSize(ImVec2(500, 500), ImGuiCond_FirstUseEver);
    bool connect = false;

    if (ImGui::Begin("Connect to Chat Server", nullptr, ImGuiWindowFlags_NoCollapse)) {

        static char username[64] = "Fox";
        static char serverIP[64] = "127.0.0.1";
        static char port[32] = "65432";

```



Username: Fox
Server IP: 127.0.0.1
Port: 65432

Tips:
1. Run the server on port 65432
2. Use different names for multiple clients
3. Click on user names to start private chat

Figure 19: *The login in window for connect the server.*



Username: Fox
Server IP: 127.0.0.1
Port: 65432

Tips:
1. Run the server on port 65432
2. Use different names for multiple clients
3. Click on user names to start private chat

System: Connect to chat server
System: tiger joined the chat
System: tiger left the chat
System: Connect lost

Figure 20: *Lost the connection with the server.*

```
        ImGui::Text("Username:");
        ImGui::SameLine(200);
        if (ImGui::InputText("##username", username, sizeof(username), ImGuiInputTextFlags_EnterReturnsTrue))
            connect = true;
        ImGui::Text("Server IP:");
        ImGui::SameLine(200);
        if (ImGui::InputText("##server", serverIP, sizeof(serverIP), ImGuiInputTextFlags_EnterReturnsTrue))
            connect = true;

        ImGui::Text("Port:");
        ImGui::SameLine(200);
        if (ImGui::InputText("##port", port, sizeof(port), ImGuiInputTextFlags_EnterReturnsTrue))
            connect = true;

        ImGui::Spacing();
        ImGui::Separator();
        ImGui::Spacing();

        if (ImGui::Button("Connect", ImVec2(120, 30)) || connect) {
            int portNum = atoi(port);
            if (strlen(username) > 0 && portNum > 0 && serverIP != "") {
                if (connect_server(serverIP, portNum, username)) {
                }
                else {
                    public_message.push_back(ChatMessage("System", "Connection failed"));
                }   } }

        ImGui::SameLine();
        ImGui::Spacing();
        ImGui::TextColored(ImVec4(1, 0, 1, 1), "Tips:");
        ImGui::Text("1. Run the server on port 65432");
        ImGui::Text("2. Use different names for multiple clients");
        ImGui::Text("3. Click on user names to start private chat");
    }
    ImGui::End();
}
```

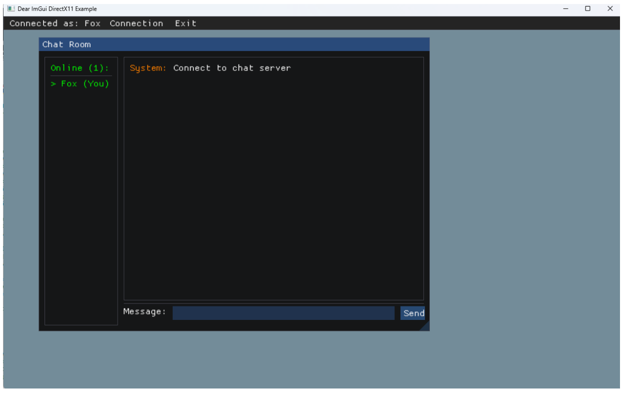
The user window displays a list of all online users. It iterates through the users online vector using a for loop. If it's the user's own name, it will be highlighted in green and appended with "(YOU)". Other users' text is selectable. Clicking a text window will check if the user's private chat window exists in the private chat map. If not, a new one will be created, and an asterisk (*) will be added after the user's name.

```
void ChatWindow::user_win() {
    ImGui::BeginChild("Users", ImVec2(150, 0), true);

    ImGui::TextColored(ImVec4(0, 1, 0, 1), "Online (%d):", (int)users_online.size());
    ImGui::Separator();
    for (const auto& user : users_online) {
        if (user == username) {
            ImGui::TextColored(ImVec4(0, 1, 0, 1), "> %s (You)", user.c_str());
        }
        else {
            bool hasPrivateChat = (private_chat.find(user) != private_chat.end());

            if (ImGui::Selectable(user.c_str())) {
                // open chat
                if (private_chat.find(user) == private_chat.end()) {
                    private_chat[user] = std::vector<ChatMessage>();
                    private_input[user].fill('\0');
                } }

            if (hasPrivateChat) {
                ImGui::SameLine();
                ImGui::TextColored(ImVec4(1, 0.5f, 1, 1), " *");
            }   } }

    ImGui::EndChild();
}
```

The chat window will display all public messages, then differentiate them by the sender's username using different colors. A for loop will render all messages at once. A public input is bound to the input field as a temporary buffer. Pressing Enter or clicking the send button sends the content from the input buffer to the server as a public message. This message is then pushed to the local public message vector for display, and finally the input buffer is cleared. Figure 21, Figure 23, Figure 22

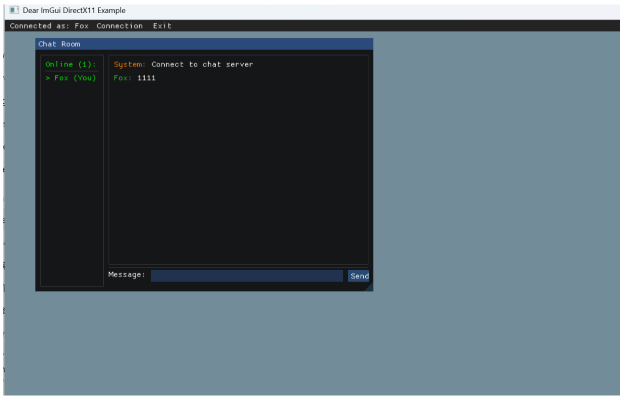
```
void ChatWindow::chat_win() {
    ImGui::BeginChild("Chat messages", ImVec2(0, -ImGui::GetFrameHeightWithSpacing() * 1.5f), true);

    for (const auto& p_message : public_message) {
        // set user color
        if (p_message.sender == username) {
            ImGui::PushStyleColor(ImGuiCol_Text, ImVec4(0.2f, 0.8f, 0.2f, 1.0f));}
        else if (p_message.sender == "System") {
            ImGui::PushStyleColor(ImGuiCol_Text, ImVec4(1.0f, 0.5f, 0.0f, 1.0f)); }
        else {
            ImGui::PushStyleColor(ImGuiCol_Text, ImVec4(0.8f, 0.8f, 0.2f, 1.0f)); }

        ImGui::Text("%s:", p_message.sender.c_str());
        ImGui::PopStyleColor();

        ImGui::SameLine();
        ImGui::TextWrapped("%s", p_message.text.c_str());

        ImGui::Spacing(); }

    ImGui::EndChild();

    // input
    ImGui::Separator();

    ImGui::Text("Message:");
    ImGui::SameLine();

    ImGui::PushItemWidth(-60);
    bool sendPublic = false;
    // press endter
    if (ImGui::InputText("##PublicInput", public_input, sizeof(public_input), ImGuiInputTextFlags_EnterReturnsTrue))  {
        sendPublic = true;}
    ImGui::PopItemWidth();

    ImGui::SameLine();
    if (ImGui::Button("Send", ImVec2(50, 0)) || sendPublic) {
        if (strlen(public_input) > 0) {
            // send
            PublicMessage message(username, public_input);

            if (send_message_toserver(MessageType::PUBLIC_MESSAGE, &message, sizeof(message))) {
                // show at local
                ChatMessage mess(username, public_input);
                public_message.push_back(mess);
            }
            // clear input
            public_input[0] = ’\0’; }}
```

The private chat window function iterates through the entire private chat map. For each private chat, it uses the username as the window's title key. This allows ImGUI to remember the window's information and inherit the size and position information of the window with the same name during the next 'ImGUI::Begin' rendering loop. Then, it renders all messages for that private chat, distinguishing them by usernames in dif-



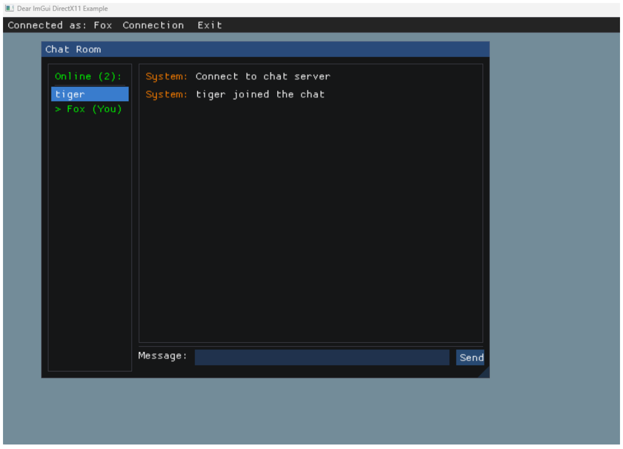
System: Connect to chat server

Figure 21: *The public chat room.*



System: Connect to chat server
Fox: 1111

Figure 22: *After send a public message.The different color of different user.*

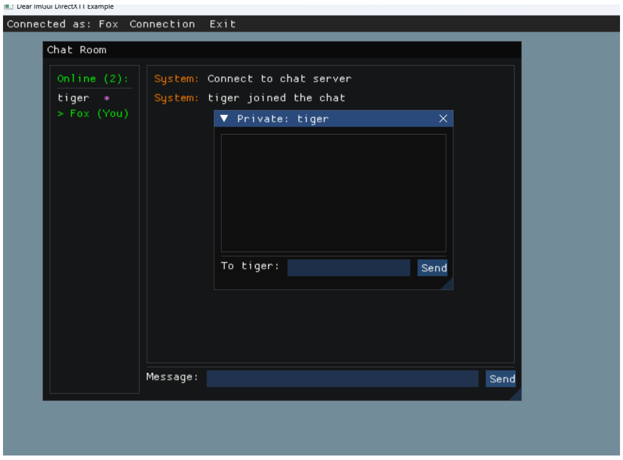


System: Connect to chat server
System: tiger joined the chat

Figure 23: *New client join in.The user list changed*

ferent colors. Sending messages uses the same logic as public messages: pushing the content of the input into the user's message vector and clearing the input buffer. If the user clicks the close window button, then clear the user's record from the private chat map, and the content in their input buffer map is also deleted. Figure 25.

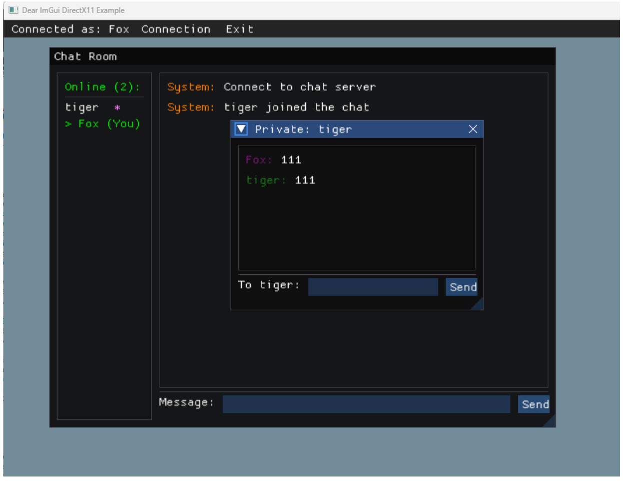
```
void ChatWindow::private_win() {
    for (auto it = private_chat.begin(); it != private_chat.end();) {
        const std::string targetUser = it->first;
        std::vector<ChatMessage>& messages = it->second;

        std::string title = "Private: " + targetUser;
        bool isopen = true;

        ImGui::SetNextWindowSize(ImVec2(400, 300), ImGuiCond_FirstUseEver);
        // imgui can renmber last window's detail by title
        if (ImGui::Begin(title.c_str(), &isopen)) {

            ImGui::BeginChild("PrivateMessages", ImVec2(0, -ImGui::GetFrameHeightWithSpacing() * 1.5f), true);

            for (const auto& mes : messages) {
                // color
                if (mes.sender == username) {
                    ImGui::PushStyleColor(ImGuiCol_Text, ImVec4(0.5f, 0.1f, 0.5f, 1.0f));
                }
                else {
                    ImGui::PushStyleColor(ImGuiCol_Text, ImVec4(0.1f, 0.5f, 0.1f, 1.0f));
                }
                ImGui::Text("%s:", mes.sender.c_str());
                ImGui::PopStyleColor();

                ImGui::SameLine();
                ImGui::TextWrapped("%s", mes.text.c_str());

                ImGui::Spacing();
            }

            ImGui::EndChild();

            // input
            ImGui::Separator();
            ImGui::Text("To %s:", targetUser.c_str());
            ImGui::SameLine();

            auto& input_buff = private_input[targetUser];

            ImGui::PushItemWidth(-60);
            bool send = false;
            // press endter
            if (ImGui::InputText("##PrivateInput", input_buff.data(), input_buff.size(), ImGuiInputTextFlags_EnterReturnsTrue)) {
                send = true;
            }
            ImGui::PopItemWidth();

            ImGui::SameLine();
            if (ImGui::Button("Send", ImVec2(50, 0)) || send) {
                if (strlen(input_buff.data()) > 0) {
                    PrivateMessage message(username, targetUser, input_buff.data());
                    if (send_message_toserver(MessageType::PRIVATE_MESSAGE, &message, sizeof(message))) {
                        std::string name = (username == username) ? targetUser : username;

                        // check private chat map
                        if (private_chat.find(name) == private_chat.end()) {
                            private_chat[name] = std::vector<ChatMessage>();
                        }
                        // add chat message
                        ChatMessage mess(username, input_buff.data(), true, targetUser);
                        private_chat[name].push_back(mess);

                        if (private_input.find(name) == private_input.end()) {
                            private_input[name].fill('\0');
                        }
                    }
                    input_buff.fill('\0');     }
        }  }
        ImGui::End();
        // if close the chat, cler the chat
        if (!isopen) {
            it = private_chat.erase(it);
            private_input.erase(targetUser);
        }
        else {
            ++it;      } } }
```



System: Connect to chat server

System: tiger joined the chat

Figure 24: *New client join in.The user list changed*



```
System: Connect to chat server
System: tiger joined the chat
Fox: 111
tiger: 111
```

Figure 25: *Shows the private Chat.*

2.4 Conclusion

This project implements a server program for receiving connections and processing messages, and a small chat room program for sending public and private messages. In terms of program design, since it doesn't consider scenarios with a large number of client connections, a task scheduling system for managing a large number of client connections is not implemented. If time permits, I would consider storing the thread for each client and implementing elegant life management. I could also consider using a thread pool to handle messages from multiple clients. In the client-side, I could consider adding an automatic reconnection function or using multithreading with timestamps to handle a large number of incoming server messages.

3 Links

Rasterizer Link: `https://github.com/Tabris-XiangyiChen/GE_Rasterizer_Optimisation`
Chat Room Link: `https://github.com/Tabris-XiangyiChen/GE_Chat_Room`