

```
1 #include <fstream>
2 #include <string>
3 #include <vector>
4 #include <iostream>
5
6 #include <GL/glew.h>
7
8 #include <SDL2/SDL_main.h>
9 #include <SDL2/SDL.h>
10 #include <SDL2/SDL_opengl.h>
11 #undef main
12
13 #define GLM_FORCE_RADIANS
14 #include <glm/glm.hpp>
15 #include <glm/gtc/matrix_transform.hpp>
16 #include <glm/gtc/type_ptr.hpp>
17
18 #define STB_IMAGE_IMPLEMENTATION
19 #include "STB/stb_image.h"
20
21 #include "myShader.h"
22
23 using namespace std;
24
25 // SDL variables
26 SDL_Window* window;
27 SDL_GLContext glContext;
28
29 bool quit = false;
30
31 int mouse_position[2];
32 bool button_pressed = false;
33
34 int window_height = 863;
35 int window_width = 1646;
36
37 float fovy = 45.0f;
38 float znear = 1.0f;
39 float zfar = 2000.0f;
40
41 glm::vec3 camera_eye = glm::vec3(0.0f, 0.0f, 2.0f);
42 glm::vec3 camera_up = glm::vec3(0.0f, 1.0f, 0.0f);
43 glm::vec3 camera_forward = glm::vec3(0.0f, 0.0f, -1.0f);
44
45
46 void rotate(glm::vec3 & inputvec, glm::vec3 rotation_axis, float theta, bool tonormalize = false)
47 {
48     const float cos_theta = cos(theta);
```

```
49     const float dot = glm::dot(inputvec, rotation_axis);
50     glm::vec3 cross = glm::cross(inputvec, rotation_axis);
51
52     inputvec.x *= cos_theta; inputvec.y *= cos_theta; inputvec.z *= cos_theta;
53     inputvec.x += rotation_axis.x * dot * (float)(1.0 - cos_theta);
54     inputvec.y += rotation_axis.y * dot * (float)(1.0 - cos_theta);
55     inputvec.z += rotation_axis.z * dot * (float)(1.0 - cos_theta);
56
57     inputvec.x -= cross.x * sin(theta);
58     inputvec.y -= cross.y * sin(theta);
59     inputvec.z -= cross.z * sin(theta);
60
61     if (tonormalize) inputvec = glm::normalize(inputvec);
62 }
63
64 // Process the event.
65 void processEvents(SDL_Event current_event)
66 {
67     switch (current_event.type)
68     {
69         // window close button is pressed
70     case SDL_QUIT:
71     {
72         quit = true;
73         break;
74     }
75     case SDL_KEYDOWN:
76     {
77         if (current_event.key.keysym.sym == SDLK_ESCAPE)
78             quit = true;
79     }
80     case SDL_MOUSEBUTTONDOWN:
81     {
82         mouse_position[0] = current_event.button.x;
83         mouse_position[1] = window_height - current_event.button.y;
84         button_pressed = true;
85         break;
86     }
87     case SDL_MOUSEBUTTONUP:
88     {
89         button_pressed = false;
90         break;
91     }
92     case SDL_MOUSEMOTION:
93     {
94         if (button_pressed == false) break;
95
96         int x = current_event.motion.x;
97         int y = window_height - current_event.motion.y;
```

```
98
99     int dx = x - mouse_position[0];
100     int dy = y - mouse_position[1];
101
102     if (dx == 0 && dy == 0) break;
103
104     mouse_position[0] = x;
105     mouse_position[1] = y;
106
107     float vx = (float)dx / (float>window_width;
108     float vy = (float)dy / (float>window_height;
109     float theta = 4.0f * (fabs(vx) + fabs(vy));
110
111     glm::vec3 camera_right = glm::normalize(glm::cross(camera_forward, ↗
        camera_up));
112
113     glm::vec3 tomovein_direction = -camera_right * vx + -camera_up * vy;
114
115     glm::vec3 rotation_axis = glm::normalize(glm::cross(tomovein_direction, ↗
        camera_forward));
116
117     rotate(camera_forward, rotation_axis, theta, true);
118     rotate(camera_up, rotation_axis, theta, true);
119     rotate(camera_eye, rotation_axis, theta, false);
120
121     break;
122 }
123 case SDL_MOUSEWHEEL:
124 {
125     if (current_event.wheel.y < 0)
126         camera_eye -= 0.1f * camera_forward;
127     else if (current_event.wheel.y > 0)
128         camera_eye += 0.1f * camera_forward;
129 }
130 default:
131     break;
132 }
133 }
134
135 int main(int argc, char *argv[])
136 {
137     // Initialize video subsystem
138     SDL_Init(SDL_INIT_TIMER | SDL_INIT_VIDEO);
139
140     // Using OpenGL 3.1 core
141     SDL_GL_SetAttribute(SDL_GL_CONTEXT_MAJOR_VERSION, 3);
142     SDL_GL_SetAttribute(SDL_GL_CONTEXT_MINOR_VERSION, 1);
143     SDL_GL_SetAttribute(SDL_GL_CONTEXT_PROFILE_MASK, ↗
        SDL_GL_CONTEXT_PROFILE_CORE);
```

```
144     SDL_GL_SetAttribute(SDL_GL_DOUBLEBUFFER, 1);
145     SDL_GL_SetAttribute(SDL_GL_MULTISAMPLEBUFFERS, 1);
146     SDL_GL_SetAttribute(SDL_GL_MULTISAMPLES, 4);
147     SDL_GL_SetAttribute(SDL_GL_ACCELERATED_VISUAL, 1);
148
149     // Create window
150     window = SDL_CreateWindow("IT-5102E", SDL_WINDOWPOS_CENTERED,           ↗
        SDL_WINDOWPOS_CENTERED,
151         window_width, window_height, SDL_WINDOW_OPENGL | SDL_WINDOW_RESIZABLE);
152
153     // Create OpenGL context
154     glContext = SDL_GL_CreateContext(window);
155
156     // Initialize glew
157     glewInit();
158     glEnable(GL_DEPTH_TEST);
159     glDepthFunc(GL_LESS);
160     glEnable(GL_TEXTURE_2D);
161     glDisable(GL_CULL_FACE);
162
163     glClearColor(0.1f, 0.1f, 0.1f, 1.0f);
164
165     myShader *shader = new myShader("textureimage-vertexshader.glsl",       ↗
        "textureimage-fragmentshader.glsl");
166     shader->start();
167
168
169     vector<glm::vec3> vertices;
170     vertices.push_back(glm::vec3(0.0f, 0.0f, 0.0f));
171     vertices.push_back(glm::vec3(1.0f, 0.0f, 0.0f));
172     vertices.push_back(glm::vec3(0.0f, 1.0f, 0.0f));
173     vertices.push_back(glm::vec3(0.0f, 0.0f, 1.0f));
174
175     vector<glm::ivec3> indices;
176     indices.push_back(glm::ivec3(1, 2, 3));
177     indices.push_back(glm::ivec3(0, 1, 2));
178     indices.push_back(glm::ivec3(2, 0, 3));
179     indices.push_back(glm::ivec3(1, 0, 3));
180
181     vector<glm::vec3> normals;
182     normals.push_back(glm::vec3(-1, -1, -1));
183     normals.push_back(glm::vec3(1, 0, 0));
184     normals.push_back(glm::vec3(0, 1, 0));
185     normals.push_back(glm::vec3(0, 0, 1));
186
187     /*****/
188     vector<glm::vec2> texturecoordinates;
189     texturecoordinates.push_back(glm::vec2(0,0));
190     texturecoordinates.push_back(glm::vec2(0, 1));
```

```

191 texturecoordinates.push_back(glm::vec2(1, 0));
192 texturecoordinates.push_back(glm::vec2(1, 1));
193 /*****
194
195 GLuint vao;
196 glGenVertexArrays(1, &vao);
197 glBindVertexArray(vao);
198
199 GLuint buffers[4];
200 glGenBuffers(4, buffers);
201
202 unsigned int location;
203
204 location = 0;
205 glBindBuffer(GL_ARRAY_BUFFER, buffers[0]);
206 glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(glm::vec3),
207              &vertices[0], GL_STATIC_DRAW);
208 glVertexAttribPointer(location, 3, GL_FLOAT, GL_FALSE, 0, 0);
209 glEnableVertexAttribArray(location);
210
211 location = 1;
212 glBindBuffer(GL_ARRAY_BUFFER, buffers[1]);
213 glBufferData(GL_ARRAY_BUFFER, normals.size() * sizeof(glm::vec3), &normals
214              [0], GL_STATIC_DRAW);
215 glEnableVertexAttribArray(location);
216 glVertexAttribPointer(location, 3, GL_FLOAT, GL_FALSE, 0, 0);
217
218 /*****
219 location = 2;
220 glBindBuffer(GL_ARRAY_BUFFER, buffers[3]);
221 glBufferData(GL_ARRAY_BUFFER, texturecoordinates.size() * sizeof
222              (glm::vec2), &texturecoordinates[0], GL_STATIC_DRAW);
223 glEnableVertexAttribArray(location);
224 glVertexAttribPointer(location, 2, GL_FLOAT, GL_FALSE, 0, 0);
225 /*****
226
227 glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, buffers[2]);
228 glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(glm::ivec3),
229              &indices[0], GL_STATIC_DRAW);
230
231 glBindVertexArray(0);
232
233 /
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999

```

```

234
235     GLubyte *mytexture = stbi_load("scenary.jpg", &width, &height, &size, 4);
236
237     glGenTextures(1, &texture_id);
238     glBindTexture(GL_TEXTURE_2D, texture_id);
239
240     glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
241     glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
242     glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
243     glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
244
245     glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, static_cast<GLuint>(width),
                static_cast<GLuint>(height), 0, GL_RGBA, GL_UNSIGNED_BYTE,
                mytexture);
246
247     delete mytexture;
248     glGenerateMipmap(GL_TEXTURE_2D);
249     glBindTexture(GL_TEXTURE_2D, 0);
250 }
251 /
                *****
                */
252
253
254 // display loop
255 while (!quit)
256 {
257     glViewport(0, 0, window_width, window_height);
258
259     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
260
261     glm::mat4 projection_matrix = glm::perspective(glm::radians(fovy),
                static_cast<float>(window_width) / static_cast<float>(window_height),
                znear, zfar);
262     glUniformMatrix4fv(glGetUniformLocation(shader->shaderprogram,
                "myprojection_matrix"), 1, GL_FALSE, glm::value_ptr
                (projection_matrix));
263
264     glm::mat4 view_matrix = glm::lookAt(camera_eye, camera_eye +
                camera_forward, camera_up);
265     glUniformMatrix4fv(glGetUniformLocation(shader->shaderprogram,
                "myview_matrix"), 1, GL_FALSE, glm::value_ptr(view_matrix));
266
267
268     glUniform4fv(glGetUniformLocation(shader->shaderprogram,
                "input_color"), 1, glm::value_ptr(glm::vec4(1.0f, 0.0f, 0.0f,
                1.0f)));

```

```

269
270      /
          *****
          *****/
271      int texture_offset = 8;
272      glActiveTexture(GL_TEXTURE0 + texture_offset);
273      glBindTexture(GL_TEXTURE_2D, texture_id);
274
275      glUniform1i(glGetUniformLocation(shader->shaderprogram, "imagetex"),
          texture_offset);
276
277      shader->setUniform("imagetex", static_cast<int>(texture_offset));
278      /
          *****
          *****/
279
280      glBindVertexArray(vao);
281      glDrawElements(GL_TRIANGLES, static_cast<GLsizei>(indices.size() * 3),
          GL_UNSIGNED_INT, 0);
282      glBindVertexArray(0);
283
284
285      glUniform4fv(glGetUniformLocation(shader->shaderprogram,
          "input_color"), 1, glm::value_ptr(glm::vec4(1.0f, 1.0f, 1.0f,
          1.0f)));
286      glBegin(GL_LINES);
287      for (unsigned int i = 0; i < vertices.size(); ++i)
288      {
289          glm::vec3 v = vertices[i] + glm::normalize(normals[i]);
290          glVertex3fv(&vertices[i][0]);
291          glVertex3fv(&v[0]);
292      }
293      glEnd();
294
295
296      SDL_GL_SwapWindow(window);
297
298      SDL_Event current_event;
299      while (SDL_PollEvent(&current_event) != 0)
300          processEvents(current_event);
301  }
302
303  // Freeing resources before exiting.
304
305  // Destroy window
306  if (glContext) SDL_GL_DeleteContext(glContext);
307  if (window) SDL_DestroyWindow(window);
308
309  // Quit SDL subsystems

```

```
310     SDL_Quit();
311
312     return 0;
313 }
```