

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.

```
1 void scene3(bool if_trans_first = false, bool
2     if_AVX = false){
3     // ...
4     // 1. Create high-poly spheres
5     int LAT = 40;
6     int LON = 80;
7     for (int i = 0; i < 6; ++i){
8         Mesh* s = new Mesh();
9         *s = Mesh::makeSphere(2.5f, LAT, LON);
10
11         float x = -6.0f + (i % 3) * 6.0f;
12         float y = (i < 3) ? 3.5f : -3.5f;
13
14         s->world = matrix::makeTranslation(x, y,
15             -10.f);
16         scene.push_back(s);
17     }
18     // ...
19     while (running)
20     {
21         // ...
22         // Animate only top row
23         for (int i = 0; i < 3; ++i)
24         {
25             scene[i]->world =
26                 scene[i]->world *
27                     matrix::makeRotateXYZ(0.01f,
28                         0.02f, 0.015f);
29         }
30     }
31 }
```

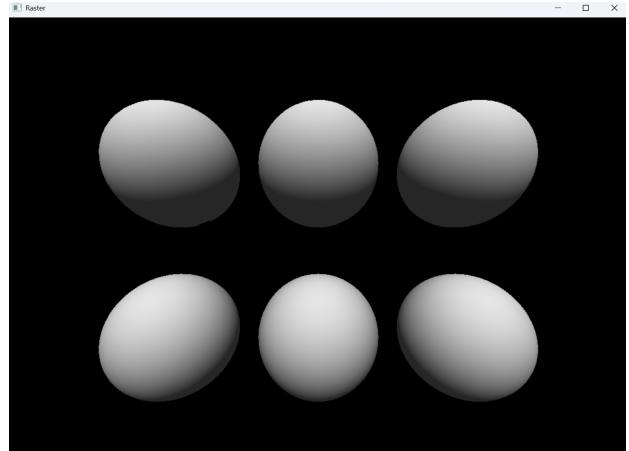


Figure 1: Scene3

```
25     matrix::makeRotateXYZ(0.01f,
26         0.02f, 0.015f);
27     }
28     //...
29 }}
```

1.3 Single Thread Optimization

1.3.1 Basic Calculations 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.

```
1 void normalise() {
2     float length = std::sqrt(x * x + y * y + z * z);
3     float invlength = 1 / length;
4     x *= invlength;
5     y *= invlength;
6     z *= invlength;
7 }
```

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.

```

1 // SSE matrix multiply: out = A * B
2 static inline void mul(MatData& out, const
3     MatData& A, const MatData& B) {
4     __m128 b0 = _mm_load_ps(&B.a[0]);
5     __m128 b1 = _mm_load_ps(&B.a[4]);
6     __m128 b2 = _mm_load_ps(&B.a[8]);
7     __m128 b3 = _mm_load_ps(&B.a[12]);
8
9     for (int r = 0; r < 4; ++r) {
10         __m128 a0 = _mm_set1_ps(A.a[r * 4 + 0]);
11         __m128 a1 = _mm_set1_ps(A.a[r * 4 + 1]);
12         __m128 a2 = _mm_set1_ps(A.a[r * 4 + 2]);
13         __m128 a3 = _mm_set1_ps(A.a[r * 4 + 3]);
14
15         __m128 res0 = _mm_add_ps(_mm_add_ps(
16             _mm_mul_ps(a0, b0),
17             _mm_mul_ps(a1, b1)),
18             _mm_add_ps(_mm_mul_ps(a2, b2),
19             _mm_mul_ps(a3, b3)));
20
21         _mm_store_ps(&out.a[r * 4], res0);
22     }
23
24 // SSE matrix * vector using _mm_dp_ps
25 static inline void mul(vec4& out, const MatData&
26     A, const vec4& v) {
27     __m128 vec = _mm_load_ps(&v[0]);
28
29     for (int row = 0; row < 4; ++row) {
30
31         __m128 res0 = _mm_dp_ps(vec, A.a[row * 4]);
32
33         __m128 res1 = _mm_dp_ps(vec, A.a[row * 4 + 1]);
34
35         __m128 res2 = _mm_dp_ps(vec, A.a[row * 4 + 2]);
36
37         __m128 res3 = _mm_dp_ps(vec, A.a[row * 4 + 3]);
38
39         __m128 res4 = _mm_dp_ps(res0, res1);
40
41         __m128 res5 = _mm_dp_ps(res4, res2);
42
43         __m128 res6 = _mm_dp_ps(res5, res3);
44
45         __m128 res7 = _mm_dp_ps(res6, res0);
46
47         __m128 res8 = _mm_dp_ps(res7, res1);
48
49         __m128 res9 = _mm_dp_ps(res8, res2);
50
51         __m128 res10 = _mm_dp_ps(res9, res3);
52
53         __m128 res11 = _mm_dp_ps(res10, res0);
54
55         __m128 res12 = _mm_dp_ps(res11, res1);
56
57         __m128 res13 = _mm_dp_ps(res12, res2);
58
59         __m128 res14 = _mm_dp_ps(res13, res3);
60
61         __m128 res15 = _mm_dp_ps(res14, res0);
62
63         __m128 res16 = _mm_dp_ps(res15, res1);
64
65         __m128 res17 = _mm_dp_ps(res16, res2);
66
67         __m128 res18 = _mm_dp_ps(res17, res3);
68
69         __m128 res19 = _mm_dp_ps(res18, res0);
70
71         __m128 res20 = _mm_dp_ps(res19, res1);
72
73         __m128 res21 = _mm_dp_ps(res20, res2);
74
75         __m128 res22 = _mm_dp_ps(res21, res3);
76
77         __m128 res23 = _mm_dp_ps(res22, res0);
78
79         __m128 res24 = _mm_dp_ps(res23, res1);
80
81         __m128 res25 = _mm_dp_ps(res24, res2);
82
83         __m128 res26 = _mm_dp_ps(res25, res3);
84
85         __m128 res27 = _mm_dp_ps(res26, res0);
86
87         __m128 res28 = _mm_dp_ps(res27, res1);
88
89         __m128 res29 = _mm_dp_ps(res28, res2);
90
91         __m128 res30 = _mm_dp_ps(res29, res3);
92
93         __m128 res31 = _mm_dp_ps(res30, res0);
94
95         __m128 res32 = _mm_dp_ps(res31, res1);
96
97         __m128 res33 = _mm_dp_ps(res32, res2);
98
99         __m128 res34 = _mm_dp_ps(res33, res3);
100
101         __m128 res35 = _mm_dp_ps(res34, res0);
102
103         __m128 res36 = _mm_dp_ps(res35, res1);
104
105         __m128 res37 = _mm_dp_ps(res36, res2);
106
107         __m128 res38 = _mm_dp_ps(res37, res3);
108
109         __m128 res39 = _mm_dp_ps(res38, res0);
110
111         __m128 res40 = _mm_dp_ps(res39, res1);
112
113         __m128 res41 = _mm_dp_ps(res40, res2);
114
115         __m128 res42 = _mm_dp_ps(res41, res3);
116
117         __m128 res43 = _mm_dp_ps(res42, res0);
118
119         __m128 res44 = _mm_dp_ps(res43, res1);
120
121         __m128 res45 = _mm_dp_ps(res44, res2);
122
123         __m128 res46 = _mm_dp_ps(res45, res3);
124
125         __m128 res47 = _mm_dp_ps(res46, res0);
126
127         __m128 res48 = _mm_dp_ps(res47, res1);
128
129         __m128 res49 = _mm_dp_ps(res48, res2);
130
131         __m128 res50 = _mm_dp_ps(res49, res3);
132
133         __m128 res51 = _mm_dp_ps(res50, res0);
134
135         __m128 res52 = _mm_dp_ps(res51, res1);
136
137         __m128 res53 = _mm_dp_ps(res52, res2);
138
139         __m128 res54 = _mm_dp_ps(res53, res3);
140
141         __m128 res55 = _mm_dp_ps(res54, res0);
142
143         __m128 res56 = _mm_dp_ps(res55, res1);
144
145         __m128 res57 = _mm_dp_ps(res56, res2);
146
147         __m128 res58 = _mm_dp_ps(res57, res3);
148
149         __m128 res59 = _mm_dp_ps(res58, res0);
150
151         __m128 res60 = _mm_dp_ps(res59, res1);
152
153         __m128 res61 = _mm_dp_ps(res60, res2);
154
155         __m128 res62 = _mm_dp_ps(res61, res3);
156
157         __m128 res63 = _mm_dp_ps(res62, res0);
158
159         __m128 res64 = _mm_dp_ps(res63, res1);
160
161         __m128 res65 = _mm_dp_ps(res64, res2);
162
163         __m128 res66 = _mm_dp_ps(res65, res3);
164
165         __m128 res67 = _mm_dp_ps(res66, res0);
166
167         __m128 res68 = _mm_dp_ps(res67, res1);
168
169         __m128 res69 = _mm_dp_ps(res68, res2);
170
171         __m128 res70 = _mm_dp_ps(res69, res3);
172
173         __m128 res71 = _mm_dp_ps(res70, res0);
174
175         __m128 res72 = _mm_dp_ps(res71, res1);
176
177         __m128 res73 = _mm_dp_ps(res72, res2);
178
179         __m128 res74 = _mm_dp_ps(res73, res3);
180
181         __m128 res75 = _mm_dp_ps(res74, res0);
182
183         __m128 res76 = _mm_dp_ps(res75, res1);
184
185         __m128 res77 = _mm_dp_ps(res76, res2);
186
187         __m128 res78 = _mm_dp_ps(res77, res3);
188
189         __m128 res79 = _mm_dp_ps(res78, res0);
190
191         __m128 res80 = _mm_dp_ps(res79, res1);
192
193         __m128 res81 = _mm_dp_ps(res80, res2);
194
195         __m128 res82 = _mm_dp_ps(res81, res3);
196
197         __m128 res83 = _mm_dp_ps(res82, res0);
198
199         __m128 res84 = _mm_dp_ps(res83, res1);
200
201         __m128 res85 = _mm_dp_ps(res84, res2);
202
203         __m128 res86 = _mm_dp_ps(res85, res3);
204
205         __m128 res87 = _mm_dp_ps(res86, res0);
206
207         __m128 res88 = _mm_dp_ps(res87, res1);
208
209         __m128 res89 = _mm_dp_ps(res88, res2);
210
211         __m128 res90 = _mm_dp_ps(res89, res3);
212
213         __m128 res91 = _mm_dp_ps(res90, res0);
214
215         __m128 res92 = _mm_dp_ps(res91, res1);
216
217         __m128 res93 = _mm_dp_ps(res92, res2);
218
219         __m128 res94 = _mm_dp_ps(res93, res3);
220
221         __m128 res95 = _mm_dp_ps(res94, res0);
222
223         __m128 res96 = _mm_dp_ps(res95, res1);
224
225         __m128 res97 = _mm_dp_ps(res96, res2);
226
227         __m128 res98 = _mm_dp_ps(res97, res3);
228
229         __m128 res99 = _mm_dp_ps(res98, res0);
230
231         __m128 res100 = _mm_dp_ps(res99, res1);
232
233         __m128 res101 = _mm_dp_ps(res100, res2);
234
235         __m128 res102 = _mm_dp_ps(res101, res3);
236
237         __m128 res103 = _mm_dp_ps(res102, res0);
238
239         __m128 res104 = _mm_dp_ps(res103, res1);
240
241         __m128 res105 = _mm_dp_ps(res104, res2);
242
243         __m128 res106 = _mm_dp_ps(res105, res3);
244
245         __m128 res107 = _mm_dp_ps(res106, res0);
246
247         __m128 res108 = _mm_dp_ps(res107, res1);
248
249         __m128 res109 = _mm_dp_ps(res108, res2);
250
251         __m128 res110 = _mm_dp_ps(res109, res3);
252
253         __m128 res111 = _mm_dp_ps(res110, res0);
254
255         __m128 res112 = _mm_dp_ps(res111, res1);
256
257         __m128 res113 = _mm_dp_ps(res112, res2);
258
259         __m128 res114 = _mm_dp_ps(res113, res3);
260
261         __m128 res115 = _mm_dp_ps(res114, res0);
262
263         __m128 res116 = _mm_dp_ps(res115, res1);
264
265         __m128 res117 = _mm_dp_ps(res116, res2);
266
267         __m128 res118 = _mm_dp_ps(res117, res3);
268
269         __m128 res119 = _mm_dp_ps(res118, res0);
270
271         __m128 res120 = _mm_dp_ps(res119, res1);
272
273         __m128 res121 = _mm_dp_ps(res120, res2);
274
275         __m128 res122 = _mm_dp_ps(res121, res3);
276
277         __m128 res123 = _mm_dp_ps(res122, res0);
278
279         __m128 res124 = _mm_dp_ps(res123, res1);
280
281         __m128 res125 = _mm_dp_ps(res124, res2);
282
283         __m128 res126 = _mm_dp_ps(res125, res3);
284
285         __m128 res127 = _mm_dp_ps(res126, res0);
286
287         __m128 res128 = _mm_dp_ps(res127, res1);
288
289         __m128 res129 = _mm_dp_ps(res128, res2);
290
291         __m128 res130 = _mm_dp_ps(res129, res3);
292
293         __m128 res131 = _mm_dp_ps(res130, res0);
294
295         __m128 res132 = _mm_dp_ps(res131, res1);
296
297         __m128 res133 = _mm_dp_ps(res132, res2);
298
299         __m128 res134 = _mm_dp_ps(res133, res3);
300
301         __m128 res135 = _mm_dp_ps(res134, res0);
302
303         __m128 res136 = _mm_dp_ps(res135, res1);
304
305         __m128 res137 = _mm_dp_ps(res136, res2);
306
307         __m128 res138 = _mm_dp_ps(res137, res3);
308
309         __m128 res139 = _mm_dp_ps(res138, res0);
310
311         __m128 res140 = _mm_dp_ps(res139, res1);
312
313         __m128 res141 = _mm_dp_ps(res140, res2);
314
315         __m128 res142 = _mm_dp_ps(res141, res3);
316
317         __m128 res143 = _mm_dp_ps(res142, res0);
318
319         __m128 res144 = _mm_dp_ps(res143, res1);
320
321         __m128 res145 = _mm_dp_ps(res144, res2);
322
323         __m128 res146 = _mm_dp_ps(res145, res3);
324
325         __m128 res147 = _mm_dp_ps(res146, res0);
326
327         __m128 res148 = _mm_dp_ps(res147, res1);
328
329         __m128 res149 = _mm_dp_ps(res148, res2);
330
331         __m128 res150 = _mm_dp_ps(res149, res3);
332
333         __m128 res151 = _mm_dp_ps(res150, res0);
334
335         __m128 res152 = _mm_dp_ps(res151, res1);
336
337         __m128 res153 = _mm_dp_ps(res152, res2);
338
339         __m128 res154 = _mm_dp_ps(res153, res3);
340
341         __m128 res155 = _mm_dp_ps(res154, res0);
342
343         __m128 res156 = _mm_dp_ps(res155, res1);
344
345         __m128 res157 = _mm_dp_ps(res156, res2);
346
347         __m128 res158 = _mm_dp_ps(res157, res3);
348
349         __m128 res159 = _mm_dp_ps(res158, res0);
350
351         __m128 res160 = _mm_dp_ps(res159, res1);
352
353         __m128 res161 = _mm_dp_ps(res160, res2);
354
355         __m128 res162 = _mm_dp_ps(res161, res3);
356
357         __m128 res163 = _mm_dp_ps(res162, res0);
358
359         __m128 res164 = _mm_dp_ps(res163, res1);
360
361         __m128 res165 = _mm_dp_ps(res164, res2);
362
363         __m128 res166 = _mm_dp_ps(res165, res3);
364
365         __m128 res167 = _mm_dp_ps(res166, res0);
366
367         __m128 res168 = _mm_dp_ps(res167, res1);
368
369         __m128 res169 = _mm_dp_ps(res168, res2);
370
371         __m128 res170 = _mm_dp_ps(res169, res3);
372
373         __m128 res171 = _mm_dp_ps(res170, res0);
374
375         __m128 res172 = _mm_dp_ps(res171, res1);
376
377         __m128 res173 = _mm_dp_ps(res172, res2);
378
379         __m128 res174 = _mm_dp_ps(res173, res3);
380
381         __m128 res175 = _mm_dp_ps(res174, res0);
382
383         __m128 res176 = _mm_dp_ps(res175, res1);
384
385         __m128 res177 = _mm_dp_ps(res176, res2);
386
387         __m128 res178 = _mm_dp_ps(res177, res3);
388
389         __m128 res179 = _mm_dp_ps(res178, res0);
390
391         __m128 res180 = _mm_dp_ps(res179, res1);
392
393         __m128 res181 = _mm_dp_ps(res180, res2);
394
395         __m128 res182 = _mm_dp_ps(res181, res3);
396
397         __m128 res183 = _mm_dp_ps(res182, res0);
398
399         __m128 res184 = _mm_dp_ps(res183, res1);
400
401         __m128 res185 = _mm_dp_ps(res184, res2);
402
403         __m128 res186 = _mm_dp_ps(res185, res3);
404
405         __m128 res187 = _mm_dp_ps(res186, res0);
406
407         __m128 res188 = _mm_dp_ps(res187, res1);
408
409         __m128 res189 = _mm_dp_ps(res188, res2);
410
411         __m128 res190 = _mm_dp_ps(res189, res3);
412
413         __m128 res191 = _mm_dp_ps(res190, res0);
414
415         __m128 res192 = _mm_dp_ps(res191, res1);
416
417         __m128 res193 = _mm_dp_ps(res192, res2);
418
419         __m128 res194 = _mm_dp_ps(res193, res3);
420
421         __m128 res195 = _mm_dp_ps(res194, res0);
422
423         __m128 res196 = _mm_dp_ps(res195, res1);
424
425         __m128 res197 = _mm_dp_ps(res196, res2);
426
427         __m128 res198 = _mm_dp_ps(res197, res3);
428
429         __m128 res199 = _mm_dp_ps(res198, res0);
430
431         __m128 res200 = _mm_dp_ps(res199, res1);
432
433         __m128 res201 = _mm_dp_ps(res200, res2);
434
435         __m128 res202 = _mm_dp_ps(res201, res3);
436
437         __m128 res203 = _mm_dp_ps(res202, res0);
438
439         __m128 res204 = _mm_dp_ps(res203, res1);
440
441         __m128 res205 = _mm_dp_ps(res204, res2);
442
443         __m128 res206 = _mm_dp_ps(res205, res3);
444
445         __m128 res207 = _mm_dp_ps(res206, res0);
446
447         __m128 res208 = _mm_dp_ps(res207, res1);
448
449         __m128 res209 = _mm_dp_ps(res208, res2);
450
451         __m128 res210 = _mm_dp_ps(res209, res3);
452
453         __m128 res211 = _mm_dp_ps(res210, res0);
454
455         __m128 res212 = _mm_dp_ps(res211, res1);
456
457         __m128 res213 = _mm_dp_ps(res212, res2);
458
459         __m128 res214 = _mm_dp_ps(res213, res3);
460
461         __m128 res215 = _mm_dp_ps(res214, res0);
462
463         __m128 res216 = _mm_dp_ps(res215, res1);
464
465         __m128 res217 = _mm_dp_ps(res216, res2);
466
467         __m128 res218 = _mm_dp_ps(res217, res3);
468
469         __m128 res219 = _mm_dp_ps(res218, res0);
470
471         __m128 res220 = _mm_dp_ps(res219, res1);
472
473         __m128 res221 = _mm_dp_ps(res220, res2);
474
475         __m128 res222 = _mm_dp_ps(res221, res3);
476
477         __m128 res223 = _mm_dp_ps(res222, res0);
478
479         __m128 res224 = _mm_dp_ps(res223, res1);
480
481         __m128 res225 = _mm_dp_ps(res224, res2);
482
483         __m128 res226 = _mm_dp_ps(res225, res3);
484
485         __m128 res227 = _mm_dp_ps(res226, res0);
486
487         __m128 res228 = _mm_dp_ps(res227, res1);
488
489         __m128 res229 = _mm_dp_ps(res228, res2);
490
491         __m128 res230 = _mm_dp_ps(res229, res3);
492
493         __m128 res231 = _mm_dp_ps(res230, res0);
494
495         __m128 res232 = _mm_dp_ps(res231, res1);
496
497         __m128 res233 = _mm_dp_ps(res232, res2);
498
499         __m128 res234 = _mm_dp_ps(res233, res3);
500
501         __m128 res235 = _mm_dp_ps(res234, res0);
502
503         __m128 res236 = _mm_dp_ps(res235, res1);
504
505         __m128 res237 = _mm_dp_ps(res236, res2);
506
507         __m128 res238 = _mm_dp_ps(res237, res3);
508
509         __m128 res239 = _mm_dp_ps(res238, res0);
510
511         __m128 res240 = _mm_dp_ps(res239, res1);
512
513         __m128 res241 = _mm_dp_ps(res240, res2);
514
515         __m128 res242 = _mm_dp_ps(res241, res3);
516
517         __m128 res243 = _mm_dp_ps(res242, res0);
518
519         __m128 res244 = _mm_dp_ps(res243, res1);
520
521         __m128 res245 = _mm_dp_ps(res244, res2);
522
523         __m128 res246 = _mm_dp_ps(res245, res3);
524
525         __m128 res247 = _mm_dp_ps(res246, res0);
526
527         __m128 res248 = _mm_dp_ps(res247, res1);
528
529         __m128 res249 = _mm_dp_ps(res248, res2);
530
531         __m128 res250 = _mm_dp_ps(res249, res3);
532
533         __m128 res251 = _mm_dp_ps(res250, res0);
534
535         __m128 res252 = _mm_dp_ps(res251, res1);
536
537         __m128 res253 = _mm_dp_ps(res252, res2);
538
539         __m128 res254 = _mm_dp_ps(res253, res3);
540
541         __m128 res255 = _mm_dp_ps(res254, res0);
542
543         __m128 res256 = _mm_dp_ps(res255, res1);
544
545         __m128 res257 = _mm_dp_ps(res256, res2);
546
547         __m128 res258 = _mm_dp_ps(res257, res3);
548
549         __m128 res259 = _mm_dp_ps(res258, res0);
550
551         __m128 res260 = _mm_dp_ps(res259, res1);
552
553         __m128 res261 = _mm_dp_ps(res260, res2);
554
555         __m128 res262 = _mm_dp_ps(res261, res3);
556
557         __m128 res263 = _mm_dp_ps(res262, res0);
558
559         __m128 res264 = _mm_dp_ps(res263, res1);
560
561         __m128 res265 = _mm_dp_ps(res264, res2);
562
563         __m128 res266 = _mm_dp_ps(res265, res3);
564
565         __m128 res267 = _mm_dp_ps(res266, res0);
566
567         __m128 res268 = _mm_dp_ps(res267, res1);
568
569         __m128 res269 = _mm_dp_ps(res268, res2);
570
571         __m128 res270 = _mm_dp_ps(res269, res3);
572
573         __m128 res271 = _mm_dp_ps(res270, res0);
574
575         __m128 res272 = _mm_dp_ps(res271, res1);
576
577         __m128 res273 = _mm_dp_ps(res272, res2);
578
579         __m128 res274 = _mm_dp_ps(res273, res3);
580
581         __m128 res275 = _mm_dp_ps(res274, res0);
582
583         __m128 res276 = _mm_dp_ps(res275, res1);
584
585         __m128 res277 = _mm_dp_ps(res276, res2);
586
587         __m128 res278 = _mm_dp_ps(res277, res3);
588
589         __m128 res279 = _mm_dp_ps(res278, res0);
590
591         __m128 res280 = _mm_dp_ps(res279, res1);
592
593         __m128 res281 = _mm_dp_ps(res280, res2);
594
595         __m128 res282 = _mm_dp_ps(res281, res3);
596
597         __m128 res283 = _mm_dp_ps(res282, res0);
598
599         __m128 res284 = _mm_dp_ps(res283, res1);
600
601         __m128 res285 = _mm_dp_ps(res284, res2);
602
603         __m128 res286 = _mm_dp_ps(res285, res3);
604
605         __m128 res287 = _mm_dp_ps(res286, res0);
606
607         __m128 res288 = _mm_dp_ps(res287, res1);
608
609         __m128 res289 = _mm_dp_ps(res288, res2);
610
611         __m128 res290 = _mm_dp_ps(res289, res3);
612
613         __m128 res291 = _mm_dp_ps(res290, res0);
614
615         __m128 res292 = _mm_dp_ps(res291, res1);
616
617         __m128 res293 = _mm_dp_ps(res292, res2);
618
619         __m128 res294 = _mm_dp_ps(res293, res3);
620
621         __m128 res295 = _mm_dp_ps(res294, res0);
622
623         __m128 res296 = _mm_dp_ps(res295, res1);
624
625         __m128 res297 = _mm_dp_ps(res296, res2);
626
627         __m128 res298 = _mm_dp_ps(res297, res3);
628
629         __m128 res299 = _mm_dp_ps(res298, res0);
630
631         __m128 res300 = _mm_dp_ps(res299, res1);
632
633         __m128 res301 = _mm_dp_ps(res300, res2);
634
635         __m128 res302 = _mm_dp_ps(res301, res3);
636
637         __m128 res303 = _mm_dp_ps(res302, res0);
638
639         __m128 res304 = _mm_dp_ps(res303, res1);
640
641         __m128 res305 = _mm_dp_ps(res304, res2);
642
643         __m128 res306 = _mm_dp_ps(res305, res3);
644
645         __m128 res307 = _mm_dp_ps(res306, res0);
646
647         __m128 res308 = _mm_dp_ps(res307, res1);
648
649         __m128 res309 = _mm_dp_ps(res308, res2);
650
651         __m128 res310 = _mm_dp_ps(res309, res3);
652
653         __m128 res311 = _mm_dp_ps(res310, res0);
654
655         __m128 res312 = _mm_dp_ps(res311, res1);
656
657         __m128 res313 = _mm_dp_ps(res312, res2);
658
659         __m128 res314 = _mm_dp_ps(res313, res3);
660
661         __m128 res315 = _mm_dp_ps(res314, res0);
662
663         __m128 res316 = _mm_dp_ps(res315, res1);
664
665         __m128 res317 = _mm_dp_ps(res316, res2);
666
667         __m128 res318 = _mm_dp_ps(res317, res3);
668
669         __m128 res319 = _mm_dp_ps(res318, res0);
670
671         __m128 res320 = _mm_dp_ps(res319, res1);
672
673         __m128 res321 = _mm_dp_ps(res320, res2);
674
675         __m128 res322 = _mm_dp_ps(res321, res3);
676
677         __m128 res323 = _mm_dp_ps(res322, res0);
678
679         __m128 res324 = _mm_dp_ps(res323, res1);
680
681         __m128 res325 = _mm_dp_ps(res324, res2);
682
683         __m128 res326 = _mm_dp_ps(res325, res3);
684
685         __m128 res327 = _mm_dp_ps(res326, res0);
686
687         __m128 res328 = _mm_dp_ps(res327, res1);
688
689         __m128 res329 = _mm_dp_ps(res328, res2);
690
691         __m128 res330 = _mm_dp_ps(res329, res3);
692
693         __m128 res331 = _mm_dp_ps(res330, res0);
694
695         __m128 res332 = _mm_dp_ps(res331, res1);
696
697         __m128 res333 = _mm_dp_ps(res332, res2);
698
699         __m128 res334 = _mm_dp_ps(res333, res3);
700
701         __m128 res335 = _mm_dp_ps(res334, res0);
702
703         __m128 res336 = _mm_dp_ps(res335, res1);
704
705         __m128 res337 = _mm_dp_ps(res336, res2);
706
707         __m128 res338 = _mm_dp_ps(res337, res3);
708
709         __m128 res339 = _mm_dp_ps(res338, res0);
710
711         __m128 res340 = _mm_dp_ps(res339, res1);
712
713         __m128 res341 = _mm_dp_ps(res340, res2);
714
715         __m128 res342 = _mm_dp_ps(res341, res3);
716
717         __m128 res343 = _mm_dp_ps(res342, res0);
718
719         __m128 res344 = _mm_dp_ps(res343, res1);
720
721         __m128 res345 = _mm_dp_ps(res344, res2);
722
723         __m128 res346 = _mm_dp_ps(res345, res3);
724
725         __m128 res347 = _mm_dp_ps(res346, res0);
726
727         __m128 res348 = _mm_dp_ps(res347, res1);
728
729         __m128 res349 = _mm_dp_ps(res348, res2);
730
731         __m128 res350 = _mm_dp_ps(res349, res3);
732
733         __m128 res351 = _mm_dp_ps(res350, res0);
734
735         __m128 res352 = _mm_dp_ps(res351, res1);
736
737         __m128 res353 = _mm_dp_ps(res352, res2);
738
739         __m128 res354 = _mm_dp_ps(res353, res3);
740
741         __m128 res355 = _mm_dp_ps(res354, res0);
742
743         __m128 res356 = _mm_dp_ps(res355, res1);
744
745         __m128 res357 = _mm_dp_ps(res356, res2);
746
747         __m128 res358 = _mm_dp_ps(res357, res3);
748
749         __m128 res359 = _mm_dp_ps(res358, res0);
750
751         __m128 res360 = _mm_dp_ps(res359, res1);
752
753         __m128 res361 = _mm_dp_ps(res360, res2);
754
755         __m128 res362 = _mm_dp_ps(res361, res3);
756
757         __m128 res363 = _mm_dp_ps(res362, res0);
758
759         __m128 res364 = _mm_dp_ps(res363, res1);
760
761         __m128 res365 = _mm_dp_ps(res364, res2);
762
763         __m128 res366 = _mm_dp_ps(res365, res3);
764
765         __m128 res367 = _mm_dp_ps(res366, res0);
766
767         __m128 res368 = _mm_dp_ps(res367, res1);
768
769         __m128 res369 = _mm_dp_ps(res368, res2);
770
771         __m128 res370 = _mm_dp_ps(res369, res3);
772
773         __m128 res371 = _mm_dp_ps(res370, res0);
774
775         __m128 res372 = _mm_dp_ps(res371, res1);
776
777         __m128 res373 = _mm_dp_ps(res372, res2);
778
779         __m128 res374 = _mm_dp_ps(res373, res3);
780
781         __m128 res375 = _mm_dp_ps(res374, res0);
782
783         __m128 res376 = _mm_dp_ps(res375, res1);
784
785         __m128 res377 = _mm_dp_ps(res376, res2);
786
787         __m128 res378 = _mm_dp_ps(res377, res3);
788
789         __m128 res379 = _mm_dp_ps(res378, res0);
790
791         __m128 res380 = _mm_dp_ps(res379, res1);
792
793         __m128 res381 = _mm_dp_ps(res380, res2);
794
795         __m128 res382 = _mm_dp_ps(res381, res3);
796
797         __m128 res383 = _mm_dp_ps(res382, res0);
798
799         __m128 res384 = _mm_dp_ps(res383, res1);
800
801         __m128 res385 = _mm_dp_ps(res384, res2);
802
803         __m128 res386 = _mm_dp_ps(res385, res3);
804
805         __m128 res387 = _mm_dp_ps(res386, res0);
806
807         __m128 res388 = _mm_dp_ps(res387, res1);
808
809         __m128 res389 = _mm_dp_ps(res388, res2);
810
811         __m128 res390 = _mm_dp_ps(res389, res3);
812
813         __m128 res391 = _mm_dp_ps(res390, res0);
814
815         __m128 res392 = _mm_dp_ps(res391, res1);
816
817         __m128 res393 = _mm_dp_ps(res392, res2);
818
819         __m128 res394 = _mm_dp_ps(res393, res3);
820
821         __m128 res395 = _mm_dp_ps(res394, res0);
822
823         __m128 res396 = _mm_dp_ps(res395, res1);
824
825         __m128 res397 = _mm_dp_ps(res396, res2);
826
827         __m128 res398 = _mm_dp_ps(res397, res3);
828
829         __m128 res399 = _mm_dp_ps(res398, res0);
830
831         __m128 res400 = _mm_dp_ps(res399, res1);
832
833         __m128 res401 = _mm_dp_ps(res400, res2);
834
835         __m128 res402 = _mm_dp_ps(res401, res3);
836
837         __m128 res403 = _mm_dp_ps(res402, res0);
838
839         __m128 res404 = _mm_dp_ps(res403, res1);
840
841         __m128 res405 = _mm_dp_ps
```

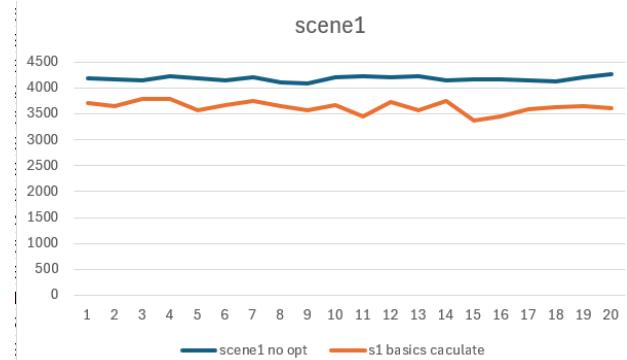


Figure 2: Scene1 Caculations Optimization

```

    __m128 matRow = _mm_load_ps(&A.a[row * 4]); // load row

    __m128 dp = _mm_dp_ps(matRow, vec, 0x1F1);
    ; // mask 0x1F1: multiply all, store in
    lowest
    out[row] = _mm_cvtsd_f32(dp);
    // extract lowest float
}

```

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.

```
1 void clear() {
2     // could also use fill_n
3     _m256 onef = _mm256_set1_ps(1.0f);
4     for (unsigned int i = 0; i < width * height;
5          i += 8) {
6         _mm256_store_ps(&buffer[i], onef);
7     }
}
```

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

```
1 class alignas(16) colour_opt {
2 public:
3     union {
4         struct {
5             float r, g, b; // Red, Green, and
6             Blue components of the colour
7             float padding;
8         };
9         float rgb[4];      // Array
10        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

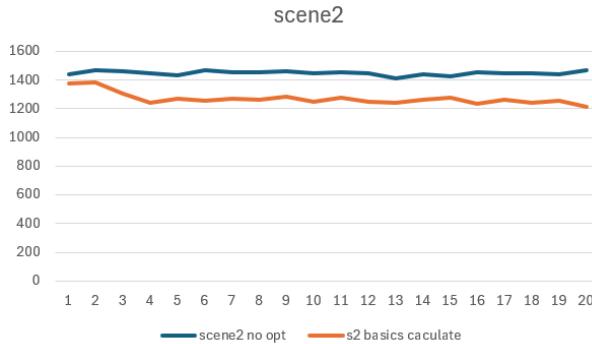


Figure 3: Scene2 Calculations Optimization

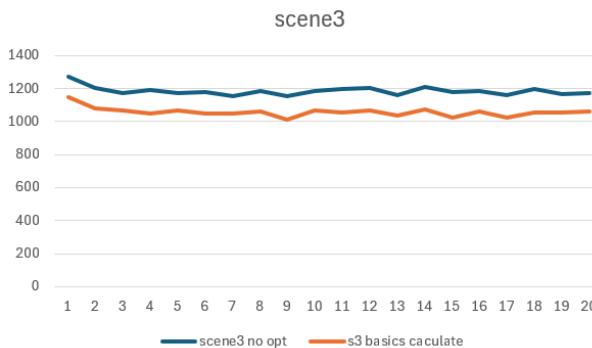


Figure 4: Scene3 Calculations Optimization

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.

```

1 for (int i = 0; i < vCount; ++i){
2     Vertex& out = vsCache[i];
3     const Vertex& in = mesh->vertices[i];
4
5     // view pos
6     view_positions[i] = view * in.p;
7
8     // clip space
9     out.p = p * in.p;
10    out.p.divideW();
11
12    // viewport transform
13    out.p[0] = (out.p[0] + 1.f) * 0.5f *
14        renderer.canvas.getWidth();
15    out.p[1] = (out.p[1] + 1.f) * 0.5f *
16        renderer.canvas.getHeight();
17    out.p[1] = renderer.canvas.getHeight() - out
18        .p[1];
19
20    // normal (world space)
21    out.normal = mesh->world * in.normal;
22    out.normal.normalise();
23
24    out.rgb = in.rgb;
25 }
```

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

```

1 for (unsigned int triIdx = 0; triIdx < triCount;
2       ++triIdx) {
3     triIndices& ind = mesh->triangles[triIdx];
4
5     vec4& v0_view = view_positions[ind.v[0]];
6     vec4& v1_view = view_positions[ind.v[1]];
7     vec4& v2_view = view_positions[ind.v[2]];
8     vec4 e1 = v1_view - v0_view;
9     vec4 e2 = v2_view - v0_view;
10    vec4 view_normal = vec4::cross(e1, e2);
11    if (vec4::dot(view_normal, -v0_view) >= 0.0f)
12    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,

$$E(x, y) = A * x + B * y + C \quad (1)$$

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.

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

$$= A * x + A + B * y + C \quad (3)$$

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 pre-calculated 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.

```

1 eg.
2 const float invArea = 1.0f / area;
3
4 const float dz_dx = (v[0].p[2] * e0.x + v[1].p
5 [2] * e1.x + v[2].p[2] * e2.x) * invArea;
6 const float dz_dy = (v[0].p[2] * e0.y + v[1].p
7 [2] * e1.y + v[2].p[2] * e2.y) * invArea;
8
9 const vec4 dn_dx = (v[0].normal * e0.x + v[1].
10 normal * e1.x + v[2].normal * e2.x) *
```

```

8  const vec4 dn_dy = (v[0].normal * e0.y + v[1].
9    normal * e1.y + v[2].normal * e2.y) *
10   invArea;
11
12 // --- SIMD constants ---
13 const __m256 zero = _mm256_setzero_ps();
14 //const __m256 lane = _mm256_set_ps(7, 6, 5, 4,
15 //    3, 2, 1, 0);
16 const __m256 lane = _mm256_setr_ps(0.0f, 1.0f,
17 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.

```

1 for (int y = minY; y <= maxY; ++y){
2     int x = minX;
3     for (; x <= maxX - 7; x += 8) {
4         ...
5         w0v = _mm256_add_ps(w0v, w0_step8);
6         w1v = _mm256_add_ps(w1v, w1_step8);
7         w2v = _mm256_add_ps(w2v, w2_step8);
8         zv = _mm256_add_ps(zv, z_step8);
9
10        nx = _mm256_add_ps(nx, n_step8x);
11        ny = _mm256_add_ps(ny, n_step8y);
12        nz = _mm256_add_ps(nz, n_step8z);
13
14        cr = _mm256_add_ps(cr, c_step8r);
15        cg = _mm256_add_ps(cg, c_step8g);
16        cb = _mm256_add_ps(cb, c_step8b);
17    }
18    w0_row += w0_dy;
19    w1_row += w1_dy;
20    w2_row += w2_dy;
21    z_row += dz_dy;
22    n_row += dn_dy;
23    c_row += dc_dy;
24 }

```

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

```

1 __m256 w0v_zero = _mm256_cmp_ps(w0v, zero,
2     CMP_GE_OQ);
3 __m256 w1v_zero = _mm256_cmp_ps(w1v, zero,
4     CMP_GE_OQ);
5 __m256 w2v_zero = _mm256_cmp_ps(w2v, zero,
6     CMP_GE_OQ);
7 __m256 inside = _mm256_and_ps(w0v_zero, w1v_zero
8     );
9 inside = _mm256_and_ps(inside, w2v_zero);
10 int mask = _mm256_movemask_ps(inside);
11 if(mask == 0)
12 {
13     w0v = _mm256_add_ps(w0v, w0_step8);
14     w1v = _mm256_add_ps(w1v, w1_step8);
15     w2v = _mm256_add_ps(w2v, w2_step8);
16     zv = _mm256_add_ps(zv, z_step8);
17
18     nx = _mm256_add_ps(nx, n_step8x);
19     ny = _mm256_add_ps(ny, n_step8y);
20     nz = _mm256_add_ps(nz, n_step8z);
21     continue;
22 }

```

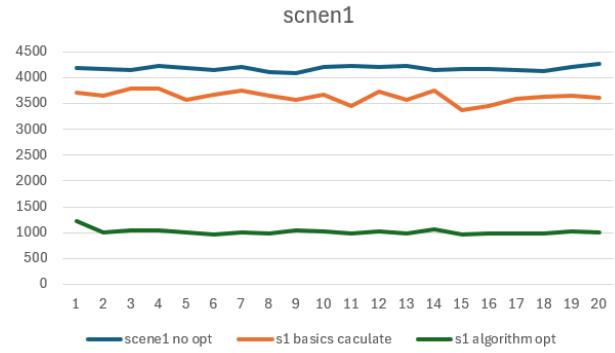


Figure 5: Scene1 Algorithm Optimization

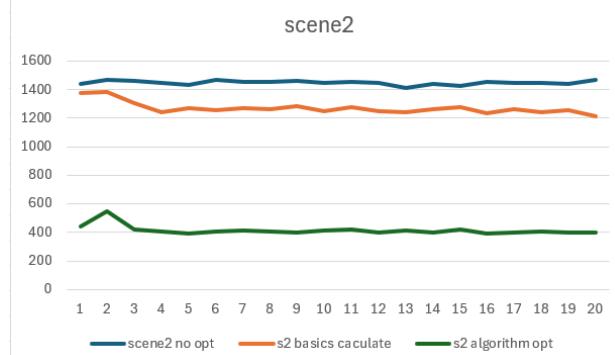


Figure 6: Scene2 Algorithm Optimization

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.

```

1 int m = final_mask;
2 while (m)
3 {
4     // count trailing zeros
5     int i = _tzcnt_u32(m);
6     m &= m - 1;
7     renderer.canvas.draw(
8         x + i, y,
9         (unsigned char)(rr[i]),
10        (unsigned char)(gg[i]),
11        (unsigned char)(bb[i])
12    );
13     renderer.zbuffer(x + i, y) = zz[i];
14 }

```

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.

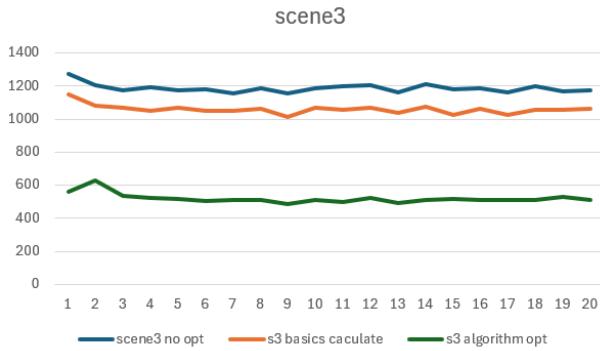


Figure 7: Scene3 Algorithm Optimization

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

```

1 class Mesh_SoA{
2 public:
3 colour col;
4 float kd;
5 float ka;
6 matrix world;
7
8 alignas(32) std::vector<float> positions_x,
9     positions_y, positions_z, positions_w;
10 alignas(32) std::vector<float> normals_x,
11     normals_y, normals_z;
12 alignas(32) std::vector<float> colors_r,
13     colors_g, colors_b;
14 std::vector<triIndices> triangles; // List of
15     triangles in the mesh
16 }
```

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

```

1 int i = 0;
2 for (; i + 7 < vCount; i += 8) {
3     // view positon caculate
4
5     // cam position
6     _m256 outX = _mm256_add_ps(
7         _mm256_add_ps(_mm256_mul_ps(p00, vx),
8             _mm256_mul_ps(p01, vy)),
9             _mm256_add_ps(_mm256_mul_ps(p02, vz),
10                _mm256_mul_ps(p03, vw)))
11 );
12 // y z w
13
14 _m256 invW = _mm256_rcp_ps(outW); // fast
15 approx
16 // Newton-Raphson refine for better
17 precision
18 invW = _mm256_mul_ps(invW, _mm256_sub_ps(
19     _mm256_set1_ps(2.0f), _mm256_mul_ps(outW,
20     invW)));
21
22 // normal matrix transformation
23 _m256 nx = _mm256_load_ps(&mesh->normals_x[
24     i]);
25 _m256 ny = _mm256_load_ps(&mesh->normals_y[
```

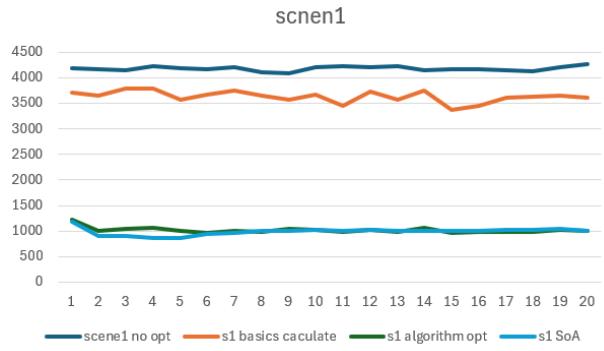


Figure 8: Scene1 Data Structures Optimization

```

26     _m256 nz = _mm256_load_ps(&mesh->normals_z[i]);
27
28     _m256 outNX = _mm256_add_ps(
29         _mm256_add_ps(_mm256_mul_ps(w00, nx),
30             _mm256_mul_ps(w01, ny)),
31             _mm256_mul_ps(w02, nz))
32 ); // normal y z
33
34 // normalize
35 _m256 lenSq = _mm256_add_ps(_mm256_add_ps(
36     _mm256_mul_ps(outNX, outNX),
37         _mm256_mul_ps(outNY, outNY)),
38         _mm256_mul_ps(outNZ, outNZ));
39 _m256 invLen = _mm256_sqrt_ps(lenSq);
40 invLen = _mm256_mul_ps(invLen, _mm256_sub_ps(
41     _mm256_set1_ps(1.5f),
42         _mm256_mul_ps(_mm256_set1_ps(0.5f),
43             _mm256_mul_ps(lenSq, _mm256_mul_ps(
44                 invLen, invLen))))))
44 outNX = _mm256_mul_ps(outNX, invLen);
45 outNY = _mm256_mul_ps(outNY, invLen);
46 outNZ = _mm256_mul_ps(outNZ, invLen);
47 // store in vertex cache
48 for (int j = 0; j < 8; ++j) {
49     Vertex& out = vsCache[i + j];
50     out.p = vec4(((float*)&outX)[j], ((float*)
51     &outY)[j], ((float*)&outZ)[j], 1.f);
52     out.normal = vec4(((float*)&outNX)[j],
53         ((float*)&outNY)[j], ((float*)&outNZ)[j], 0.
54 f);
55     out.rgb.set(mesh->colors_r[i + j], mesh
56     ->colors_g[i + j], mesh->colors_b[i + j]);
57 }
```

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

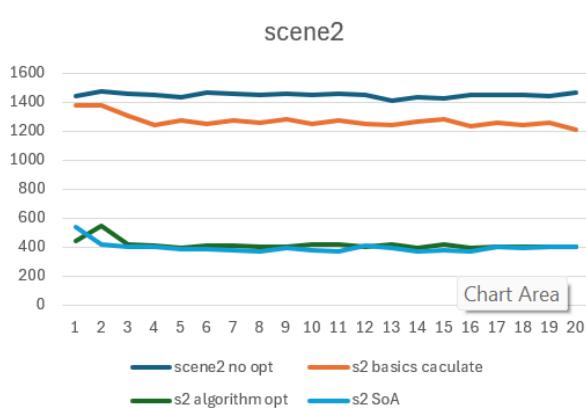


Figure 9: Scene2 Data Structures Optimization

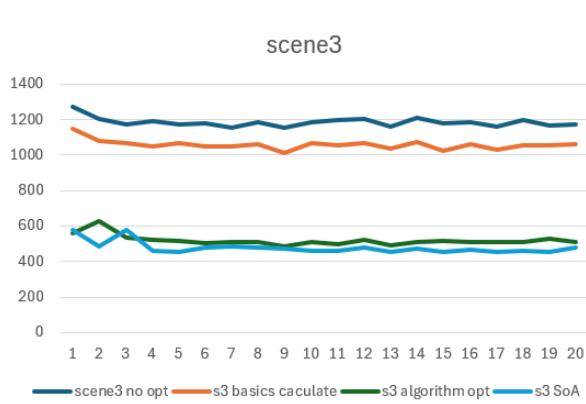


Figure 10: Scene3 Data Structures Optimization

'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.

```

1 class ThreadPool {
2 private:
3     std::vector<std::thread> workers;
4
5     // task queue
6     std::queue<std::shared_ptr<Task>> tasks;
7
8     // lock
9     mutable std::mutex queue_mutex;
10    std::mutex complete_mutex;
11    std::condition_variable condition;
12    std::condition_variable complete_cv;
13
14    // status flags
15    std::atomic<bool> stop;
16    std::atomic<int> working;
17    std::atomic<int> task_num;
18 };

```

The enqueue() 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.

```

1     // submit task
2     void enqueue(std::function<void()> job) {
3         {
4             std::unique_lock<std::mutex> lock(
5                 queue_mutex);
5             if (stop) {
6                 throw std::runtime_error("Cannot
7                 enqueue on stopped ThreadPool");
7             }
8             tasks.emplace(std::make_shared<
9                 StdFunctionTask>(std::move(job)));
10            ++task_num;
11        }
12        condition.notify_one();
13    }
14
15     // wait for all tasks complete
16     void waitAll() {
17         std::unique_lock<std::mutex> lock(
18             complete_mutex);
19         complete_cv.wait(lock, [this]{
20             return tasks.empty() && working == 0
21             && task_num == 0;
22         });
23     }

```

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.

```

1 void workerLoop(int thread_id) {
2     while (true) {
3         std::shared_ptr<Task> task;
4         // get task
5         std::unique_lock<std::mutex> lock(
6             queue_mutex);
7
8         // wait for task
9         condition.wait(lock, [this]() {
10             return stop || (!tasks.empty());
11         });
12
13         if (stop && tasks.empty()) {
14             return;
15         }
16
17         // get task
18         task = std::move(tasks.front());
19         tasks.pop();
20         working++;
21
22         // do task
23         task->execute();
24         std::unique_lock<std::mutex> lock(
25             queue_mutex);
26         working--;
27         task_num--;
28
29         // notify all
30         if (tasks.empty() && working == 0) {
31             std::lock_guard<std::mutex> lock(
32                 complete_mutex);
33             complete_cv.notify_all();
34         }
35     }
36 }
```

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 $n \times n$ tiles is slower than dividing it into $1 \times n$ 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.

```
1 class Tile {
```

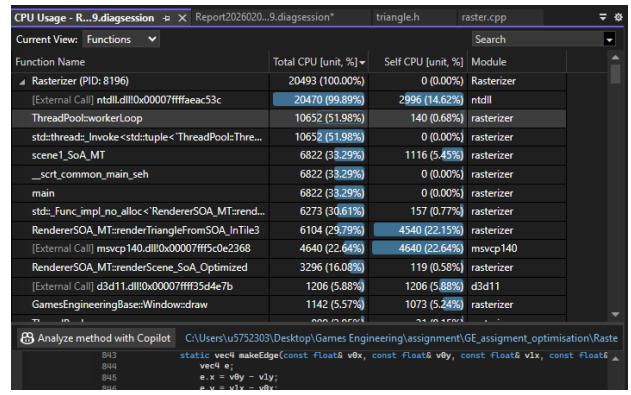


Figure 11: Scene1 Multithread Profiler

```

2 public:
3     int x, y;           // top left coordinates
4     int width, height; // size
5     Zbuffer<float> zbuffer;
6     std::vector<unsigned char> colors;
7
8     Tile() : x(0), y(0), width(0), height(0) {}
9
10    Tile(int _x, int _y, int _w, int _h) : x(_x)
11        , y(_y), width(_w), height(_h) {
12        //zbuffer.resize(width * height, std::
13        numeric_limits<float>::max());
14        zbuffer.create(width, height);
15        colors.resize(width * height * 3, 0);
16    }
17
18    class RendererSOA_MT {
19    private:
20        int TILE_WIDTH;
21        int TILE_HEIGHT;
22        std::vector<Tile> tiles;
23        int tiles_x, tiles_y;
24        RendererSOA_MT(int canvas_width, int
25                        canvas_height, ThreadPool& pool)
26        : TILE_WIDTH(1024), TILE_HEIGHT(96){
27            TILE_HEIGHT = canvas_height / pool.
28            getThreadCount();
29
30            // caculate tiles count
31            tiles_x = (canvas_width + TILE_WIDTH - 1) /
32            TILE_WIDTH;
33            tiles_y = (canvas_height + TILE_HEIGHT - 1) /
34            TILE_HEIGHT;
35
36            tiles.reserve(tiles_x * tiles_y);
37            // create tiles
38            for (int ty = 0; ty < tiles_y; ++ty) {
39                for (int tx = 0; tx < tiles_x; ++tx) {
40                    int tile_x = tx * TILE_WIDTH;
41                    int tile_y = ty * TILE_HEIGHT;
42                    int tile_w = std::min(TILE_WIDTH,
43                        canvas_width - tile_x);
44                    int tile_h = std::min(TILE_HEIGHT,
45                        canvas_height - tile_y);
46
47                    tiles.emplace_back(tile_x, tile_y,
48                        tile_w, tile_h);
49                }
50            }
51        };
52    };
53 }
```

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.

```

1 // triangle binning to tiles
2 const int triCount = (int)transformed_mesh.
    triangles.size();
3 std::vector<std::vector<int>> triangle_buckets(
    tileCount);
4
5 {const float* pos_x = transformed_mesh.
    transformed_positions_x.data();
6 const float* pos_y = transformed_mesh.
    transformed_positions_y.data();
7 float min_x = std::min({ pos_x[idx.v[0]], pos_x[
    idx.v[1]], pos_x[idx.v[2]] });
8 float max_x = std::max({ pos_x[idx.v[0]], pos_x[
    idx.v[1]], pos_x[idx.v[2]] });
9 float min_y = std::min({ pos_y[idx.v[0]], pos_y[
    idx.v[1]], pos_y[idx.v[2]] });
10 float max_y = std::max({ pos_y[idx.v[0]], pos_y[
    idx.v[1]], pos_y[idx.v[2]] });
11
12 min_x = std::max(0.0f, min_x);
13 max_x = std::min((float)renderer.canvas.getWidth
    () - 1, max_x);
14 min_y = std::max(0.0f, min_y);
15 max_y = std::min((float)renderer.canvas.
    getHeight() - 1, max_y);
16 if (min_x > max_x || min_y > max_y) continue;
17
18 int min_tx = int(std::floor(min_x)) / TILE_WIDTH
    ;
19 int max_tx = int(std::ceil(max_x)) / TILE_WIDTH;
20 int min_ty = int(std::floor(min_y)) /
    TILE_HEIGHT;
21 int max_ty = int(std::ceil(max_y)) / TILE_HEIGHT
    ;
22
23 min_tx = std::max(0, min_tx);
24 max_tx = std::min(tiles_x - 1, max_tx);
25 min_ty = std::max(0, min_ty);
26 max_ty = std::min(tiles_y - 1, max_ty);
27
28 for (int ty = min_ty; ty <= max_ty; ++ty) {
29     for (int tx = min_tx; tx <= max_tx; ++tx) {
30         int tile_idx = ty * tiles_x + tx;
31         triangle_buckets[tile_idx].push_back(t);
32     }
33 }

```

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

```

1 // draw
2 for (int tile_idx = 0; tile_idx < tileCount; ++
    tile_idx) {
3     if (!triangle_buckets[tile_idx].empty()) {
4         pool.enqueue([&, tile_idx](){
5             Tile& tile = tiles[tile_idx];
6             // tile.zbuffer.clear();
7             // std::fill(tile.colors.begin(),
    tile.colors.end(), 0);
8             for (int tri_idx : triangle_buckets[
    tile_idx]) {
9                 const triIndices& ind =
    transformed_mesh.triangles[tri_idx];
10                // clip out of the screen

```

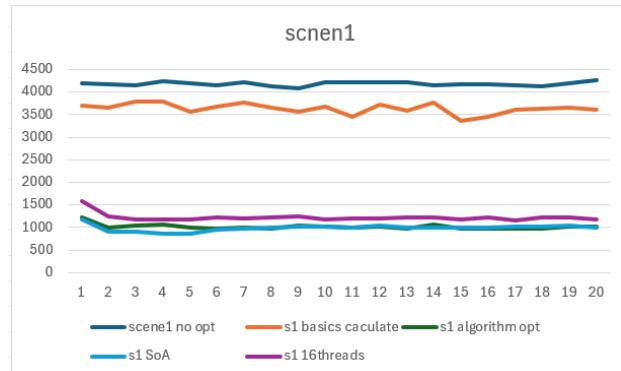


Figure 12: Scene1 Mutithreading Optimization

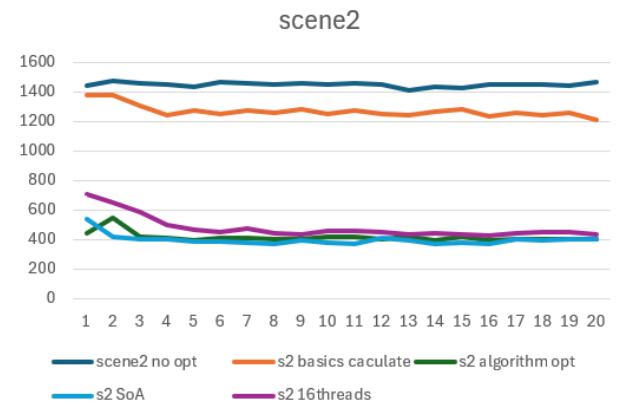


Figure 13: Scene2 Mutithreading Optimization

```

        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]];
14     if (!(fabs(z0) > 1.0f || fabs(z1)
) > 1.0f || fabs(z2) > 1.0f)) {
15
16     renderTriangleFromSOA_InTile3( renderer,
    transformed_mesh, light, ka, kd, tri_idx ,
    tile );
17 }
18 pool.waitForAll();

```

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.

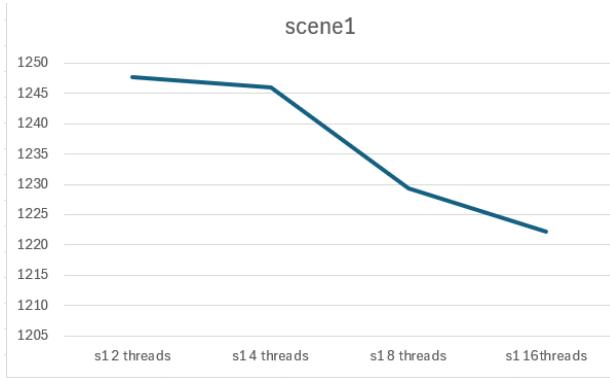


Figure 14: Scene1 Using different Number of Thread

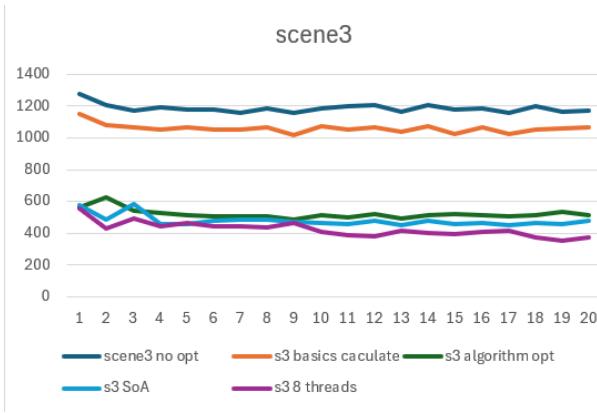


Figure 15: Scene3 Mutithreading Optimization

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.

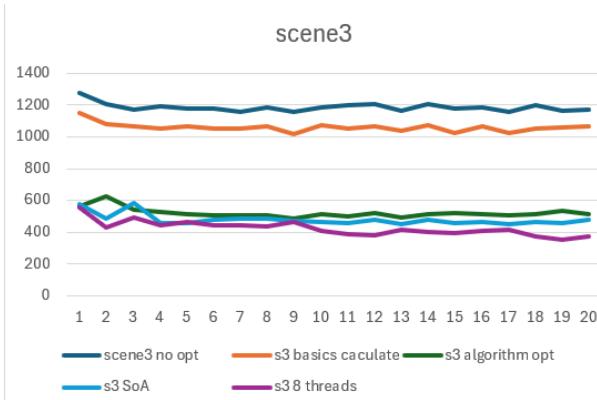


Figure 16: Scene3 Using different Number of Thread

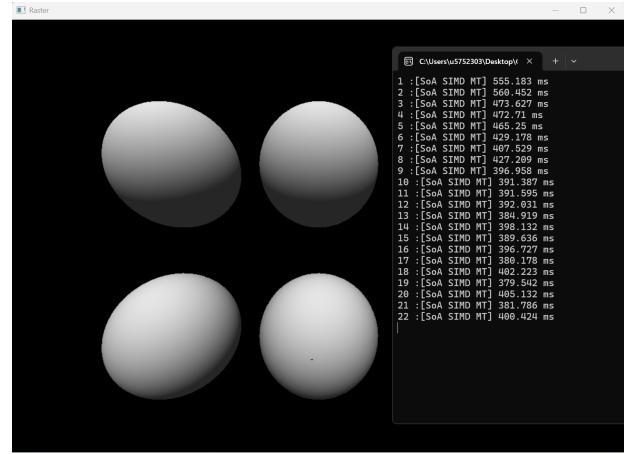


Figure 17: Scene3 Mutithreading Working Output

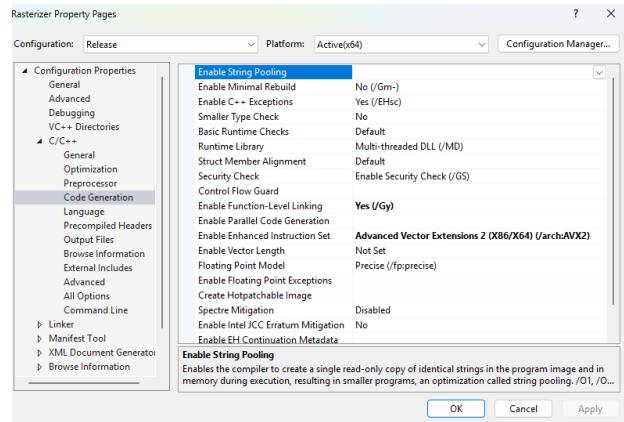


Figure 18: Settings

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.

```

1 enum class MessageType {
2     // connect to the server
3     CLIENT_CONNECT = 1,
4     CLIENT_DISCONNECT = 2,
5     // public and private message
6     PUBLIC_MESSAGE = 3,
7     PRIVATE_MESSAGE = 4,
8     // userlist message for update the online
9     user
10    USER_LIST_UPDATE = 5,
11 };
12 struct MessageHeader {
13     MessageType type;
14     MessageHeader() : type(MessageType::CLIENT_CONNECT) {}

```

```

15     MessageHeader(MessageType t, unsigned int s)
16     : type(t) {s}

```

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.

```

1 struct ClientConnectMessage {
2     char username[32];
3 };
4 struct PublicMessage {
5     char sender[32];
6     char content[256];
7 };
8 struct PrivateMessage {
9     char sender[32];
10    char target[32];
11    char content[256];
12 };
13 struct UserListMessage {
14     int user_count;
15     char users[32][32];
16 };

```

2.2.2 Server Design

The "serversocket" store the server's socket to listen 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 released.

```

1 class ChatServer {
2 public:
3     SOCKET server_socket;
4     std::unordered_map<SOCKET, std::string>
5         clients;
6     std::mutex clients_mutex;
7     bool running;
8     bool init(int port) {
9         // Step 1: Initialize WinSock
10        WSADATA wsaData;
11        if (WSAStartup(MAKEWORD(2, 2), &wsaData)
12            != 0) {
13            std::cerr << "WSAStartup failed with
14            error: " << WSAGetLastError() << std::endl;
15            return false;
16        }
17        // create socket
18        server_socket = socket(AF_INET,
19                               SOCK_STREAM, IPPROTO_TCP);

```

```
17     if (server_socket == INVALID_SOCKET) {
18         std::cerr << "Socket creation failed
19         " << std::endl;
20         WSACleanup();
21         return false;
22     }
23     // bind port and address
24     sockaddr_in serverAddr;
25     serverAddr.sin_family = AF_INET;
26     serverAddr.sin_port = htons(port);
27     serverAddr.sin_addr.s_addr = INADDR_ANY;
28     if (bind(server_socket, (sockaddr*)&
29     serverAddr, sizeof(serverAddr)) ==
30     SOCKET_ERROR) {
31         std::cerr << "Bind failed" << std::
32         endl;
33         closesocket(server_socket);
34         WSACleanup();
35         return false;
36     }
37     if (listen(server_socket, SOMAXCONN) ==
38     SOCKET_ERROR) {
39         std::cerr << "Listen failed" << std
40         ::endl;
41         closesocket(server_socket);
42         WSACleanup();
43         return false;
44     }
45     running = true;
46     std::cout << "Chat Server started on
47     port " << port << std::endl;
48
49     std::thread
50     // detach thread, when close the server,
51     // the while loop
52     // in thread will break, thread will be
53     release
54     acceptThread(&ChatServer::accept_client,
55     this);
56     acceptThread.detach();
57     return true;
58 }
59 };
```

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.

```
1 void accept_client() {
2     while (running) {
3         sockaddr_in client_address;
4         int len = sizeof(client_address);
5         SOCKET client_socket = accept(
6             server_socket, (sockaddr*)&client_address, &
7             len);
8         if (client_socket == INVALID_SOCKET) {
9             if (running)
10                 std::cerr << "Accept failed" <<
11                 std::endl;
12             continue;
13         }
14         char clientIP [INET_ADDRSTRLEN];
15         inet_ntop(AF_INET, &client_address.
16             sin_addr, clientIP, INET_ADDRSTRLEN);
17         std::cout << "Accept connection from: "
18         << clientIP << ":" << ntohs(client_address.
19             sin_port) << std::endl;
20         // create client thread
21         std::thread clientThread(&ChatServer::
22             handle_client, this, client_socket);
23         clientThread.detach();
24     }
25 }
```

The handle client function is used to parse the message content sent by the client and perform corresponding operations. After the server and client build the connections, the client needs to send a connection message to tell the server what its username is. 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.

```
1 int receive_message(SOCKET socket, char* buffer,
2         int size){
3     int totalReceived = 0;
4     while (totalReceived < size) {
5         int received = recv(socket, buffer + totalReceived, size - totalReceived, 0);
6         if (received <= 0) {
7             return received;
8         }
9     }
10    totalReceived += received;
11    return totalReceived;
12 }
13 void handle_client(SOCKET client_socket) {
14     std::string username;
15     MessageHeader header;
16     if (receive_message(client_socket, (char*)& header, sizeof(header)) != sizeof(header)) {
17         std::cout << "Failed to receive header"
18         << std::endl;
19         close_client(client_socket, username);
20         return;
21     }
22     if (header.type != MessageType::CLIENT_CONNECT) {
23         close_client(client_socket, username);
24         return;
25     }
26     ClientConnectMessage connect_message;
27     if (receive_message(client_socket, (char*)& connect_message, sizeof(connect_message)) != sizeof(connect_message)) {
28         std::cout << "Failed to receive connect message" << std::endl;
29         close_client(client_socket, username);
30         return;
31     }
32     username = connect_message.username;
33     // create client
34     {
35         std::lock_guard<std::mutex> lock(
36             clients_mutex);
37         clients[client_socket] = username;
38     }
39     std::cout << "User " << username << " joined
40     the room" << std::endl;
41
42     // send to new user
43     send_userlist(client_socket);
44     // send a public message to all user
45     PublicMessage message("System", username + " joined the chat");
46     MessageHeader send_header(MessageType::PUBLIC_MESSAGE, sizeof(PublicMessage));
47
48     {
49         std::lock_guard<std::mutex> lock(
50             clients_mutex);
51         for (const auto& client : clients) {
```

```

45         // not send to myself
46         if (client.second != username) {
47             send(client.first, (char*)&
48             send_header, sizeof(send_header), 0);
49             send(client.first, (char*)&
50             message, sizeof(message), 0);
51         }
52     broadcast_userlist();
53
54     while (running) {
55         // information processing logic
56     }
57     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.

```

1 while (running) {
2     if (receive_message(client_socket, (char*)&
3         header, sizeof(header)) != sizeof(header)) {
4         std::cout << "Client '" << username << "
5         , disconnected" << std::endl;
6         break;
7     }
8
9     if (header.type == MessageType::
10         PUBLIC_MESSAGE) {
11         PublicMessage message;
12
13         if (receive_message(client_socket, (char*)
14             &message, sizeof(message)) != sizeof(
15             message)) {
16             std::cout << "Failed to receive
17             public message from " << username << std::
18             endl;
19             break;
20
21             std::cout << "Public message from " <<
22             message.sender << ":" << message.content <<
23             std::endl;
24
25             MessageHeader header(MessageType::
26                 PUBLIC_MESSAGE, sizeof(PublicMessage));
27
28             for (const auto& client : clients) {
29                 send(client.first, (char*)&header,
30                     sizeof(header), 0);
31                 send(client.first, (char*)&message,
32                     sizeof(message), 0);
33             }
34         }
35     }
36
37     while (running) {
38         // information processing logic
39     }
40     close_client(client_socket, username);}
```

```

29     }
30
31         std::cout << "Private message from " <<
32         message.sender << " to " << message.target
33         << std::endl;
34
35         std::lock_guard<std::mutex> lock(
36             clients_mutex);
37
38         // search target
39         SOCKET targetSocket = INVALID_SOCKET;
40         for (const auto& client : clients) {
41             if (client.second == message.target)
42             {
43                 targetSocket = client.first;
44                 break;
45             }
46
47             if (targetSocket != INVALID_SOCKET) {
48                 MessageHeader header(MessageType::
49                     PRIVATE_MESSAGE, sizeof(PrivateMessage));
50                 send(targetSocket, (char*)&header,
51                     sizeof(header), 0);
52                 send(targetSocket, (char*)&message,
53                     sizeof(message), 0);
54             }
55         }
56
57         else if (header.type == MessageType::
58             CLIENT_DISCONNECT) {
59             std::cout << "Client " << username << "
60             requested disconnect" << std::endl;
61             break;
62         }
63         else {
64             std::cout << " unknown " << username <<
65             std::endl;
66         }
67     }
```

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

```

1 void send_userlist(SOCKET target) {
2     std::lock_guard<std::mutex> lock(
3         clients_mutex);
4
5     UserListMessage list;
6     list.user_count = 0;
7
8     // collect all username
9     for (const auto& client : clients) {
10         strncpy_s(list.users[list.user_count],
11             client.second.c_str(), sizeof(list.users[
12                 list.user_count]) - 1);
13         list.users[list.user_count][sizeof(list.
14             users[list.user_count]) - 1] = '\0';
15         list.user_count++;
16     }
17     MessageHeader header(MessageType::
18         USER_LIST_UPDATE, sizeof(UserListMessage));
19
20     send(target, (char*)&header, sizeof(header),
21         0);
22     send(target, (char*)&list, sizeof(list), 0);
23 }
```

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

```

1 void close() {
2     running = false;
3
4     // close linsten socket
5     if (server_socket != INVALID_SOCKET) {
```

```

6     closesocket(server_socket);
7     server_socket = INVALID_SOCKET;
8 }
9
10 // close all clients {
11     std::lock_guard<std::mutex> lock(
12         clients_mutex);
13     for (auto& client : clients)
14     {
15         closesocket(client.first);
16     }
17     clients.clear();
18 }
19 WSACleanup();
20 std::cout << "Server stopp" << std::endl;
20 }
```

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.

```

1 enum class NetworkEventType {
2     CONNECTED = 0, // didn't use
3     DISCONNECTED,
4     PUBLIC_MESSAGE,
5     PRIVATE_MESSAGE,
6     USER_LIST_UPDATE
7 };
8 struct NetworkEvent {
9     NetworkEventType type;
10    std::string sender;
11    std::string text;
12    std::string target;
13    std::vector<std::string> users;
14
15    NetworkEvent() : type(NetworkEventType::CONNECTED) {}
16    NetworkEvent(NetworkEventType t) : type(t) {}
17};
```

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.

```

1 struct ChatMessage {
2     std::string sender;
3     std::string target;
4     std::string text;
5     bool isPrivate;
6     ChatMessage(const std::string& s = "", const
7                 std::string& t = "", bool priv = false,
8                 const std::string& tar = "")
9         : sender(s), text(t), isPrivate(priv),
10           target(tar) {}
11
12 class ChatWindow {
13 public:
14     std::string username;
15     std::vector<std::string> users_online;
16     std::vector<ChatMessage> public_message;
17     std::map<std::string, std::vector<
18             ChatMessage>> private_chat;
19
20     // input buffer
21     char public_input[200];
22     std::map<std::string, std::array<char, 200>>
23         private_input;
24
25     bool connected;
26
27     // own socket
28     SOCKET client_socket;
29     std::thread recieve_thread;
30     std::atomic<bool> running;
31     std::queue<NetworkEvent> event_queue;
32     std::mutex event_mutex;
33
34     FMOD::System* system;};

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.

```

1 bool ChatWindow::connect_server(const std::
2     string& ip, int port, const std::string&
3     user_name) {
4     // Step 1: Initialize WinSock
5     WSADATA wsaData;
6     if (WSAStartup(MAKEWORD(2, 2), &wsaData) !=
7         0) {
8         std::cerr << "WSAStartup failed with
9         error: " << WSAGetLastError() << std::endl;
10        return false;
11    }
12
13    // Step 2: Create a socket
14    client_socket = socket(AF_INET, SOCK_STREAM,
15                           IPPROTO_TCP);
16    if (client_socket == INVALID_SOCKET) {
17        std::cerr << "Socket creation failed
18        with error: " << WSAGetLastError() << std::endl;
19        WSACleanup();
20        return false;
21    }
22
23    // set server address
24    sockaddr_in serverAddr;
25    serverAddr.sin_family = AF_INET;
26    serverAddr.sin_port = htons(port);
27    if (inet_pton(AF_INET, ip.c_str(), &
28                  serverAddr.sin_addr) <= 0) {
29        std::cout << "Invalid address" << std::
30        endl;
31    }
32}
```

```

22     if (client_socket != INVALID_SOCKET) {
23         closesocket(client_socket);
24         client_socket = INVALID_SOCKET;
25     }
26     WSACleanup();
27     return false;
28 }
29 // connect server
30 if (connect(client_socket, (sockaddr*)&serverAddr, sizeof(serverAddr)) ==
31     SOCKET_ERROR) {
32     std::cout << "connect failed" << std::endl;
33     if (client_socket != INVALID_SOCKET) {
34         closesocket(client_socket);
35         client_socket = INVALID_SOCKET;
36     }
37     WSACleanup();
38     return false;
39 }
40 ClientConnectMessage connect_message(
41     user_name);
42 if (!send_message_toserver(MessageType:::
43     CLIENT_CONNECT, &connect_message, sizeof(
44     connect_message))){
45     std::cerr << "Failed to send connect
46     message" << std::endl;
47     if (client_socket != INVALID_SOCKET) {
48         closesocket(client_socket);
49         client_socket = INVALID_SOCKET;
50     }
51     WSACleanup();
52     return false;
53 }
54 username = user_name;
55 connected = true;
56 running = true;
57 // receive thread
58 recieve_thread = std::thread(&ChatWindow::
59     receive_message, this);
60 }

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.

```

```

1 void ChatWindow::receive_message() {
2     while (running && client_socket !=
3         INVALID_SOCKET) {
4         MessageHeader header;
5         int received = recv(client_socket, (char
6         *)&header, sizeof(header), 0);
7         if (received <= 0) {
8             NetworkEvent event(NetworkEventType
9             ::DISCONNECTED);
10            event_queue.push(event);
11            break;
12        }
13        NetworkEvent event;
14        switch (header.type) {
15            case MessageType::PUBLIC_MESSAGE: {

```

```

16                recv(client_socket, (char*)&message,
17                sizeof(message), 0);
18                event.type = NetworkEventType::
19                PUBLIC_MESSAGE;
20                event.sender = message.sender;
21                event.text = message.content;
22                break;
23            case MessageType::PRIVATE_MESSAGE: {
24                PrivateMessage private_message;
25                recv(client_socket, (char*)&
26                private_message, sizeof(private_message), 0)
27                ;
28                event.type = NetworkEventType::
29                PRIVATE_MESSAGE;
30                event.sender = private_message.
31                sender;
32                event.target = private_message.
33                target;
34                event.text = private_message.content
35                ;
36                break;
37            case MessageType::USER_LIST_UPDATE: {
38                UserListMessage userlist;
39                recv(client_socket, (char*)&userlist,
40                sizeof(userlist), 0);
41                event.type = NetworkEventType::
42                USER_LIST_UPDATE;
43                for (int i = 0; i < userlist.
44                user_count; i++) {
45                    event.users.push_back(userlist.
46                    users[i]);
47                }
48                break;
49            default:
50                continue;
51            }
52            std::lock_guard<std::mutex> lock(
53                event_mutex);
54            event_queue.push(event);
55        }
56        // if not my message,paly sound
57        if (event.sender != username)
58            ChatWindow::play_music(event.type);
59    }

```

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.

```

1 void ChatWindow::process_event() {
2     while (!event_queue.empty()) {
3         // get event
4         NetworkEvent event;
5     }

```

```

6         std::lock_guard<std::mutex> lock(
7             event_mutex);
8             event = event_queue.front();
9             event_queue.pop();
10
11         switch (event.type) {
12             case NetworkEventType::DISCONNECTED:
13                 connected = false;
14                 close_connect();
15                 public_message.push_back(ChatMessage
("System", "Connect lost"));
16                 // clear all users
17                 users_online.clear();
18                 if (!username.empty()) {
19                     users_online.push_back(username)
20                 }
21                 break;
22
23             case NetworkEventType::PUBLIC_MESSAGE:
24                 if (event.sender != username)
25                     public_message.push_back(ChatMessage
(event.sender, event.text));
26                 break;
27
28             case NetworkEventType::PRIVATE_MESSAGE:{
29                 std::string name = (event.sender ==
username) ? event.target : event.sender;
30
31                 // check private chat map
32                 if (private_chat.find(name) ==
private_chat.end()){
33                     private_chat[name] = std::vector
<ChatMessage>();
34                 }
35                 // add chat message
36                 ChatMessage mess(event.sender, event
.text, true, event.target);
37                 private_chat[name].push_back(mess);
38
39                 if (private_input.find(name) ==
private_input.end()){
40                     private_input[name].fill('\0');
41                 }
42             }
43             break;
44         case NetworkEventType::USER_LIST_UPDATE:
45             // when someone link the server
46             update_userlist(event.users);
47             break;
48         default:
49             break;
50     } }
51
52         ImGui::SetNextWindowPos(ImVec2(600, 350)
53 , ImGuiCond_FirstUseEver);
54         if (ImGui::Begin("notification", nullptr
55 , ImGuiWindowFlags_NoCollapse)) {
56             ImGui::BeginChild("Messages", ImVec2
(0, 0), true);
57             for (const auto& message :
58             public_message) {
59                 if (message.sender == "System")
60                 {
61                     ImGui::PushStyleColor(
62                     ImGuiCol_Text, ImVec4(1.0f, 0.5f, 0.0f,
1.0f));
63                 }
64                 else {
65                     ImGui::PushStyleColor(
66                     ImGuiCol_Text, ImVec4(0.5f, 0.5f, 0.5f,
1.0f));
67                 }
68                 ImGui::Text("%s: %s", message.
69 sender.c_str(), message.text.c_str());
70                 ImGui::PopStyleColor();
71             }
72             ImGui::EndChild();
73         }
74         ImGui::End();
75     }
76     else {
77         // main
78         ImGui::SetWindowSize(ImVec2(800,
600), ImGuiCond_FirstUseEver);
79         ImGui::Begin("Chat Room", nullptr,
80         ImGuiWindowFlags_NoCollapse);
81
82         // user
83         user_win();
84         ImGui::SameLine();
85
86         // chat
87         ImGui::BeginGroup();
88         chat_win();
89         ImGui::EndGroup();
90
91         ImGui::End();
92     }
93 }
94
95     // private
96     private_win();
97 } }

```

2.3.2 Client UI Design

The main loop will call render_all 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.

```

1 void ChatWindow::render_all() {
2     process_event();
3     if (!connected) {
4         // login
5         login_win();
6         // Display some public messages
7         ImGui::SetWindowSize(ImVec2(600,
400), ImGuiCond_FirstUseEver);

```

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.

```

1 void ChatWindow::login_win() {
2     ImGui::SetNextWindowPos(ImVec2(100, 100),
3     ImGuiCond_FirstUseEver);
4     ImGui::SetWindowSize(ImVec2(500, 500),
3     ImGuiCond_FirstUseEver);
5     bool connect = false;
6
7     if (ImGui::Begin("Connect to Chat Server",
8         nullptr, ImGuiWindowFlags_NoCollapse)) {
9
10         static char username[64] = "Fox";
11         static char serverIP[64] = "127.0.0.1";
12         static char port[32] = "65432";
13
14         ImGui::Text("Username: %s", username);
15         ImGui::InputText("Username", username, 64);
16
17         ImGui::Text("Server IP: %s", serverIP);
18         ImGui::InputText("Server IP", serverIP, 64);
19
20         ImGui::Text("Port: %s", port);
21         ImGui::InputText("Port", port, 32);
22
23         if (ImGui::Button("Connect"))
24             connect = true;
25
26         if (connect)
27             connectserver(username, serverIP, port);
28
29         if (connect)
30             ImGui::Text("Connected!");
31         else
32             ImGui::Text("Connection failed!");
33
34     }
35 }

```

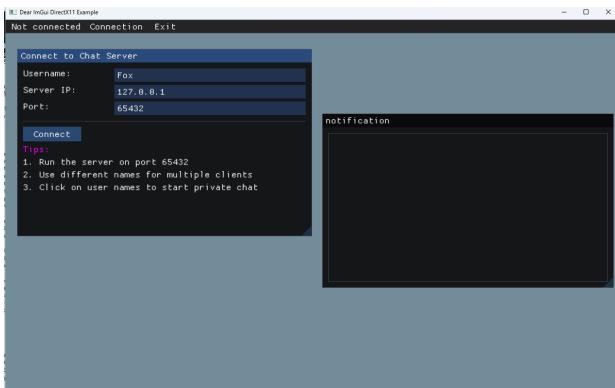


Figure 19: The login in window for connect the server.

```

12     ImGui::Text("Username:");
13     ImGui::SameLine(200);
14     if (ImGui::InputText("#username",
15         username, sizeof(username),
16         ImGuiInputTextFlags_EnterReturnsTrue))
17         connect = true;
18     ImGui::Text("Server IP:");
19     ImGui::SameLine(200);
20     if (ImGui::InputText("#server",
21         serverIP, sizeof(serverIP),
22         ImGuiInputTextFlags_EnterReturnsTrue))
23         connect = true;
24     ImGui::Text("Port:");
25     ImGui::SameLine(200);
26     if (ImGui::InputText("#port", port,
27         sizeof(port),
28         ImGuiInputTextFlags_EnterReturnsTrue))
29         connect = true;
30
31     ImGui::Spacing();
32     ImGui::Separator();
33     ImGui::Spacing();
34
35     if (ImGui::Button("Connect", ImVec2(120,
36         30)) || connect) {
37         int portNum = atoi(port);
38         if (strlen(username) > 0 && portNum
39             > 0 && serverIP != "") {
40             if (connect_server(serverIP,
41                 portNum, username)) {
42                 public_message.push_back(
43                     ChatMessage("System", "Connection failed"));
44             }
45         }
46     }
47 }
```

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

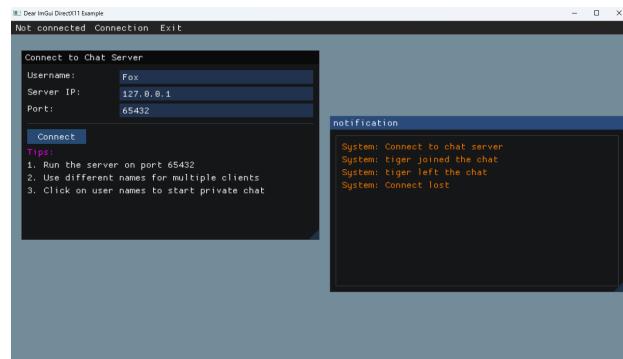


Figure 20: Lost the connection with the server.

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.

```

1 void ChatWindow::user_win() {
2     ImGui::BeginChild("Users", ImVec2(150, 0),
3         true);
4
5     ImGui::TextColored(ImVec4(0, 1, 0, 1), "Online (%d):", (int)users_online.size());
6     ImGui::Separator();
7     for (const auto& user : users_online) {
8         if (user == username) {
9             ImGui::TextColored(ImVec4(0, 1, 0,
10                 1), "> %s (You)", user.c_str());
11         }
12         else {
13             bool hasPrivateChat = (private_chat.
14                 find(user) != private_chat.end());
15
16             if (ImGui::Selectable(user.c_str()))
17             {
18                 // open chat
19                 if (private_chat.find(user) ==
20                     private_chat.end()) {
21                     private_chat[user] = std::vector<ChatMessage>();
22                     private_input[user].fill('\0
23                 }
24             }
25         }
26     }
27 }
```

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

```
1 void ChatWindow::chat_win() {
2     ImGui::BeginChild("Chat messages", ImVec2(0,
3         -ImGui::GetFrameHeightWithSpacing() * 1.5f),
4         true);
5
6     for (const auto& p_message : public_message)
7     {
8         // set user color
9         if (p_message.sender == username) {
10             ImGui::PushStyleColor(ImGuiCol_Text,
11             ImVec4(0.2f, 0.8f, 0.2f, 1.0f));
12         }
13         else if (p_message.sender == "System") {
14             ImGui::PushStyleColor(ImGuiCol_Text,
15             ImVec4(1.0f, 0.5f, 0.0f, 1.0f));
16         }
17         else {
18             ImGui::PushStyleColor(ImGuiCol_Text,
19             ImVec4(0.8f, 0.8f, 0.2f, 1.0f));
20         }
21
22         ImGui::Text("%s:", p_message.sender.
23             c_str());
24         ImGui::PopStyleColor();
25
26         ImGui::SameLine();
27         ImGui::TextWrapped("%s", p_message.text.
28             c_str());
29
30         ImGui::Spacing();
31
32     ImGui::EndChild();
33
34     // input
35     ImGui::Separator();
36
37     ImGui::Text("Message:");
38     ImGui::SameLine();
39
40     ImGui::PushItemWidth(-60);
41     bool sendPublic = false;
42     // press enter
43     if (ImGui::InputText("##PublicInput",
44         public_input, sizeof(public_input),
45         ImGuiInputTextFlags_EnterReturnsTrue)) {
46         sendPublic = true;}
47     ImGui::PopItemWidth();
48
49     ImGui::SameLine();
50     if (ImGui::Button("Send", ImVec2(50, 0)) ||
51         sendPublic) {
52         if (strlen(public_input) > 0) {
53             // send
54             PublicMessage message(username,
55             public_input);
56
57             if (send_message_toserver(
58                 MessageType::PUBLIC_MESSAGE, &message,
59                 sizeof(message))) {
60                 // show at local
61                 ChatMessage mess(username,
62                 public_input);
63                 public_message.push_back(mess);
64             }
65         }
66         // clear input
67         public_input[0] = '\0'; } } }
```

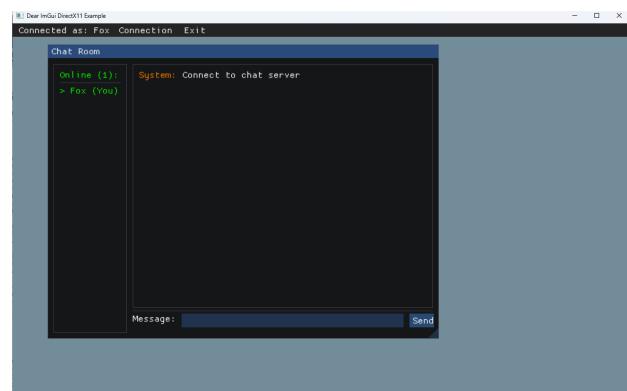


Figure 21: The public chat room.

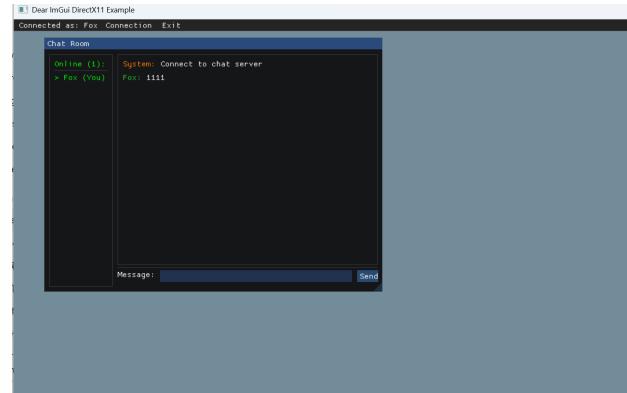


Figure 22: After send a public message. The different color of different user.

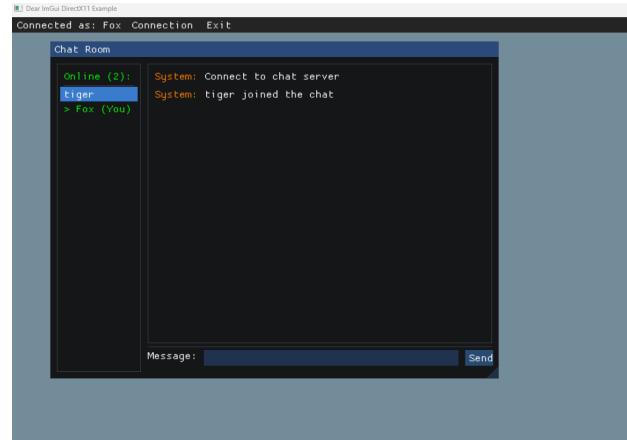


Figure 23: New client join in. The user list changed

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-

Figure 25.

```
1 void ChatWindow::private_win() {
2     for (auto it = private_chat.begin(); it != private_chat.end(); ) {
3         const std::string targetUser = it->first;
4         std::vector<ChatMessage>& messages = it->second;
5
6         std::string title = "Private: " + targetUser;
7         bool isopen = true;
8
9         ImGui::SetNextWindowSize(ImVec2(400, 300), ImGuiCond_FirstUseEver);
10        // imgui can remember last window's detail by title
11        if (ImGui::Begin(title.c_str(), &isopen)) {
12
13            ImGui::BeginChild("PrivateMessages", ImVec2(0, -ImGui::GetFrameHeightWithSpacing() * 1.5f), true);
14
15            for (const auto& mes : messages) {
16                // color
17                if (mes.sender == username) {
18                    ImGui::PushStyleColor(ImGuiCol_Text, ImVec4(0.5f, 0.1f, 0.5f, 1.0f));
19                }
20                else {
21                    ImGui::PushStyleColor(ImGuiCol_Text, ImVec4(0.1f, 0.5f, 0.1f, 1.0f));
22                }
23                ImGui::Text("%s:", mes.sender.c_str());
24                ImGui::PopStyleColor();
25
26                ImGui::SameLine();
27                ImGui::TextWrapped("%s", mes.text.c_str());
28
29                ImGui::Spacing();
30            }
31
32            ImGui::EndChild();
33
34            // input
35            ImGui::Separator();
36            ImGui::Text("To %s:", targetUser.c_str());
37            ImGui::SameLine();
38
39            auto& input_buff = private_input[targetUser];
40
41            ImGui::PushItemWidth(-60);
42            bool send = false;
43            // press enter
44            if (ImGui::InputText("##PrivateInput", input_buff.data(), input_buff.size(), ImGuiInputTextFlags_EnterReturnsTrue)) {
45                send = true;
46            }
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
```

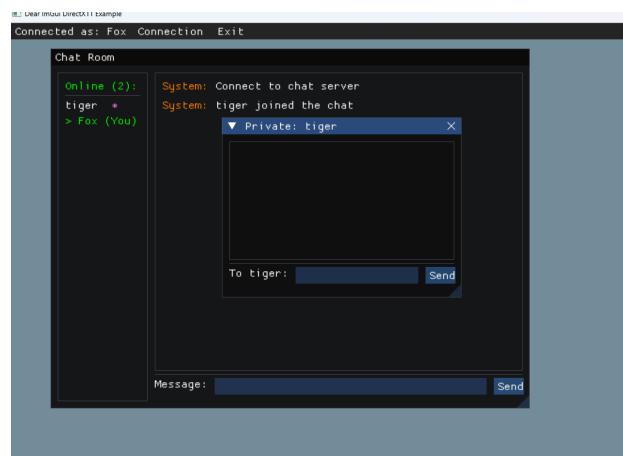


Figure 24: New client join in. The user list changed

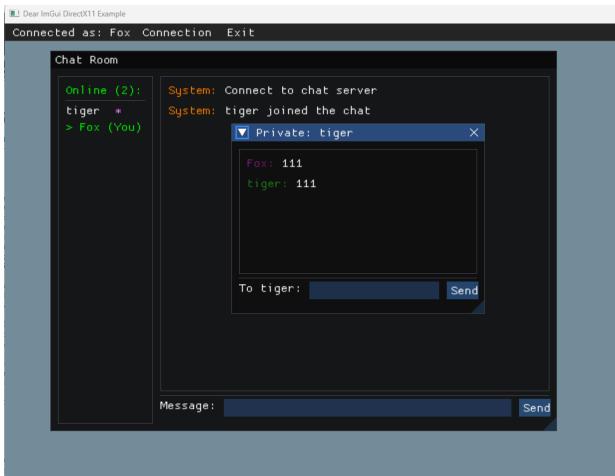


Figure 25: Shows the private Chat.

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