



SET08116 Computer Graphics

Workbook 2016/17

Dr Kevin Chalmers
Sam Serrels

Edinburgh Napier University

February 21, 2017

Version 2.0

Copyright © 2017 Edinburgh Napier University

This work has been produced for delivery of a Computer Graphics course to be taught at Edinburgh Napier University or its associated partners. All images are used under fair use and are for educational purposes only. No commercial usage is permitted.

GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright (C) 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc. <http://fsf.org/>

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Contents

	Page
I Getting Started	18
1 Getting Started	19
1.1 Getting the content	19
1.1.1 Pulling from Git	19
1.1.2 Creating the Solution, with CMake	19
1.1.3 Workbook Style	20
1.2 Graphics Framework	20
1.2.1 Libraries	20
1.2.2 <code>app</code> Class	21
1.2.3 <code>renderer</code> Class	22
1.2.4 Basic Graphics Application	22
1.3 What next?	22
2 Rendering a Triangle	24
2.1 Defining a Triangle	24
2.2 <code>geometry</code> Class	25
2.2.1 Creating a Buffer	25
2.2.2 Adding to Geometry Object	26
2.2.3 Try it - Set the Buffer Data	26
2.3 Exercise	26
3 Rendering a Quad with Triangles	28
3.1 Code	28
4 Rendering a Quad with Quads	30
5 Rendering a Quad with Triangle Strips	31
6 Rendering a Quad with Lines	32
7 Rendering a Quad with Line Strips	33
8 Rendering a Quad with Line Loops	34
9 Rendering a Quad with Triangle Fans	35
10 Transformations	36
10.1 Vectors	36
10.1.1 Vector Addition	36
10.1.2 Scaling Vectors	36
10.1.3 Length of a Vector	37

10.1.4 Normalising a Vector	37
10.1.5 Dot Product	37
10.1.6 Cross Product	38
10.1.7 Exercises	40
10.2 Matrices	42
10.2.1 Addition	42
10.2.2 Scaling	42
10.2.3 Multiplying	42
10.2.4 Exercises	44
10.3 Transformation Matrices	45
10.3.1 Translation	45
10.3.2 Rotation	46
10.3.3 Scale	48
10.3.4 Combining Transformations	48
10.3.5 Exercises	50
10.4 Answers	51
10.4.1 Vectors	51
10.4.2 Matrices	53
10.4.3 Transformation Matrices	54
11 Quaternions	56
11.1 Why use Quaternions?	56
11.2 What are Quaternions?	56
11.3 Using Quaternions to Represent Rotations	57
11.4 Combining Rotations	58
11.4.1 Quaternion Multiplication	58
11.5 Converting a Quaternion to a Rotation Matrix	59
11.6 GLM to the Rescue	59
12 Working with GLM	60
12.1 Vectors	60
12.1.1 Defining Vectors	60
12.1.2 Vector Addition and Subtractions	61
12.1.3 Vector Scaling	61
12.1.4 Length of a Vector	62
12.1.5 Normalizing a Vector	62
12.1.6 Dot Product	62
12.1.7 Cross Product	63
12.2 Matrices	63
12.2.1 Defining a Matrix	63
12.2.2 Matrix Addition	63
12.2.3 Matrix Scaling	64
12.2.4 Matrix Multiplication	64
12.3 Transformations	64
12.3.1 Translation Matrix	65
12.3.2 Rotation Matrix	65
12.3.3 Scale Matrix	66
12.3.4 Combining Matrices	66
12.4 Quaternions	66
12.4.1 Defining Quaternions	66
12.4.2 Quaternions for Rotations	66
12.4.3 Quaternion Multiplication	67
12.4.4 Conversion to a Matrix	67

13 Rotating a Triangle	68
13.1 Updating Lesson	68
13.2 Exercise	68
14 Scaling a Triangle	70
14.1 Exercise	70
15 Scaling and Rotating a Triangle	71
16 Moving Quad	72
16.1 Exercise	73
17 Full Transformation	74
18 Point Based Sierpinski Gasket	75
18.1 What is the Sierpinski Gasket?	75
18.2 Getting Started	75
18.2.1 Random Number Generation in C++	76
18.3 Generating Points	76
18.3.1 Challenge	76
18.4 Exercises	77
II Working with Meshes	80
19 Rendering a Cube	81
19.1 Updating the Project	81
19.2 Exercise	81
20 Transforming a Cube	83
21 Working with Indices	84
21.1 What is Indexing?	84
22 Indexed Cube	87
23 Sierpinski Gasket	88
23.1 Getting Started	88
23.2 Challenge Part 1	88
23.3 Challenge Part 2	89
23.4 Challenge Part 3	90
24 Sphere by Subdivision	93
24.1 Technique	93
24.1.1 Getting Started	94
24.1.2 Implementation	94
25 Meshes	97
25.1 What is a Mesh?	97
25.2 Creating a Mesh	98
25.3 Rendering a Mesh	99
26 Transforming a Mesh	100
26.1 <code>transform</code>	100
26.2 Using the Transformation	101
26.3 Getting the Transform Matrix	102

III Shaders and Texturing Effects	103
27 Shaders	104
27.1 What is a Shader?	104
27.2 How do Shaders Work?	105
27.2.1 Why Shaders are Fast	105
27.3 Shader Types	105
27.3.1 Vertex Shader	105
27.3.2 Geometry Shader	106
27.3.3 Fragment (or Pixel) Shader	106
27.4 GLSL	106
27.5 Using Shaders	107
27.5.1 Compiling Shaders	107
27.5.2 Compilation Problems	108
27.5.3 Linking Programs	108
27.5.4 Linking Problems	109
27.6 Shaders in the Graphics Framework	109
27.6.1 Creating Effects	109
27.6.2 Loading Shaders	110
27.6.3 Building Effects	110
27.6.4 Binding Effects	110
27.7 Further Reading	111
28 Colour Shader	112
28.1 A Closer Look at Shaders	112
28.1.1 How Shaders Run on the GPU	112
28.1.2 Previous Vertex Shader Example	112
28.1.3 Fragment Shader Example	114
28.2 Coloured Shader	115
28.2.1 Passing Values to Shaders	115
28.2.2 Setting the Value in our Main Application	116
28.3 Exercises	117
29 Texturing	118
29.1 What is Texturing?	118
29.2 How do we Texture?	118
30 Texturing in Shaders	120
30.1 Texturing in GLSL	120
30.2 Using Textures in our Render Framework	121
30.2.1 Setting Texture Coordinates	121
30.2.2 Loading Textures	121
30.2.3 Setting Textures in Shaders	122
30.3 Exercises	122
31 Mipmaps	124
31.1 Minification and Magnification	124
31.2 Mipmap Test Application	125
32 Anisotropic Filtering	128
33 Multi-texturing	130
33.1 Blend Maps	130
34 Blended Textures	133

34.1 Blend Shader	133
34.2 Excercise	134
35 Dissolve Shader	136
35.1 Discarding Fragments	136
35.2 Shader Structure	136
35.2.1 Fragment Shader	136
36 Simple Cell Shading	139
36.1 1D Textures	139
36.2 Cell Shader	139
36.3 Creating Textures from Colour Data	139
36.4 Exercises	140
37 Repeat Textures	142
37.1 Non-uniform Texture Coordinates	142
37.1.1 Defining Non-uniform Texture Coordinates	143
37.2 Updating Main	144
38 Working with CodeXL	145
38.1 Starting CodeXL	145
38.2 Analysing Buffers	145
38.3 Shaders Viewer	146
38.4 Stepping through a Program	146
38.5 Excercises	147
IV Models Meshes and Transforms	148
39 Geometry Builder	149
39.1 Box	149
39.2 Tetrahedron	150
39.3 Pyramid	150
39.4 Disk	150
39.5 Cylinder	151
39.6 Sphere	151
39.7 Torus	151
39.8 Plane	152
39.9 Exercise	152
40 Transform Hierarchy	155
40.1 Transform refresher	155
40.2 Chaining transforms	155
41 Loading Models for Geometry	157
41.1 Loading a Model	157
41.2 Updating Lesson	157
41.3 Exercise	157
V Cameras	159
42 Cameras	160
42.1 What do we mean by Cameras?	160
42.1.1 Why do we need Cameras?	161

42.2 From the Eye to a Camera	161
42.2.1 The Eye	161
42.2.2 Pinhole Camera	161
42.2.3 Synthetic Camera Model	162
42.3 Base Camera Class	163
43 Coordinate Spaces	166
43.1 Linear Transformations	166
43.2 Transformation Matrix	167
43.3 Coordinate Spaces of Interest	167
43.4 Transforming Between Spaces	168
43.5 Recommended Reading	169
44 View Transformation Matrix	170
44.1 Defining a View	170
44.2 Constructing a View Matrix	170
44.3 Transforming Objects - World Space to Camera Space	172
44.3.1 Performing a Camera Transformation	172
44.4 Exercises	173
44.5 Answers	174
45 Projection Transformation Matrix	176
45.1 Defining a Projection	176
45.2 Constructing a Projection Matrix	176
45.3 Transforming Objects - Camera Space to Screen Space	178
45.3.1 Homogeneous Coordinates	179
45.4 Exercises	180
45.5 Answers	181
46 Target Camera	183
46.1 What is a Target Camera?	183
46.2 Target Camera Properties	183
46.3 Updating the Target Camera	183
46.4 Exercise	184
47 Free Camera	186
47.1 What is a Free Camera?	186
47.1.1 Restricted Free Camera	186
47.2 Free Camera Properties	186
47.3 Controlling the Free Camera	187
47.4 Updating the Free Camera	187
47.5 Exercise	188
47.5.1 Working with the Mouse	188
47.5.2 Update	189
48 Chase Camera	191
48.1 What is a Chase Camera?	191
48.2 Chase Camera Properties	191
48.3 Controlling the Chase Camera	191
48.4 Updating the Chase Camera	192
48.5 Exercise	193
49 Arc-ball Camera	195
49.1 What is an Arc-ball Camera?	195
49.2 Arc-ball Camera Properties	195

49.3 Controlling the Arc-ball Camera	195
49.4 Updating the Arc-ball Camera	196
49.5 Exercise	196
50 Picking	199
50.1 What is Picking?	199
50.2 Picking with Ray-casting	199
50.3 Converting a Screen Position to a World Ray	199
50.3.1 Back to Coordinate Spaces	200
50.3.2 Inverse Transforms	201
50.3.3 World Ray Calculation	201
50.3.4 Screen Position to World Ray Algorithm	201
50.4 Defining an AABB	202
50.5 Converting an AABB to an OOOB	202
50.6 Determining Ray and OOOB Intersection	202
50.7 Exercise	203
VI Lighting	206
51 Lighting	207
51.1 What is Lighting?	207
51.2 Light Source	207
51.3 Materials	208
51.3.1 Scattering	208
51.4 Sensors	208
51.5 Recommended Reading	209
52 Simple Ambient Lighting	210
52.1 Examples	210
52.2 Ambient Light Equation	210
52.3 Ambient Lighting Shader	211
52.4 Exercise	211
53 Diffuse Lighting	213
53.1 Examples	213
53.2 Diffuse Lighting Equation	214
54 Surface Normals	216
54.1 What is a Surface Normal?	216
54.2 Calculating Surface Normals	216
55 Diffuse Lighting Attempt 1	218
56 Diffuse Lighting Attempt 2	220
56.1 Working Diffuse Shader	220
57 Specular Lighting	222
57.1 Specular Lighting Equation	222
57.2 Specular Shader	224
57.3 Exercise	225
58 Combined Lighting	228
59 Emissive Lighting	230
59.1 What is Emissive Light?	230

60 Gouraud Shading	232
60.1 structs in Shaders	232
60.1.1 Declaring structs in Shaders	233
60.1.2 Setting Uniform Values in structs	234
60.2 Gouraud Shader	236
60.3 Exercise	236
61 Phong Shading	239
61.1 Phong Shader	240
62 Point Lights	242
62.1 What is a Point Light?	242
62.2 Standard Directional Light	242
62.3 Working with Distance	243
62.3.1 Attempt 1	243
62.3.2 Attempt 2	244
62.3.3 Attempt 3	245
62.3.4 Point Light Equation	245
62.4 Point Light Data in the Render Framework	246
62.5 Exercises	247
63 Spot Lights	249
63.1 What is a Spot Light?	249
63.2 Standard Point Light	249
63.3 Spot Light Equation	249
63.4 Spot Light Data in the Render Framework	250
64 Multiple Lights	253
64.1 Array Uniforms	253
64.1.1 Array Uniform Naming	254
64.2 Functions in Shaders	255
64.2.1 Looping in Shaders	255
64.3 Binding Vectors of Lights	256
64.4 Exercise	256
VII Lighting Effects	257
65 Multi-file Shaders	258
65.1 What is a GLSL Program?	258
65.2 Loading Multiple Shaders in the Graphics Framework	258
65.3 Ensuring we Don't Have Multiple Definitions	259
65.4 Example - Directional Light Shader	259
65.5 Defining that a Function Exists	260
65.6 Exercise	260
66 Shadow Mapping	262
66.1 The Depth Buffer	262
66.2 Creating a <code>shadow_map</code> in the Graphics Framework	262
66.2.1 <code>depth_buffer</code>	262
66.2.2 <code>shadow_map</code>	263
66.3 Rendering to a Depth Buffer	263
66.4 Completing the Lesson	264
67 Shadowing	266

67.1 The <code>shadow.frag</code> Shader	266
67.1.1 Exercise	267
67.2 Calculating the Light Space Coordinate	267
67.2.1 Exercise	267
67.2.2 Exercise 2 - Complete <code>shader.frag</code>	267
67.3 Performing the Render	268
67.4 Exercise	268
68 Normal Mapping	269
68.1 Shaders	270
68.2 Completing the Lesson	271
69 Cube Maps	273
69.1 What is a Cubemap?	273
69.2 3D Texture Coordinates	273
69.3 Loading a Cubemap in the Graphics Framework	274
69.4 Completing the Lesson	275
70 Skybox	276
70.1 Skybox Geometry	276
70.1.1 Exercise - Create Skybox Geometry	276
70.2 Rendering a Skybox	276
70.3 Skybox Shader	277
70.4 Completing the Lesson	277
70.5 Exercise	277
71 Environment Maps	279
72 Tarnished Object	281
73 Terrain	282
73.1 Generating Terrain Data	282
73.2 Part 1 - Generating Geometry	282
73.2.1 Getting Texture Data	283
73.2.2 Generate Position Geometry	284
73.2.3 Generate Index Data	285
73.2.4 Completing Part 1	285
73.3 Part 2 - Generating Normals	285
73.3.1 Completing Part 2	286
73.4 Part 3 - Generating Texture Coordinates	286
73.5 Part 4 - Generating Texture Weights	288
73.5.1 Terrain Shader	288
73.5.2 Generating Texture Weights	289
73.5.3 Completing the Lesson	290
74 Fog	292
74.1 Getting Started	292
74.2 Linear Fog	293
74.3 Exponential Fog	293
74.4 Exponential Squared Fog	294
75 BRDF	297

VIII Geometry Shader	298
76 Geometry Shader	299
76.1 Available Inputs to the Geometry Shader	299
76.1.1 Input Types	299
76.1.2 Available Variables	300
76.1.3 Defining Incoming Values	300
76.1.4 Incoming Values from Previous Stage (Vertex Shader)	301
76.2 Available Outputs from the Geometry Shader	301
76.2.1 Output Types	301
76.2.2 Output Variables	301
76.2.3 Defining Outgoing Values	302
76.2.4 Outgoing Values to the Next Stage (Fragment Shader)	302
76.2.5 Outputting Vertex Data	302
76.3 Example Geometry Shader	303
76.4 Loading a Geometry Shader	303
76.5 Example Shader - Copy Geometry	303
77 Exploding Shape	306
78 Showing Normals for Debugging	308
78.1 Normal Rendering Geometry Shader	308
78.2 Completing the Lesson	309
79 Billboardng	310
79.1 What is Billboardng?	310
79.2 How Billboardng Works with the Geometry Shader	310
79.3 Billboardng Shader Structure	311
79.4 Completing the Lesson	311
80 Particle Effects	314
80.1 Transform Feedback	314
80.2 The Shader	315
80.3 Particles	315
80.4 Creating Transform Buffers	316
80.4.1 Allocating the Buffers with OpenGL	316
80.4.2 Setting the Buffer Data	316
80.4.3 Describing the Buffer	317
80.5 Performing the Update	317
80.5.1 Disabling Rendering	318
80.5.2 Setting up the Data Streams	318
80.5.3 Performing the Feedback	319
80.5.4 Ending the Update	319
80.6 Rendering the Transform Feedback with OpenGL	319
80.7 Completing the Lesson	320
81 Smoke	321
IX Post-processing	322
82 Post-processing	323
83 Frame Buffers	324
83.1 One Last Look at the Pipeline	324

83.2 Creating a <code>frame_buffer</code>	325
83.3 Working with Frame Buffers	325
83.4 Completing the Lesson	325
84 Displaying to Screen	327
84.1 Things you Should Remember - Screen Dimensions	327
85 Greyscale Post-process	329
85.1 Exercises	329
86 Blur Post-process	331
86.1 Exercises	332
87 Motion Blur Post-process	334
88 Depth-of-Field	336
88.1 Exercise	336
89 Masking Post-process	338
90 Ambient Occlusion	339
91 Deferred Shading	340
GNU Free Documentation License	341
1. APPLICABILITY AND DEFINITIONS	341
2. VERBATIM COPYING	342
3. COPYING IN QUANTITY	342
4. MODIFICATIONS	342
5. COMBINING DOCUMENTS	343
6. COLLECTIONS OF DOCUMENTS	343
7. AGGREGATION WITH INDEPENDENT WORKS	343
8. TRANSLATION	343
9. TERMINATION	343
10. FUTURE REVISIONS OF THIS LICENSE	343
11. RELICENSING	343

List of Figures

1.1 Output from Basic Application	23	28.2 Output from Colour Shader Application	117
2.1 Output from Triangle Application	27	29.1 Textured Object	119
3.1 A Quad Made of Two Triangles	28	29.2 Texture Coordinates	119
3.2 A Quad using Triangles	29	30.1 Simple Texture Shader	121
5.1 Triangle Strip	31	30.2 Textured Object	123
6.1 Lines Output	32	31.1 Magnification	124
9.1 Triangle Fan	35	31.2 Minification	125
10.1 Two Vectors forming a Triangle	39	31.3 Mipmaps	126
10.2 Matrix Multiplication	43	31.4 Mipmap Test Application Output .	127
13.1 Rotating Triangle Output	69	32.1 Anisotropic Test Application Output	128
14.1 Scaling Triangle	70	33.1 Multi-texturing	131
15.1 Scaling and Rotating Triangle	71	33.2 Blend Map	132
16.1 Output from Moving Quad Lesson	72	34.1 Blend Shader	134
17.1 Full Transformation on a Quad	74	34.2 Blended Textures on a Cube	135
18.1 Sierpinski Gaskets at Different Division Levels	78	35.1 Dissolve Shader Structure	137
18.2 Output from Point Based Sierpinski Gasket	79	35.2 Dissolve Shader Output	138
19.1 Cube Render Output	82	36.1 Simple Cell Shader	140
20.1 Transformed Cube Output	83	36.2 Cell Shading with a 1D Texture . .	141
21.1 Quad Built Using Triangles	85	37.1 Non-uniform Texture Coordinates	142
21.2 Indexed Quad	86	37.2 Repeating Texture Coordinates . .	143
22.1 Rendered Cube using Index Data	87	37.3 Output from Texture Repeat Lesson	144
23.1 Sierpinski at 0 Divisions	89	38.1 CodeXL Menu in Visual Studio . .	146
23.2 Sierpinski at 1 Division	90	38.2 CodeXL Explorer	147
23.3 Sierpinski at 2 Divisions	92	39.1 Output from <code>geometry_builder</code> Application	154
23.4 Sierpinski Gasket Output	92	40.1 Output of Practical	156
24.1 Circle by Subdivision	93	41.1 Output from Model Loading Lesson	158
24.2 Sphere by Subdivision Output	96	42.1 View Transformation	160
25.1 Mesh Class Structure	98	42.2 Screen Transformation	161
27.1 Graphics Pipeline	104	42.3 The Eye	162
27.2 Console Window with Shader Error	111	42.4 Pinhole Camera	163
28.1 Colour Shader	115	42.5 ynthetic Camera Model	164
		46.1 Converting to Actual Up	184
		46.2 Output from Target Camera Lesson	185
		47.1 Output from Free Camera Lesson .	190
		48.1 Chase Camera	192
		48.2 Output from Chase Camera Lesson	194
		49.1 Arc-ball Camera	197
		49.2 Output from Arc-ball Camera Lesson	198

50.1 Raycasting in the View Frustum	200	66.1 CodeXL Depth Buffer View	263
50.2 Ray Intersecting with Minimal and Maximal Planes	203	66.2 Captured Depth Buffer	265
50.3 Output from Picking Lesson	205	67.1 Shadow Mapping Shader	266
51.1 Reflection and Refraction	209	67.2 Output from Shadow Mapping Lesson	268
52.1 Ambient Shader Structure	212	68.1 A Normal Map	270
52.2 Output from Ambient Lighting Lesson	212	68.2 Tangent Space	271
53.1 Ambient Diffuse Comparison	213	68.3 Normal Map Shader Structure	272
54.1 Surface Normal	217	68.4 Normal Map Output	272
55.1 Simple Diffuse Shader Structure	218	69.1 An Example Cubemap	274
55.2 Output from First Attempt at a Diffuse Shader	219	69.2 CodeXL Cubemap View	275
56.1 Working Diffuse Shader Structure	220	70.1 Output from Skybox Lesson	278
56.2 Output from Working Diffuse Shader	221	71.1 Environment Map Shader Structure	279
57.1 Specular Highlights	223	71.2 Output from Environment Map Lesson	280
57.2 Eye-Light Interaction	224	72.1 Output from Tarnished Object Lesson	281
57.3 Specular Shader Structure	225	73.1 Heightmap Example	283
57.4 Specular Shader Structure	226	73.2 Terrain Mesh	284
57.5 Output from Specular Shader	227	73.3 Output form Part 1 of Terrain Rendering	286
58.1 Structure of Combined Lighting Shader	229	73.4 Correctly Lit Terrain	288
58.2 Output from Combined Lighting Lesson	229	73.5 Textured Terrain	289
59.1 Emissive Light	231	73.6 Terrain Shader Structure	290
60.1 Gouraud Shader Structure	237	73.7 Final Terrain Render	291
60.2 Output from Gouraud Shading Lesson	238	74.1 Fog Shader Structure	293
61.1 Flat, Gouraud and Phong Shading	239	74.2 Linear Fog	294
61.2 Normal Interpolation Across a Surface	240	74.3 Exponential Fog	295
61.3 Phong Shader Structure	241	74.4 Exponential Squared Fog	296
62.1 Point Light Example	243	76.1 Line and Triangle Adjacency	300
62.2 Point Light with No Distance	244	76.2 Copy Geometry Shader	304
62.3 Point Light with Range	245	76.3 Copy Geometry Shader Output	305
62.4 Scaled Point Light	246	77.1 Explode Shader Structure	306
62.5 Output from Point Light Lesson	248	77.2 Output from Exploding Shade Shader	307
63.1 Spot Light	250	78.1 Normal Rendering Shader Structure	308
63.2 Output from Spot Light Lesson	252	78.2 Output from Normals Application	309
64.1 Output from Multiple Lights Lesson	256	79.1 Billboarding in Doom	311
65.1 Structure of an Effect with Multiple Parts	258	79.2 Billboarding Process	312
65.2 Output from Multi-file Shader Lesson	261	79.3 Billboarding Shader Structure	312
		79.4 Billboarding Lesson Output	313
		80.1 Transform Feedback Loop	314
		80.2 Particle Shader Structure	315
		80.3 Output from Particles Lesson	320
		81.1 Output form Smoke Application	321

83.1 Graphics Pipeline	324
83.2 Using a Frame Buffer to Capture a Render	326
84.1 Displaying a Frame Buffer to the Screen	328
85.1 Greyscale Post-Process	330
86.1 utput from Blur Shader	332
87.1 Motion Blur Post-process Pipeline	334
87.2 Motion Blur Post Process	335
88.1 Depth of Field Post-process Pipeline	336
88.2 Output from Depth-of-Field Post Pro- cess	337
89.1 Output from Masking Post-Process	338

List of Algorithms

1	Divide Triangle at 0 Subdivisions	89
2	Divide Triangle for 1 Subdivision	91
3	Recursive Sierpinski Gasket	91
4	Sphere Subdivision for 0	94
5	Sphere Subdivision for 1	95
6	Recursive Sphere Subdivision	95
7	Blending Shader	133
8	Dissolve Shader	137
9	View Matrix Generation	171
10	Perspective Projection Matrix Generation	178
11	Target Camera Update	184
12	Free Camera Update	187
13	Using the Mouse to Control Camera Rotation	189
14	Chase Camera Update	193
15	Arc-ball Camera Update	196
16	Converting a Screen Position to a World Ray	202
17	Testing OOOB and Ray Intersection	204
18	Combined Lighting Algorithm	228
19	Gouraud Vertex Shader	237
20	Gouraud Fragment Shader	237
21	Calculate Shadow Algorithm	267
22	Calculating a Normal from a Normal Map	269
23	Generating Geometry Data for Terrain	285
24	Generating Index Data for Terrain Geometry	285
25	Generating Normals for Terrain	287
26	Generating Texture Coordinates for Terrain	287
27	Generating Texture Weights for Terrain	290

Part I

Getting Started

Lesson 1

Getting Started

Welcome to Computer Graphics! This material is designed to take you from a basic start in rendering using OpenGL to a using some techniques that will enable you to create realistic looking scenes that will work in real-time. The workbook style is full of lessons to complete (lots of programming in C++ and GLSL) as well as covering some of the mathematics you need to understand and work with graphics in this manner.

The pre-requisite knowledge you require is:

- a working knowledge of object-oriented programming in a high-level language. It an ideal world this is C++ but Java and C# programmers should be able to cope with the material. We do very little pointer work.
- at least a grasp of mathematical concepts such as trigonometry, algebra, and geometry. Linear algebra and matrix mathematics are very advantageous.
- a willingness to spend time solving the problems presented in this workbook. Some of the exercises are challenging and require effort. There is no avoiding this. However, solving these problems will significantly aid your understanding.

1.1 Getting the content

1.1.1 Pulling from Git

The content is available from a Git repository. If you are unsure about what Git is, now is the time to find out before continuing on. We won't go anywhere.

```
git clone https://github.com/edinburgh-napier/set08116
```

1.1.2 Creating the Solution, with CMake

Once you have the source downloaded you will need to use CMake to make a build file for you.

Follow this guide:

https://github.com/edinburgh-napier/aux_guides/blob/master/cmake_guide.pdf

The source directory is `set08116/labs`

Remember to place the build folder outside of the set08116 folder. Preferably your desktop, NOT your H drive.

Once configured and generated, you can open the .sln file in the build folder. You should not need to touch any solution or project settings form within Visual Studio. The solution is set up so you don't have to do much work yourself or even understand Visual Studio settings. Everything is there for you to work through the different lessons. To make sure you are working on the correct lesson, right click on the project you want to work on in Visual Studio and select *Set as StartUp Project*.

1.1.3 Workbook Style

The workbook is task and lesson based, requiring you to solve problems to complete the code provided as a starting point for the lesson. This code will either be in the main application source file (.cpp) for the lesson, or in one or more shader files (typically found in the same folder rn in: `resources/shaders` folder). The sections you have to complete are highlighted in stars:

```
1 // ****
2 // You have to complete this problem
3
4
5 // ****
```

The problems faced generally build on previous lessons until you become familiar with the work involved. The number of empty lines between the asterisks give an indication on how many lines the solution should be.

1.2 Graphics Framework

In this module we will be working with a graphics framework that has been developed specifically for the work we will be undertaking in computer graphics. The framework itself is fairly lightweight, but provides some object-oriented wrappers around standard OpenGL (which is a C API). The main job of the graphics framework is to combine a collection of libraries together to allow us to focus on computer graphics rather than writing some boiler plate code.

1.2.1 Libraries

The libraries used in the graphics framework are:

OpenGL takes care of our computer graphics rendering by enabling us to talk to our graphics card. Most of the other libraries we will use provide an interface into OpenGL. In Windows, we use the `g1/g1.h` header and the `OpenGL32.lib` library file, both of which come with the standard Windows SDK.

GLEW the GL Extension Wrangler. GLEW is a standard inclusion in modern OpenGL applications as it provides access to the modern OpenGL API.

GLFW the GL FrameWork (although it is not really a framework). GLFW provides a simple mechanism to create a window that we can use to draw to, and simple methods to access keyboard and mouse input.

GLM the GL Mathematics library. GLM is also cross-platform and a header only library (it has a number of different headers we include for different purposes). GLM provides the mathematical types and functions required to work with OpenGL (which provides no basic mathematical types). GLM is designed to be similar in syntax to GLSL (the GL Shader Language) which we will become familiar with in the module.

DevIL allows us to load textures and attach them to OpenGL (OpenGL does not provide texture loading mechanisms). As with our other libraries, Devil is cross-platform.

AssImp the Asset Importer library - a cross platform library that makes it easy to load in 3D model data (again OpenGL has no such mechanism).

As we are compiling all our dependencies from sources, the project also downloads some additions libraries required by the libraries described above:

Zlib A general purpose compression library. We use this mainly for file I/O

libJPEG A standard cross platform .jpg image encoder/decoder implementation

libPNG A standard cross platform .png image encoder/decoder implementation

As you can see, we work using a number of cross-platform libraries. This means that, although we will be working on Windows using Visual Studio, our code will work on any standard desktop platform (i.e. Linux and Mac OS X).

1.2.2 app Class

The graphics framework takes care of most of the back end initialisation and handling of windows. To get an application working you have to use the `app` class. In our main method we have to create an instance of `app`, attach a function that we will use to perform our rendering, and then tell the application to run.

To create an application we use the following:

```
1 app application;
```

To set our render method we use the following:

```
1 application.set_render([name of function]);
```

The `app` class has a number of helper methods to set functions such as update and load content. As a minimum, the `app` class requires a render function. When we look at an example shortly.

Finally, to run an `app` we use the `run` method:

```
1 application.run();
```

We will look at other uses of the `app` class as we work through the module.

1.2.3 renderer Class

Another central class to our work is the `renderer`. It will take care of most of the underlying calls to OpenGL to actually draw our geometry. It will also take care of managing our window, and a few other useful features. It is really there to stop us writing repetitive code when drawing geometry and do some safety checks. We will look at the calls provided by the `renderer` as we work through the module.

1.2.4 Basic Graphics Application

In project “01 Getting Started” you will find the following basic code for a simple application using the graphics framework. As you can see, getting an application up and running is very simple.

```

1 // The main header for the graphics framework
2 #include "graphics_framework.h"
3
4 // The namespaces we are using
5 using namespace std;
6 using namespace graphics_framework;
7
8 // Our rendering code will go in here
9 bool render()
10 {
11     return true;
12 }
13
14 void main()
15 {
16     // Create application
17     app application;
18     // Set render function
19     application.set_render(render);
20     // Run application
21     application.run();
22 }
```

You should run this application to see the output. You will get a cyan coloured window as shown in Figure 1.1.

We use cyan as a background colour as it allows us to see our rendered objects easily. It is also similar to a sky colour (cyan in RGBA terms is just green and blue combined).

1.3 What next?

Move onto the next lesson and learn how to render a triangle using our graphics framework.



Figure 1.1: Output from Basic Application

Lesson 2

Rendering a Triangle

Our first application is going to be very simple - it will render a red triangle in the world. The framework will take care of most of the work for us, and the code you have to change is very simple - you just have to define the position data that makes up the three corners of the triangle. Everything else has been taken care of for you.

2.1 Defining a Triangle

Triangles make up the 3D objects that we use in our graphics applications. As we work through our first few lessons this will become very apparent. Triangles are also a useful primitive. Given any three points in 3D space, we know that it will form a triangle - it is guaranteed to be planar. No other polygon can guarantee this.

A triangle is therefore just three points in space. We are working in 3D, so our points will need x , y , and z components. Let us define our three corners as follows:

$$\begin{aligned} &(0.0, 1.0, 0.0) \\ &(-1.0, -1.0, 0.0) \\ &(1.0, -1.0, 0.0) \end{aligned}$$

Winding Order

It is important the order in which you declare your vertices. To improve performance, the graphics card will remove faces that are not facing the camera. A triangle has a front face and a back face. The front face is rendered if it faces the camera. The back face is not rendered if it faces the camera.

We can tell OpenGL which face is the front and which is the back by telling OpenGL which order the vertices are declared in. These can be:

Counter Clockwise vertices are declared in a counter-clockwise order to determine which side faces forward. This can also be called right-handed winding. Imagine curling the fingers of your right hand in the direction of the vertex declaration. Your thumb will face towards you. This is the default method OpenGL uses, and is the method that our graphics framework adheres to.

Clockwise vertices are declared in a clockwise order to determine which side faces forward.

This can also be called left-handed winding. Imagine curling the fingers of your left hand in the direction of the vertex declaration. Your thumb will face towards you.

So for us to see our triangle we should declare our vertices in counter clockwise order - or the order in which they are shown above.

2.2 geometry Class

The graphics framework declares a class called `geometry` which maintains data related to a piece of geometry. It is a fundamental type for our graphics framework. The `renderer` class understands this data type as one of the ones it can attempt to draw.

Our job is to add data to an instance of the `geometry` class that describes the 3D object we wish to draw. A 3D object actually has a number of different values that can make describe it such as position data, colour data, texture coordinates and surface normals. An instance of `geometry` contains a collection of buffers that describe the 3D object.

Your task in this lesson is to define the position data. To do this, you need to declare a vector of data and then attach it to the `geometry` object.

2.2.1 Creating a Buffer

The `geometry` class provides the following method to attach a buffer:

```
1 add_buffer(vector<[type]>, index);
```

The `add_buffer` method takes two values - a `vector` (a dynamically sized array) of data - typically of types `vec2`, `vec3`, or `vec4` and an index. The index is a message to the graphics card to tell it where to send data to the GPU - we will look at this later.

To declare our buffer of data for our triangle we would use the following:

```
1 vector<vec3> positions
2 {
3     vec3(0.0f, 1.0f, 0.0f),
4     vec3(-1.0f, -1.0f, 0.0f),
5     vec3(1.0f, -1.0f, 0.0f)
6 };
```

C++11 - Initializer Lists

A new addition to C++11 is the concept of initializer lists. An initializer list allows us to pass values to the constructor of an object in a list format (between curly braces). This means our syntax above is perfectly legal, and we could add as many vertices as we like to this declaration. This is quite a useful syntax when we are declaring lists of values.

2.2.2 Adding to Geometry Object

We already saw that our `geometry` class uses the `add_buffer` to add buffer data. Behind the scenes, the graphics framework takes care of allocating memory on the GPU and attaching it to the existing data describing the geometry.

Data Types

We are using a library called GLM (GL Mathematics) which provides a number of useful data types. We will explore GLM further as we work through the module. However, at the moment it is worth knowing the following data types:

`vec2` a two-dimensional vector. Generally used for texture coordinates in our work.

`vec3` a three-dimensional vector. Used for position data and surface normals.

`vec4` a four-dimensional vector. To start with we will be using these for colours (RGBA).

Camera Position and Target

We have not really covered cameras yet - but we will later in the module. At the moment, the application creates a camera for us. This camera is placed at position (10, 10, 10). It is also set to look at position (0, 0, 0). When we render our triangle it will be at relative to position (0, 0, 0). This means that we will be looking down on our triangle from the

2.2.3 Try it - Set the Buffer Data

You will find in the lesson code a piece a commented area in the `load_content` function telling you to declare the position data. Do this now. When run, you will get the output shown in Figure 2.1.

2.3 Exercise

Try changing the colour data so that the one colour of the triangle is red, one is green, and one is blue. What is the output? What does it tell you about how colours are determined for pixels in 3D rendering?

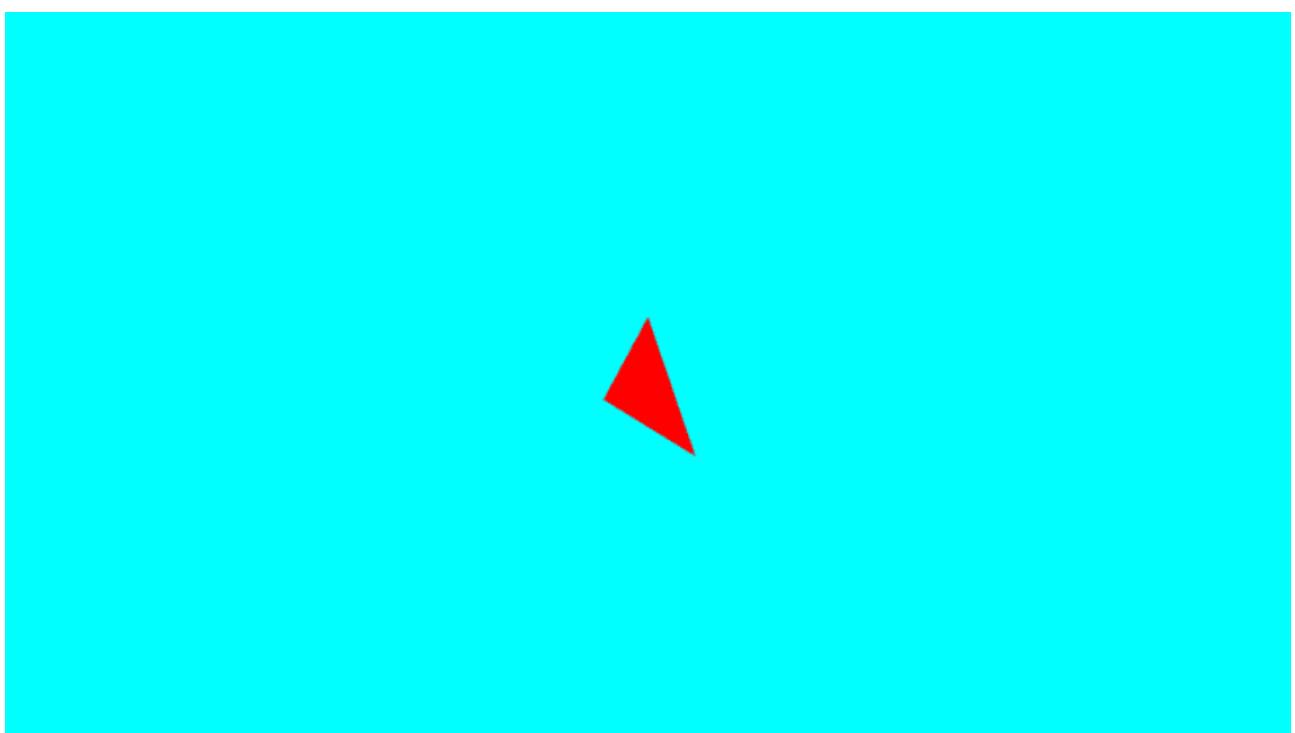


Figure 2.1: Output from Triangle Application

Lesson 3

Rendering a Quad with Triangles

Now let us expand our triangle example to draw a quad. We are actually going to do this in a number of different ways over the next few lessons, which will illustrate some of the different techniques we can use to draw items on the screen. First, we will modify the existing triangle example so that we draw two triangles to create a square.

3.1 Code

The project code for this lesson is the same as the previous lesson. Your job this time is to define two triangles so that they make up a quad. Remember your winding order. To help, look at Figure 3.1.

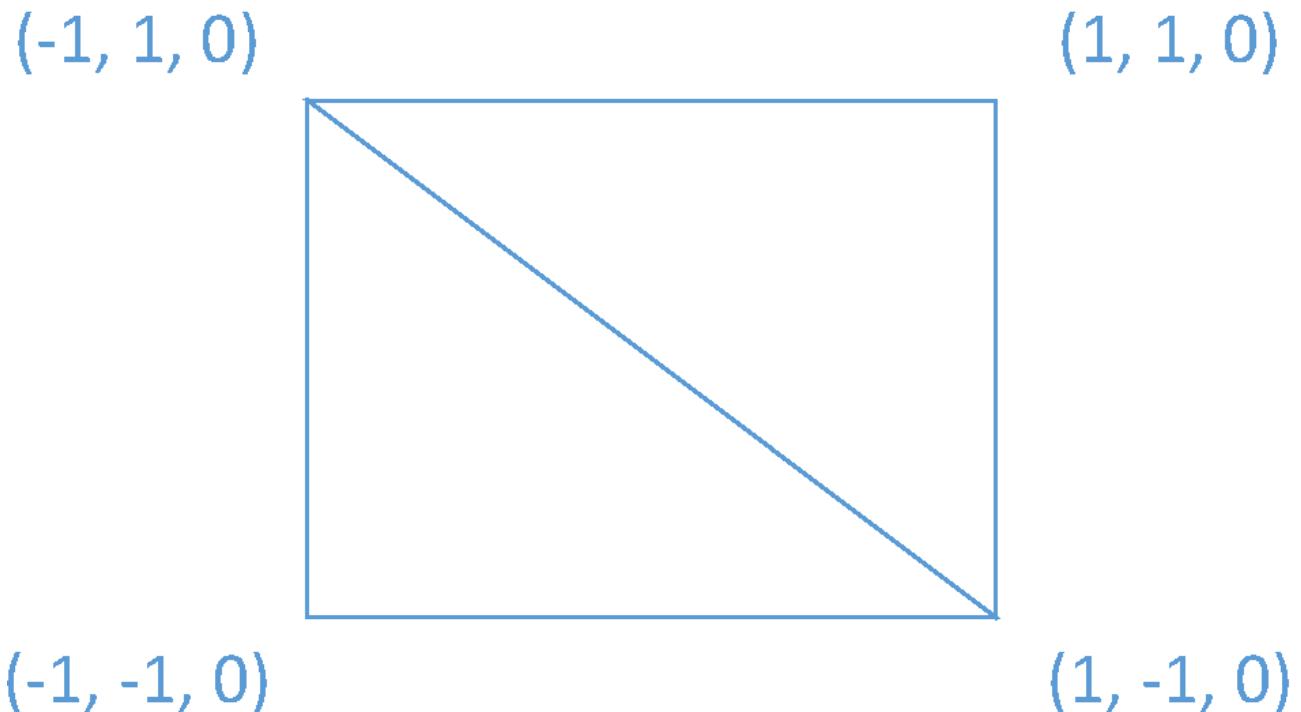


Figure 3.1: A Quad Made of Two Triangles

Here we are just define two triangles, and render both of these. The output is shown in Figure 3.2.

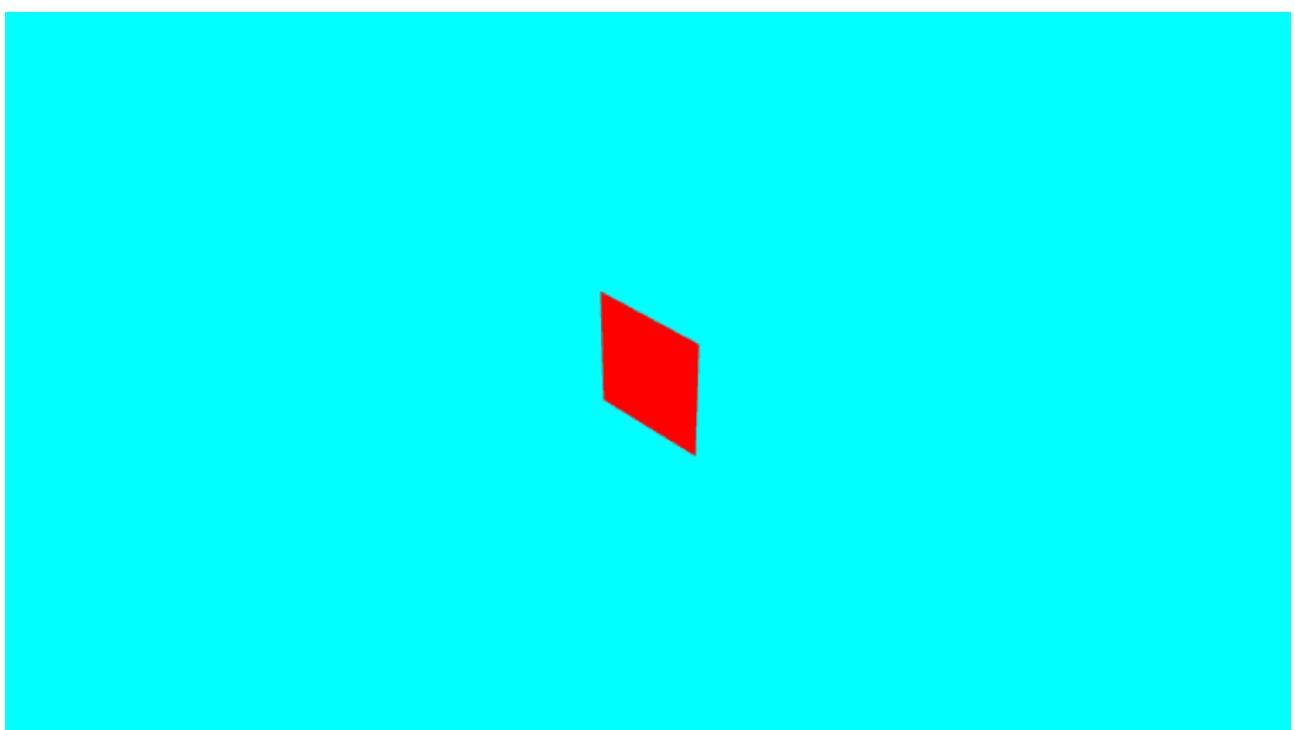


Figure 3.2: A Quad using Triangles

Lesson 4

Rendering a Quad with Quads

Let us now repeat this output but using a quad instead of two triangles. OpenGL provides a method to render a quad from four vertices. This is quite handy in a number of circumstances, although you have to remember that four 3D positions are not guaranteed to form a quad (they may not be on the same plane).

To tell OpenGL that we wish to have quads for our geometry, you can use the `set_type` method on the `geometry` object. For example, to set the type of geometry primitive we are using to quads, we would use the following:

```
1 geom.set_type(GL_QUADS);
```

Using this line of code will mean we only need the four corners to define a quad rather than six when using two triangles. `GL_QUADS` is a value provided by OpenGL. There are a few more values which we will look at in the following lessons.

You will need to update the project to use the correct type, and also use only four positions. The output will be the same as the previous lesson.

Lesson 5

Rendering a Quad with Triangle Strips

To save storage, it is often useful to work in the concept of strips of data. What these means is that we reuse previous vertices to create the current triangle. As an example, consider Figure 5.1.

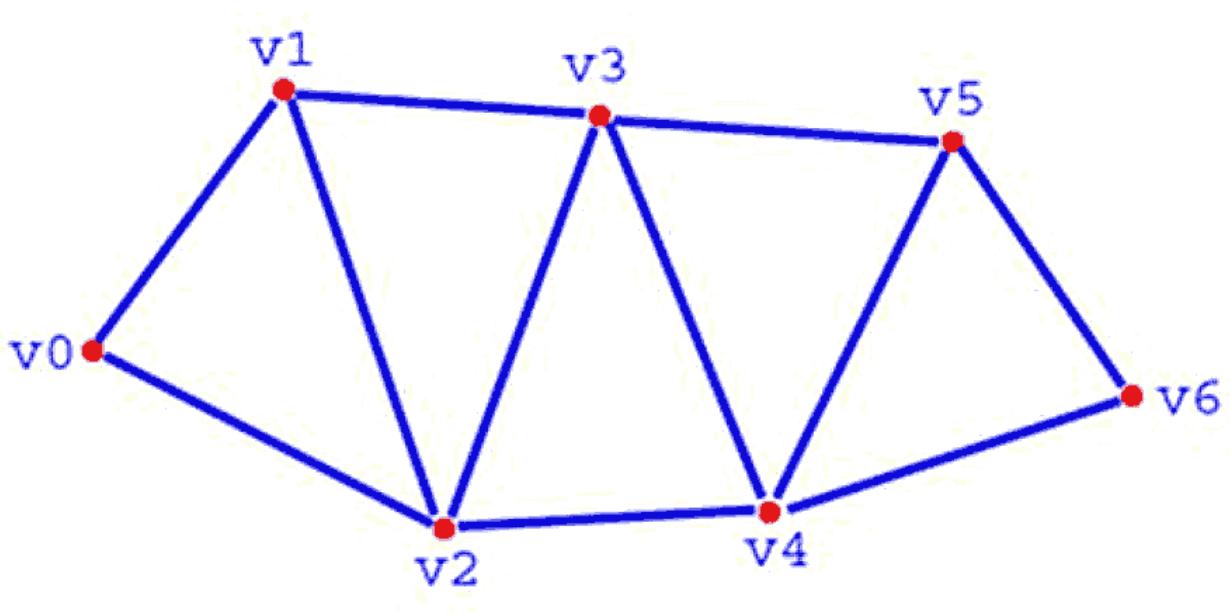


Figure 5.1: Triangle Strip

Triangle 1 is defined by v0, v1, v2. Triangle 2 is made from v1, v2, v3. Triangle 3 is made from v2, v3, v4, etc. For the actual call, all we need to define is the 7 vertices (0 - 7), but we still get 5 triangles.

Your job is to replicate this in our graphics framework. You need four positions (the positions that define the quad) and consider the winding order of the first triangle. The second triangle will use the last two vertices of the previous triangle. The winding order flips at this point to make life easier.

The type of geometry you want this time is `GL_TRIANGLE_STRIP`. The output will be the same as the previous quad output.

Lesson 6

Rendering a Quad with Lines

So far we have focused on using polygons to render our quads. Let us now do some rendering using just lines, therefore just drawing the outline of the quad.

Lines are defined by two points. Therefore for our quad we will require eight vertices. The type required for lines is `GL_LINES`. You need to set this in the project for this lesson, and then define your eight vertices. The output is shown in Figure 6.1.

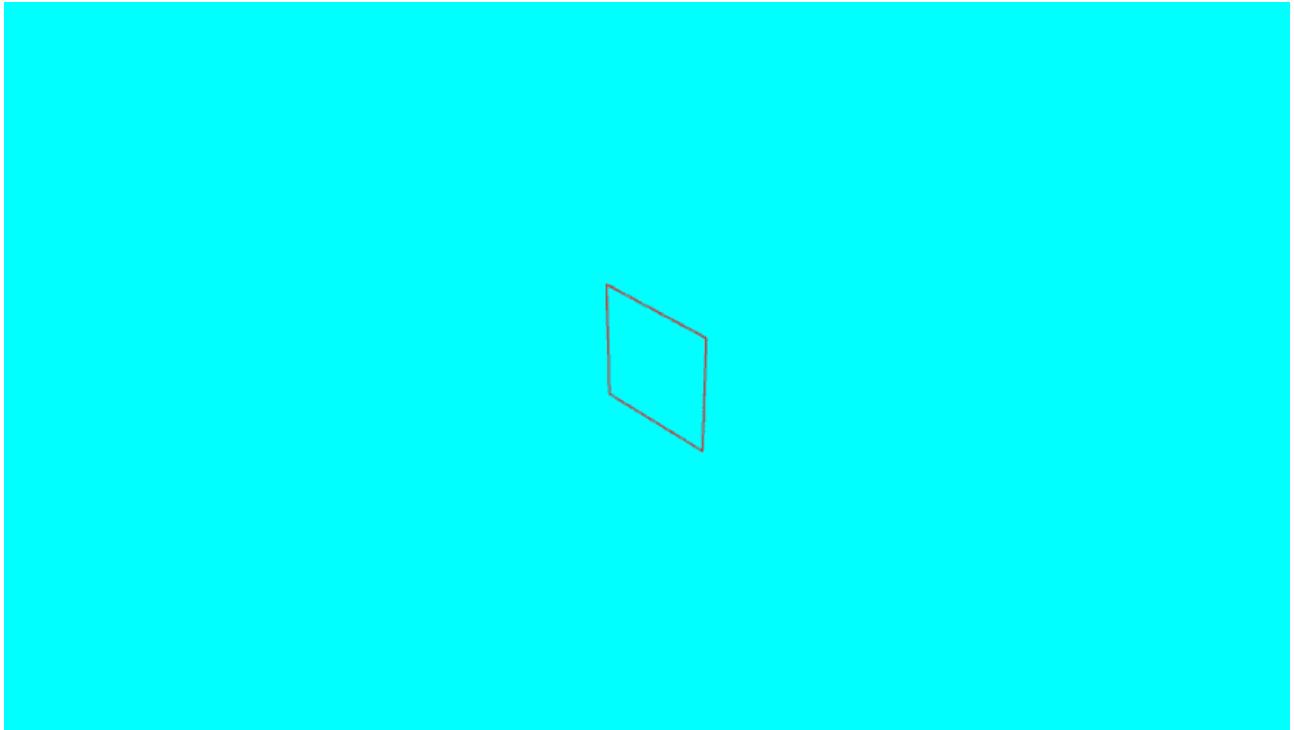


Figure 6.1: **Lines Output**

Lesson 7

Rendering a Quad with Line Strips

As with triangles, we can reuse our vertex data for defining lines. As lines are defined by two points, we can define a strip of lines by defining our initial two points for a line, and then reusing the previous vertex and a new vertex to define a line.

To define a line strip, we use the type `GL_LINE_STRIP`. This time you will only need five vertices. Modify the project for this lesson to do this.

Lesson 8

Rendering a Quad with Line Loops

Our final line approach we will use in the line loop. A line loop is almost identical to a line strip, except that the final vertex is connected to the first vertex. This means that you only require four vertices now to define the quad.

The geometry type for a line loop is `GL_LINE_LOOP`. Change the project for this lesson so that it uses a line loop and has the correct vertices defined.

Lesson 9

Rendering a Quad with Triangle Fans

Our final approach to rendering a quad is using a triangle fan. A triangle fan is useful as it allows us to define a shared vertex (a centre point), and then we work round defining triangles using the initial point and the last two defined points. Figure 9.1 illustrates this idea.

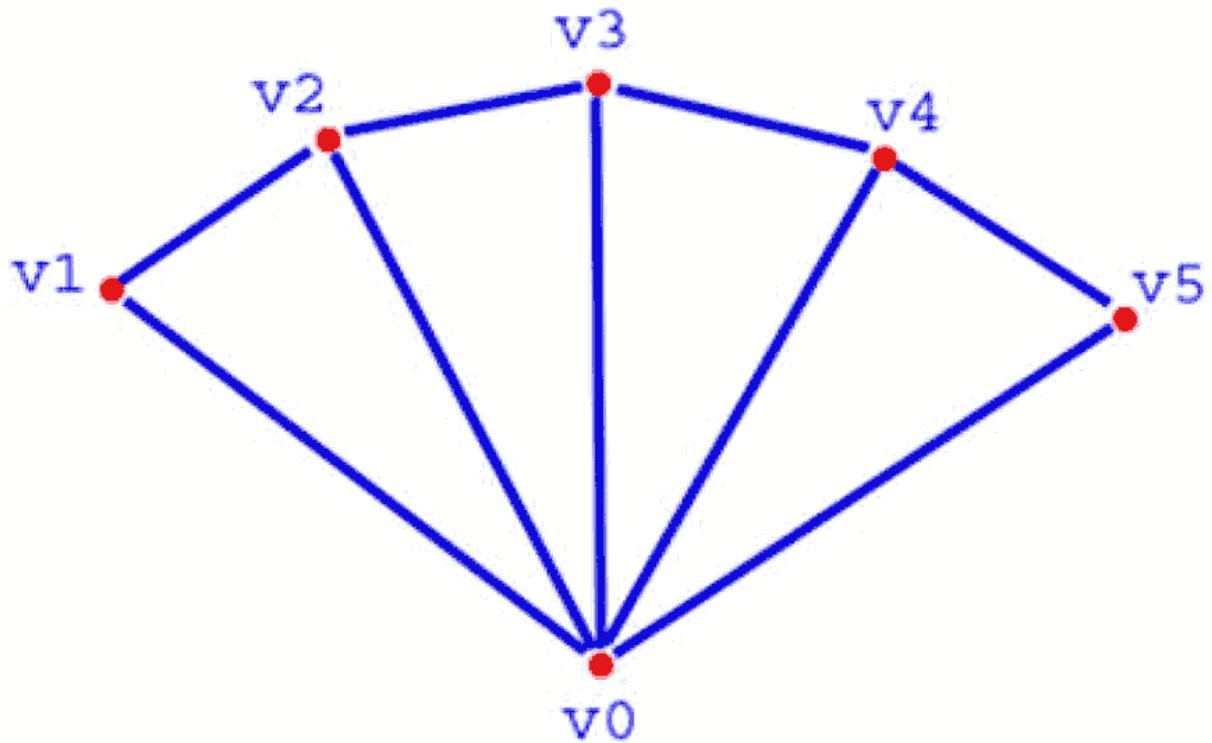


Figure 9.1: Triangle Fan

The geometry type for triangle fan is `GL_TRIANGLE_FAN`. The project for this lesson requires you to define the geometry type and then define the necessary vertices for the quad.

Lesson 10

Transformations

We have now got up and running with OpenGL, but so far we are only performing some basic rendering ideas. We are not really doing anything about having our shapes move. This lesson discusses transformations, and although we will not be coding any of these concepts right now, they are important aspects to understand.

10.1 Vectors

Hopefully by now you understand what is meant by a vector as it is one of the most important constructs when working in 3D simulation and games development in particular. Vectors are used for both the positioning of objects in space (both 2D and 3D) and for the movement of these objects. As a quick refresher, a 3D vector can be defined as follows:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

The vector has three components –one for each of the distances along the relative axis. There are a number of standard operations we can perform on a vector.

10.1.1 Vector Addition

Adding vectors is relatively simple and we just add the individual components together:

$$\begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix} + \begin{bmatrix} x_2 \\ y_2 \\ z_2 \end{bmatrix} = \begin{bmatrix} x_1 + x_2 \\ y_1 + y_2 \\ z_1 + z_2 \end{bmatrix}$$

The same rule applies for subtraction.

10.1.2 Scaling Vectors

Scaling a vector involves us just multiplying each component by the scalar value:

$$s \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} sx \\ sy \\ sz \end{bmatrix}$$

The same rule applies for division (division by s is the same as scaling by $\frac{1}{s}$).

10.1.3 Length of a Vector

The length (or magnitude) of a vector can be determined using standard Pythagoras theorem:

$$\left\| \begin{bmatrix} x \\ y \\ z \end{bmatrix} \right\| = \sqrt{x^2 + y^2 + z^2}$$

The brackets $\| \|$ indicate that we want the length of the vector. Notice that the value will always be positive.

10.1.4 Normalising a Vector

Normalising a vector is an important operation when working with computer graphics (as you shall soon see). Normalising means converting a vector so that it still faces in the same direction, but has a unit length (length of 1). To do this, we only need to scale the vector by the inverse of its length:

$$\hat{u} = \begin{bmatrix} \frac{x}{\|u\|} \\ \frac{y}{\|u\|} \\ \frac{z}{\|u\|} \end{bmatrix}$$

Note that we use the accent $\hat{\cdot}$ to denote that we have a normalised vector.

10.1.5 Dot Product

Multiplying two vectors together can be done in a number of ways. The first of these we will cover is called the dot product, and it is calculated as follows:

$$\begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix} \cdot \begin{bmatrix} x_2 \\ y_2 \\ z_2 \end{bmatrix} = x_1 \times x_2 + y_1 \times y_2 + z_1 \times z_2$$

Notice that the result of this operation is a scalar value, not another vector.

Properties of the Dot Product

The dot product between two vectors actually provides a number of useful properties of the two vectors. The first is that the dot product between two vectors has a relationship to the cosine of the angle between the two vectors:

$$u \cdot v = \|u\| \|v\| \cos \theta$$

$\|u\|$ means the length of vector u , and θ is the angle between the two vectors.

Because there is a relation to the dot product and the cosine of the angle, we can also determine when two vectors are orthogonal (or perpendicular, at right angles), by the observation that $\cos \frac{\pi}{2} = 0$, thus $\|u\| \|v\| \cos \frac{\pi}{2} = 0$. Therefore, if the dot product of two vectors is 0, we know that they are orthogonal.

Furthermore, we can determine whether an angle is greater than or less than $\frac{\pi}{2}$ by the value of the dot product:

$$\begin{aligned} u \cdot v < 0 &\Rightarrow \theta > \frac{\pi}{2} \\ u \cdot v > 0 &\Rightarrow \theta < \frac{\pi}{2} \end{aligned}$$

A final property of the dot product, again because of the cosine relation, is the calculation of the scalar projection of a vector onto another vector. The scalar projection tells us how far along a vector another vector reaches, which can be useful in some applications (we can use it in reflection calculations for example). The scalar projection can be calculated as follows:

$$a_b = \|a\| \cos \theta = \frac{a \cdot b}{\|b\|}$$

We can use this value to create a vector of the length of the scalar projection that is also parallel to the vector of interest. In other words, we can calculate a projected vector onto our original vector. The calculation for this follows from the one above:

$$\text{proj}_b a = \frac{a \cdot b}{\|b\|^2} b$$

10.1.6 Cross Product

The cross product is another method of multiplying two vectors together. This time the result is another vector. The cross product of two vectors can be calculated as:

$$\begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix} \times \begin{bmatrix} x_2 \\ y_2 \\ z_2 \end{bmatrix} = \begin{bmatrix} y_1 z_2 - y_2 z_1 \\ z_1 x_2 - z_2 x_1 \\ x_1 y_2 - x_2 y_1 \end{bmatrix}$$

Notice the pattern forms a cross for each individual component (hence the name cross product).

Properties of the Cross Product

As the dot product, the cross product has certain properties that are of interest when manipulating geometry. The first is that when we calculate the cross product of two vectors, the new vector is orthogonal (perpendicular) to the two original vectors. That is:

$$\begin{aligned}(u \times v) &\perp u \\ (u \times v) &\perp v\end{aligned}$$

This allows us to calculate vectors perpendicular to others, which will come in useful when we are dealing with cameras much later in the lessons.

Because the cross product of two vectors is orthogonal to the original two vectors, we can use this information to calculate surface normals (that is, vectors that are unit length and perpendicular to a surface). Consider that two vectors can create a surface as shown in Figure 10.1.

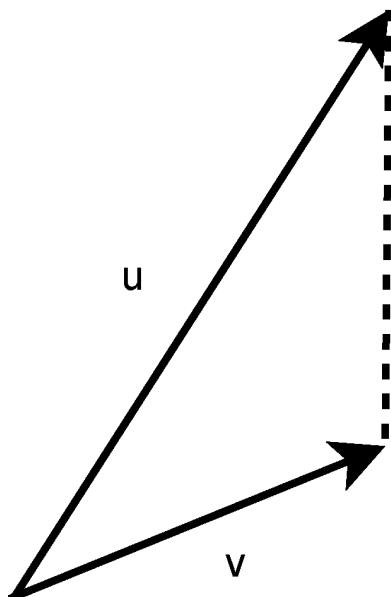


Figure 10.1: Two Vectors forming a Triangle

Then the cross product of u and v will be orthogonal to the surface created by the two vectors. To calculate the surface normal, we use the following:

$$u \hat{\times} v = \frac{u \times v}{\|u \times v\|}$$

Surface normals are important when we consider lighting calculations later in the lesson schedule.

Finally, the cross product, like the dot product, has a relation to the size of the angle between the two vectors. However, this time the relation is with the sine value of the angle. The cross product is related to the sine of the angle between the two vectors as follows:

$$\|u \times v\| = \|u\| \|v\| \sin \theta$$

10.1.7 Exercises

Take your time and do the following exercises. The answers are at the end of the chapter. Avoiding doing the exercises or delaying until later won't help you. This is part of the learning process.

Working with Vectors

1. Given the vectors $\mathbf{u} = \langle 2, -7, 1 \rangle$, $\mathbf{v} = \langle -3, 0, 4 \rangle$ and $\mathbf{w} = \langle 0, 8, -8 \rangle$, find:
 - (a) $\mathbf{u} + \mathbf{v}$
 - (b) $\mathbf{v} - \mathbf{w}$
 - (c) $-3\mathbf{u}$
 - (d) $3\mathbf{u} - 4\mathbf{v}$
 - (e) $2\mathbf{u} + 4\mathbf{v} - 5\mathbf{w}$
2. Let $\mathbf{u} = \langle 2, -7, 1 \rangle$, $\mathbf{v} = \langle -3, 0, 4 \rangle$ and $\mathbf{w} = \langle 0, 8, -8 \rangle$, Find:
 - (a) $\mathbf{u} \cdot \mathbf{v}$
 - (b) $\mathbf{u} \cdot \mathbf{w}$
 - (c) $\mathbf{u} \times \mathbf{w}$
 - (d) $\mathbf{v} \times \mathbf{w}$
 - (e) $\|\mathbf{u}\|$ - the length or magnitude of \mathbf{u}

Normalising Vectors

1. Calculate the length of the following vectors:
 - (a) $\langle 3, 4, 0 \rangle$
 - (b) $\langle 10, 20, 5 \rangle$
 - (c) $\langle 1, 6, -4 \rangle$
 - (d) $\langle 2, 2, 2 \rangle$
 - (e) $\langle 8, 12, -30 \rangle$
2. What is the unit length (normalised) form of the following vectors:
 - (a) $\langle 3, 4, 0 \rangle$
 - (b) $\langle 10, 20, 5 \rangle$
 - (c) $\langle 1, 6, -4 \rangle$
 - (d) $\langle 2, 2, 2 \rangle$
 - (e) $\langle 8, 12, -30 \rangle$

Dot Product

1. Given the vectors $\mathbf{u} = \langle 2, -7, 1 \rangle$, $\mathbf{v} = \langle -3, 0, 4 \rangle$ and $\mathbf{w} = \langle 0, 8, -8 \rangle$, find:

- (a) $\mathbf{u} \cdot \mathbf{v}$
- (b) $\mathbf{v} \cdot \mathbf{u}$
- (c) $\mathbf{v} \cdot \mathbf{w}$
- (d) $\mathbf{w} \cdot \mathbf{v}$
- (e) $\mathbf{u} \cdot \mathbf{w}$

Angle between Vectors

1. What is the angle (in radians) between the following pairs of vectors:

- (a) $\langle 1, 0, 0 \rangle, \langle 0, 1, 0 \rangle$
- (b) $\langle 0, 1, 0 \rangle, \langle 0, 0, 1 \rangle$
- (c) $\langle 1, 1, 0 \rangle, \langle 0, 1, 1 \rangle$
- (d) $\langle 1, 1, 1 \rangle, \langle 0, 1, 1 \rangle$
- (e) $\langle 1, 0, 0 \rangle, \langle 1, 0, 0 \rangle$

Projection Vectors

1. Calculate the projection of a onto b given:

- (a) $\mathbf{a} = \langle 2, -7, 1 \rangle, \mathbf{b} = \langle -3, 0, 4 \rangle$
- (b) $\mathbf{a} = \langle -3, 0, 4 \rangle, \mathbf{b} = \langle 2, -7, 1 \rangle$
- (c) $\mathbf{a} = \langle 2, -7, 1 \rangle, \mathbf{b} = \langle 0, 8, -8 \rangle$
- (d) $\mathbf{a} = \langle 0, 8, -8 \rangle, \mathbf{b} = \langle 2, -7, 1 \rangle$
- (e) $\mathbf{a} = \langle -3, 0, 4 \rangle, \mathbf{b} = \langle 0, 8, -8 \rangle$

Cross Product

1. Given the vectors $\mathbf{u} = \langle 2, -7, 1 \rangle$, $\mathbf{v} = \langle -3, 0, 4 \rangle$ and $\mathbf{w} = \langle 0, 8, -8 \rangle$, find:

- (a) $\mathbf{u} \times \mathbf{v}$
- (b) $\mathbf{v} \times \mathbf{u}$
- (c) $\mathbf{v} \times \mathbf{w}$
- (d) $\mathbf{w} \times \mathbf{v}$
- (e) $\mathbf{u} \times \mathbf{w}$

2. Calculate the following:

- (a) $\langle 1, 0, 0 \rangle \times \langle 0, 1, 0 \rangle$

- (b) $<0, 1, 0> \times <1, 0, 0>$
- (c) $<1, 0, 0> \times <0, 0, 1>$
- (d) $<1, 0, 0> \times <0, 0, -1>$
- (e) $<1, 0, 0> \times <-1, 0, 0>$

10.2 Matrices

Now that we have refreshed ourselves with vectors, we can move onto matrices. A matrix is a $m \times n$ grid of values such as:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

From a 3D graphics point of view, we are only interested in matrices of sizes 3×3 and 4×4 . We will see why shortly. For now, let us consider some basic operations for matrices.

10.2.1 Addition

Adding two matrices together is similar to vectors, and just involves us adding the separate components together:

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} + \begin{bmatrix} b_{00} & b_{01} & b_{02} \\ b_{10} & b_{11} & b_{12} \\ b_{20} & b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{00} + b_{00} & a_{01} + b_{01} & a_{02} + b_{02} \\ a_{10} + b_{10} & a_{11} + b_{11} & a_{12} + b_{12} \\ a_{20} + b_{20} & a_{21} + b_{21} & a_{22} + b_{22} \end{bmatrix}$$

The same rule applies for subtraction.

10.2.2 Scaling

Scaling a matrix is also similar to vector operations, and involves us scaling the separate components:

$$s \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} sa_{00} & sa_{01} & sa_{02} \\ sa_{10} & sa_{11} & sa_{12} \\ sa_{20} & sa_{21} & sa_{22} \end{bmatrix}$$

The same division approach for vectors also applies to matrices.

10.2.3 Multiplying

Multiplying two matrices together requires some more work. Let us consider two 3×3 matrices as follows:

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{00} & b_{01} & b_{02} \\ b_{10} & b_{11} & b_{12} \\ b_{20} & b_{21} & b_{22} \end{bmatrix}$$

We can actually consider the first matrix as three vectors as follows:

$$\begin{bmatrix} a_{0x} \\ a_{1x} \\ a_{2x} \end{bmatrix}$$

That is three row vectors $[a_{00} \ a_{01} \ a_{02}]$, $[a_{10} \ a_{11} \ a_{12}]$ and $[a_{20} \ a_{21} \ a_{22}]$. We can also consider the second matrix as three vectors as follows:

$$\begin{bmatrix} b_{x0} & b_{x1} & b_{x2} \end{bmatrix}$$

That is three column vectors $[b_{00} \ b_{10} \ b_{20}]$, $[b_{01} \ b_{11} \ b_{21}]$ and $[b_{02} \ b_{12} \ b_{22}]$. We can now define matrix multiplication as follows:

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{00} & b_{01} & b_{02} \\ b_{10} & b_{11} & b_{12} \\ b_{20} & b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{0x} \cdot b_{x0} & a_{0x} \cdot b_{x1} & a_{0x} \cdot b_{x2} \\ a_{1x} \cdot b_{x0} & a_{1x} \cdot b_{x1} & a_{1x} \cdot b_{x2} \\ a_{2x} \cdot b_{x0} & a_{2x} \cdot b_{x1} & a_{2x} \cdot b_{x2} \end{bmatrix}$$

A possibly easier visualisation of this is taken from Wikipedia and shown in Figure 10.2.

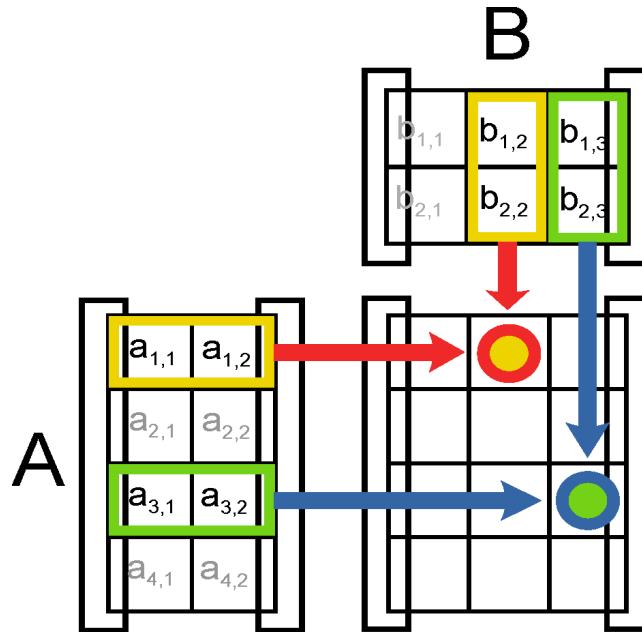


Figure 10.2: Matrix Multiplication

Notice that matrix multiplication can only take place between $m \times n$ and $n \times p$ matrices, and results in a matrix that is $m \times p$ in size.

From the point of view of 3D graphics, it has already been mentioned that we are interested in 3×3 and 4×4 matrices. Multiplying these together is not an issue. We are also interested in vectors, which in 3D terms look like a 3×1 matrix:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

We will be manipulating our 3D vectors with 3×3 and 4×4 matrices. Multiplying a 3×3 matrix and a 3×1 vector is possible, and will result in a 3×1 value (another vector). For a 4×4 matrix, we need to work with a 4×1 vector. Therefore we will be representing our vectors as follows:

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}$$

Depending on if we are working with vectors (a direction or magnitude) or a position in space, we set w to be 0 or 1 respectively.

This will allow us to perform vector transformations by multiplying our vectors by our matrices:

$$\mathbf{M} \times u = \begin{bmatrix} M_{00} & M_{01} & M_{02} & M_{03} \\ M_{10} & M_{11} & M_{12} & M_{13} \\ M_{20} & M_{21} & M_{22} & M_{23} \\ M_{30} & M_{31} & M_{32} & M_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} M_{0x} \cdot u \\ M_{1x} \cdot u \\ M_{2x} \cdot u \\ M_{3x} \cdot u \end{bmatrix}$$

10.2.4 Exercises

Matrix Operations

1. Compute:

(a)

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 1 & -1 & 2 \\ 0 & 3 & -5 \end{bmatrix}$$

(b)

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} - \begin{bmatrix} 1 & -1 & 2 \\ 0 & 3 & -5 \end{bmatrix}$$

(c)

$$-2 \begin{bmatrix} 1 & 7 \\ 2 & -3 \\ 0 & -1 \end{bmatrix}$$

(d)

$$2 \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} - 3 \begin{bmatrix} 1 & -1 & 2 \\ 0 & 3 & -5 \end{bmatrix}$$

(e)

$$3 \begin{bmatrix} 2 & -5 & 1 \\ 3 & 0 & -4 \end{bmatrix} - 2 \begin{bmatrix} 1 & -2 & -3 \\ 0 & -1 & 5 \end{bmatrix} + 4 \begin{bmatrix} 0 & 1 & -2 \\ 1 & -1 & -1 \end{bmatrix}$$

Matrix Multiplication

1. Compute:

(a)

$$\begin{bmatrix} 1 & 6 \\ -3 & 5 \end{bmatrix} \times \begin{bmatrix} 4 & 0 \\ 2 & -1 \end{bmatrix}$$

(b)

$$\begin{bmatrix} 1 & 6 \\ -3 & 5 \end{bmatrix} \times \begin{bmatrix} 2 \\ -7 \end{bmatrix}$$

(c)

$$\begin{bmatrix} 1 \\ -6 \end{bmatrix} \times \begin{bmatrix} 1 & 6 \\ -3 & 5 \end{bmatrix}$$

(d)

$$\begin{bmatrix} 1 \\ 6 \end{bmatrix} \times [3 \ 2]$$

(e)

$$[2 \ -1] \times \begin{bmatrix} 1 \\ -6 \end{bmatrix}$$

10.3 Transformation Matrices

There are a number of standard transformation matrices we can generate to manipulate our geometry. We will cover the most common here, and how they are generated.

10.3.1 Translation

Translation is when we wish to move our object in 3D space. We do this by applying a translation vector (t_x, t_y, t_z) in our matrix. A translation matrix is defined as follows:

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

So let us consider translating the point $(25, 14, 10)$ by $(10, 20, 30)$. We will expect that our translated point will be $(35, 34, 40)$ (we simply add the components together). Our translation matrix is therefore:

$$\begin{bmatrix} 1 & 0 & 0 & 10 \\ 0 & 1 & 0 & 20 \\ 0 & 0 & 1 & 30 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We are translating a point and we are working with a 4×4 matrix. Therefore, our point is represented as:

$$\begin{bmatrix} 25 \\ 14 \\ 10 \\ 1 \end{bmatrix}$$

We set the w component to 1 as we are dealing with a point. We then multiply the points representation by the transformation matrix:

$$\begin{bmatrix} 1 & 0 & 0 & 10 \\ 0 & 1 & 0 & 20 \\ 0 & 0 & 1 & 30 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 25 \\ 14 \\ 10 \\ 1 \end{bmatrix} = \begin{bmatrix} (1 \times 25) + (0 \times 14) + (0 \times 10) + (10 \times 1) \\ (0 \times 25) + (1 \times 14) + (0 \times 10) + (20 \times 1) \\ (0 \times 25) + (0 \times 14) + (1 \times 10) + (30 \times 1) \\ (0 \times 25) + (0 \times 14) + (0 \times 10) + (1 \times 1) \end{bmatrix} = \begin{bmatrix} 25 + 10 \\ 14 + 20 \\ 10 + 30 \\ 1 \end{bmatrix} = \begin{bmatrix} 35 \\ 34 \\ 40 \\ 1 \end{bmatrix}$$

We drop the one to get our translated 3D vector.

10.3.2 Rotation

Rotation comes in three forms, based on the axis that we are rotating around. In 3D, we can rotate around the X-axis (pitch), Y-axis (yaw) or Z-axis (roll). Each of these rotation types has a different matrix associated with them, and we will look at each of these in turn.

Pitch - Rotation on the X-Axis

The matrix for applying a rotation around the X-axis is:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta \\ 0 & \sin \theta & \cos \theta \end{bmatrix}$$

Where θ is the angle to rotate around the X-axis.

For example, let us take the vector $[0 \ 1 \ 0]$ and rotate it around the X-axis. Currently the vector points straight up, and we will rotate it by $\frac{\pi}{2}$ radians counter-clockwise (how we measure rotations). We would expect the vector to change from facing up to facing towards the viewer. First, let us generate the matrix:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \frac{\pi}{2} & -\sin \frac{\pi}{2} \\ 0 & \sin \frac{\pi}{2} & \cos \frac{\pi}{2} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix}$$

Our vector is represented as:

$$\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

We multiply the vector by the transformation matrix to get:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} (1 \times 0) + (0 \times 1) + (0 \times 0) \\ (0 \times 0) + (0 \times 1) + (-1 \times 0) \\ (0 \times 0) + (1 \times 1) + (0 \times 0) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Which is a vector pointing down the Z-axis.

Yaw - Rotation on the Y-Axis

The matrix for applying rotation around the Y-axis is:

$$\begin{bmatrix} \cos \theta & 0 & \sin \theta \\ 0 & 1 & 0 \\ -\sin \theta & 0 & \cos \theta \end{bmatrix}$$

Let us take our result from rotating around the X-axis, and rotate it by $\frac{\pi}{2}$ radians around the Y-axis. We will expect this vector to point to the right (down the positive X-axis). Our transformation matrix is:

$$\begin{bmatrix} \cos \frac{\pi}{2} & 0 & \sin \frac{\pi}{2} \\ 0 & 1 & 0 \\ -\sin \frac{\pi}{2} & 0 & \cos \frac{\pi}{2} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix}$$

Transforming our vector pointing down the Z-axis gives us:

$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} (0 \times 0) + (0 \times 0) + (1 \times 1) \\ (0 \times 0) + (1 \times 0) + (0 \times 1) \\ (-1 \times 0) + (0 \times 0) + (0 \times 1) \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

Roll - Rotation on the Z-Axis

The matrix for applying rotation around the Z-axis is:

$$\begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Let us now rotate our vector pointing down the X-axis by $\frac{\pi}{2}$ on the Z-axis. This should return our vector to pointing straight up. The transformation matrix is:

$$\begin{bmatrix} \cos \frac{\pi}{2} & -\sin \frac{\pi}{2} & 0 \\ \sin \frac{\pi}{2} & \cos \frac{\pi}{2} & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Transforming our vector then gives us:

$$\begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} (0 \times 1) + (-1 \times 0) + (0 \times 0) \\ (1 \times 1) + (0 \times 0) + (0 \times 0) \\ (1 \times 0) + (0 \times 0) + (1 \times 0) \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

which is our starting vector.

10.3.3 Scale

Scale allows us to change the size of a piece of geometry on one of its three axes. Therefore, scale can also be represented as a 3D vector $[S_x \ S_y \ S_z]$. The scale matrix is defined as:

$$\begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & S_z \end{bmatrix}$$

For example, if we consider a 3D vector $(1, -1, 0)$ and we wish to scale it by $(5, 10, 20)$ we would expect a result of $(5, -10, 0)$. The transformation matrix would be:

$$\begin{bmatrix} 5 & 0 & 0 \\ 0 & 10 & 0 \\ 0 & 0 & 20 \end{bmatrix}$$

The transformation calculation then becomes:

$$\begin{bmatrix} 5 & 0 & 0 \\ 0 & 10 & 0 \\ 0 & 0 & 20 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \\ 0 \end{bmatrix} = \begin{bmatrix} (5 \times 1) + (0 \times -1) + (0 \times 0) \\ (0 \times 1) + (10 \times -1) + (0 \times 0) \\ (0 \times 1) + (0 \times -1) + (20 \times 0) \end{bmatrix} = \begin{bmatrix} 5 \\ -10 \\ 0 \end{bmatrix}$$

Which is the expected answer.

10.3.4 Combining Transformations

To combine our matrices together, all we need to do is multiply them together. So far, we have built 3×3 and 4×4 matrices. As we know from matrix multiplication, these sizes of matrices cannot be multiplied. So what do we do?

For the 3×3 matrices, we need to convert them to 4×4 matrices. This is done by adding a row and a column to the matrix, with the added values being 0, except the bottom rightmost, which is 1. For example, the scale matrix can be defined in a 4×4 as:

$$\begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We do need to concern ourselves with the order that we do this, as matrix multiplication is not commutative. The order we wish to multiply the matrices together is:

$$T \times (R \times S)$$

That is *translation* \times (*rotation* \times *scale*). Doing the calculation in a different order will result in a different (and incorrect) output.

For example, let us consider a transformation defined as follows:

- X-axis rotation - $\frac{\pi}{2}$
- Z-axis rotation - π
- Scale - (10, 5, 20)
- Translation - (100, 20, 50)

Our scale 4×4 matrix is:

$$\begin{bmatrix} 10 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 \\ 0 & 0 & 20 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We get the X-axis rotation matrix:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix}$$

and the Z-axis rotation matrix:

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The convention for multiplying rotation matrices together in 3D graphics (where we use Euler angles) is $R_z \times R_x \times R_y$. Therefore, our combined rotation matrix is:

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

We then convert this into a 4×4 matrix:

$$\begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

If we calculate our $(R \times S)$ matrix, we get:

$$\begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 10 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 \\ 0 & 0 & 20 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} -10 & 0 & 0 & 0 \\ 0 & 0 & 20 & 0 \\ 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Our translation matrix is:

$$\begin{bmatrix} 1 & 0 & 0 & 100 \\ 0 & 1 & 0 & 20 \\ 0 & 0 & 1 & 50 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Multiplying our combined rotation and scale matrix provides the following:

$$\begin{bmatrix} 1 & 0 & 0 & 100 \\ 0 & 1 & 0 & 20 \\ 0 & 0 & 1 & 50 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -10 & 0 & 0 & 0 \\ 0 & 0 & 5 & 0 \\ 0 & 20 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} -10 & 0 & 0 & 100 \\ 0 & 0 & 5 & 20 \\ 0 & 20 & 0 & 50 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Notice that the translation matrix essentially replaces the last column in the rotation-scale matrix. We could view a combined transform matrix as:

$$\begin{bmatrix} & & T_x \\ R \times S & & T_y \\ & & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We can then use the combined transformation matrix to transform a vector. For example, if we take the point $(10, 20, 30)$, we get the following:

$$\begin{bmatrix} -10 & 0 & 0 & 100 \\ 0 & 0 & 5 & 20 \\ 0 & 20 & 0 & 50 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 10 \\ 20 \\ 30 \\ 1 \end{bmatrix} = \begin{bmatrix} (-10 \times 10) + (0 \times 20) + (0 \times 30) + (100 \times 1) \\ (0 \times 10) + (0 \times 20) + (5 \times 30) + (20 \times 1) \\ (0 \times 10) + (20 \times 20) + (0 \times 30) + (50 \times 1) \\ (0 \times 10) + (0 \times 20) + (0 \times 30) + (1 \times 1) \end{bmatrix} = \begin{bmatrix} 0 \\ 170 \\ 450 \\ 1 \end{bmatrix}$$

We will be using a library called GLM to do most of our mathematical work. This is what the next lesson is about.

10.3.5 Exercises

1. Generate the 4×4 transformation matrix for the following transformations:
 - (a) A translation of $< 4, 6, 7 >$
 - (b) A scaling of $< 10, 5, 2 >$
 - (c) A z-axis rotation of $\frac{\pi}{2}$ radians
 - (d) A x-axis rotation of π radians

- (e) A y-axis rotation of $\frac{\pi}{4}$ radians
2. Transform the following vectors using the transformation matrix

$$\begin{bmatrix} 2.121 & -2.121 & 0 & 10 \\ 2.121 & 2.121 & 0 & 15 \\ 0 & 0 & 3 & 12 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- (a) $< 4, 5, 10 >$
- (b) $< 12, 8, 16 >$
- (c) $< 1, 4, 0 >$
- (d) $< -10, 12, -4 >$
- (e) $< 0, 5, -11 >$

10.4 Answers

10.4.1 Vectors

Working with Vectors

1. Let $\mathbf{u} = < 2, -7, 1 >$, $\mathbf{v} = < -3, 0, 4 >$ and $\mathbf{w} = < 0, 8, -8 >$, Find:
- (a) $< -1, -7, 5 >$
 (b) $< -3, -8, 12 >$
 (c) $< -6, 21, -3 >$
 (d) $< 18, -21, -13 >$
 (e) $< -5, -54, 54 >$
2. Let $\mathbf{u} = < 2, -7, 1 >$, $\mathbf{v} = < -3, 0, 4 >$ and $\mathbf{w} = < 0, 8, -8 >$, Find:
- (a) -2
 (b) -64
 (c) $< 40, 16, 16 >$
 (d) $< -32, -24, -24 >$
 (e) 3.780

Normalising Vectors

1. Calculate the length of the following vectors:
- (a) 5
 (b) 22.913

(c) 7.280

(d) 3.464

(e) 33.287

2. What is the unit length (normalised) form of the following vectors:

(a) $\langle 0.6, 0.8, 0 \rangle$

(b) $\langle 0.436, 0.873, 0.218 \rangle$

(c) $\langle 0.137, 0.824, -0.549 \rangle$

(d) $\langle 0.577, 0.577, 0.577 \rangle$

(e) $\langle 0.240, 0.361, -0.901 \rangle$

Dot Product

1. Given the vectors $\mathbf{u} = \langle 2, -7, 1 \rangle$, $\mathbf{v} = \langle -3, 0, 4 \rangle$ and $\mathbf{w} = \langle 0, 8, -8 \rangle$, find:

(a) -2

(b) -2

(c) -32

(d) -32

(e) -64

Angle between Vectors

1. What is the dot product of the following pairs of vectors:

(a) $\frac{\pi}{2}$ rads

(b) $\frac{\pi}{2}$ rads

(c) $\frac{\pi}{3}$ rads

(d) 0.615rads

(e) 0rads

Projection Vectors

1. Calculate the projection of a onto b given:

(a) $\langle 0.24, 0, -0.32 \rangle$

(b) $\langle -0.074, 0.259, -0.037 \rangle$

(c) $\langle 0, -4, 4 \rangle$

(d) $\langle -2.370, 8.296, -1.185 \rangle$

(e) $\langle 0, -2, 2 \rangle$

Cross Product

1. Given the vectors $\mathbf{u} = \langle 2, -7, 1 \rangle$, $\mathbf{v} = \langle -3, 0, 4 \rangle$ and $\mathbf{w} = \langle 0, 8, -8 \rangle$, find:
 - (a) $\langle -28, -11, -21 \rangle$
 - (b) $\langle 28, 11, 21 \rangle$
 - (c) $\langle 0, 24, 0 \rangle$
 - (d) $\langle 0, -24, 0 \rangle$
 - (e) $\langle -56, -16, 0 \rangle$
2. Calculate the following:
 - (a) $\langle 0, 0, 1 \rangle$
 - (b) $\langle 0, 0, -1 \rangle$
 - (c) $\langle 0, -1, 0 \rangle$
 - (d) $\langle 0, 1, 0 \rangle$
 - (e) $\langle 0, 0, 0 \rangle$

10.4.2 Matrices

Matrix Operations

1. Compute:
 - (a)
 - (b)
 - (c)
 - (d)
 - (e)
- (a)
- $$\begin{bmatrix} 2 & 1 & 5 \\ 4 & 8 & 1 \end{bmatrix}$$
- (b)
- $$\begin{bmatrix} 0 & 3 & 1 \\ 4 & 2 & 11 \end{bmatrix}$$
- (c)
- $$\begin{bmatrix} -2 & -14 \\ -4 & 6 \\ 0 & 2 \end{bmatrix}$$
- (d)
- $$\begin{bmatrix} -1 & 7 & 0 \\ 8 & 1 & 37 \end{bmatrix}$$
- (e)
- $$\begin{bmatrix} 4 & -7 & 1 \\ 4 & -2 & -26 \end{bmatrix}$$

Matrix Multiplication

1. Compute:

(a)

$$\begin{bmatrix} 16 & -6 \\ -2 & -5 \end{bmatrix}$$

(b)

$$\begin{bmatrix} -40 \\ -41 \end{bmatrix}$$

(c) Not defined

(d)

$$\begin{bmatrix} 3 & 2 \\ 18 & 12 \end{bmatrix}$$

(e)

$$[8]$$

10.4.3 Transformation Matrices

1. Generate the 4×4 transformation matrix for the following transformations:

(a)

$$\begin{bmatrix} 1 & 0 & 0 & 4 \\ 0 & 1 & 0 & 6 \\ 0 & 0 & 0 & 7 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

(b)

$$\begin{bmatrix} 10 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

(c)

$$\begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

(d)

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

(e)

$$\begin{bmatrix} 0.707 & 0 & 0.707 & 0 \\ 0 & 1 & 0 & 0 \\ -0.707 & 0 & 0.707 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2. Transform the following positional vectors using the transformation matrix

$$\begin{bmatrix} 2.121 & -2.121 & 0 & 10 \\ 2.121 & 2.121 & 0 & 15 \\ 0 & 0 & 3 & 12 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- (a) $< 7.879, 34.089, 42 >$
- (b) $< 18.484, 57.420, 60 >$
- (c) $< 3.637, 25.605, 12 >$
- (d) $< -36.662, 19.242, 0 >$
- (e) $< -0.605, 25.605, -21 >$

Lesson 11

Quaternions

10 to 15 years ago working only with matrix transformations would have stood you in good stead when working with computer graphics. We are now going to do some work with geometry, and look at using a collection of geometric primitives to allow us to build more complicated scenes. However, first of all we are going to look at a better method for handling our rotations –using quaternions.

11.1 Why use Quaternions?

Quaternions have a number of advantages over using matrix based transformation rotations when it comes to working with 3D graphics. The main ones are:

- They utilise less space than a matrix when storing a rotation. However, this could be considered a lesser argument these days, and also typically we convert the quaternion into a matrix for use in our transformations anyway.
- Fewer arithmetic operations when concatenating (multiplying together) quaternions as opposed to transformation matrices. This can be very important in rendering applications.
- Quaternions can provide smoother animation.

11.2 What are Quaternions?

A quaternion can be thought of as a four dimensional vector, defined as follows:

$$\mathbf{q} = \langle w, x, y, z \rangle = w + xi + yj + zk$$

If you are unfamiliar with the second notation, then let us take a look at the definition. Although you are probably used to seeing a three-dimensional vector defined as a collection of three values as follows:

$$[x, y, z]$$

We can also define a vector as the addition of the scaled unit vectors. Here we have the following

definitions:

$$\begin{aligned} i &= [1, 0, 0] \\ j &= [0, 1, 0] \\ k &= [0, 0, 1] \end{aligned}$$

This means that the following two definitions are equal:

$$[5, 7, 8] = 5i + 7j + 8k$$

Sometimes you may see a quaternion defined as follows:

$$\mathbf{q} = s + \mathbf{v}$$

Here, \mathbf{v} is the three-dimensional vector part, and s is a scalar representing the w part of the quaternion definition.

11.3 Using Quaternions to Represent Rotations

There are a number of other properties of quaternions that we could go into. However, for our purposes we only need concern ourselves with using them for rotations. We will use quaternions to store cumulated rotations - that is we will be adding rotations together to create a final rotation. This is a property of rotations proved by Euclid, which basically means that any combination of rotations can be represented as one rotation around an arbitrary axis. This is what we will be using quaternions for.

Remember that our definition of a quaternion is thus:

$$\mathbf{q} = \langle w, x, y, z \rangle = w + xi + yj + zk$$

The $\langle x, y, z \rangle$ part is the axis that the rotation is being performed around (this will be a unit vector - one unit in length). All we need to do now is determine how to represent the angle. A quaternion representing a rotation, θ around an arbitrary axis $\langle x, y, z \rangle$ is defined as follows:

$$\mathbf{q} = [\cos \frac{\theta}{2}, x(\sin \frac{\theta}{2}), y(\sin \frac{\theta}{2}), z(\sin \frac{\theta}{2})]$$

For example, if we wish to rotate an object π around the x-axis, we would build a quaternion as follows:

$$\begin{aligned} \mathbf{q} &= [\cos \frac{\pi}{2}, 1(\sin \frac{\pi}{2}), 0(\sin \frac{\pi}{2}), 0(\sin \frac{\pi}{2})] \\ \mathbf{q} &= [0, 1, 0, 0] \end{aligned}$$

If we wanted to rotate an object $\frac{\pi}{2}$ around an axis pointing down the y-axis and z-axis, we would get:

$$\begin{aligned} \mathbf{q} &= [\cos \frac{\pi}{4}, 0(\sin \frac{\pi}{4}), \sqrt{2}(\sin \frac{\pi}{4}), \sqrt{2}(\sin \frac{\pi}{4})] \\ \mathbf{q} &= [\frac{1}{\sqrt{2}}, 0, \frac{1}{2}, \frac{1}{2}] \end{aligned}$$

11.4 Combining Rotations

So we now know how to represent a rotation using a quaternion. However, we also need to consider how we can combine the quaternions together to form rotations from a collection of rotations. We do this by multiplying quaternions.

11.4.1 Quaternion Multiplication

The cross product operation of a quaternion is calculated as follows:

$$\begin{aligned}\mathbf{q}_1 &= [w_1, x_1, y_1, z_1] = [w_1, \mathbf{v}_1] \\ \mathbf{q}_2 &= [w_2, x_2, y_2, z_2] = [w_2, \mathbf{v}_2] \\ \mathbf{q}_1 \times \mathbf{q}_2 &= \begin{bmatrix} w_1 w_2 - x_1 x_2 - y_1 y_2 - z_1 z_2 \\ w_1 x_2 + w_2 x_1 + y_1 z_2 - y_2 z_1 \\ w_1 y_2 + w_2 y_1 + z_1 x_2 - z_2 x_1 \\ w_1 z_2 + w_2 z_1 + x_1 y_2 - x_2 y_1 \end{bmatrix}\end{aligned}$$

Let us test this. Consider that we want to rotate an object $\frac{\pi}{2}$ around the y-axis, and $\frac{\pi}{2}$ around the z-axis. This will give us the following:

$$\begin{aligned}\mathbf{q}_1 &= [\cos \frac{\pi}{4}, 0(\sin \frac{\pi}{4}), 1(\sin \frac{\pi}{4}), 0(\sin \frac{\pi}{4})] = [\frac{1}{\sqrt{2}}, 0, \frac{1}{\sqrt{2}}, 0] \\ \mathbf{q}_2 &= [\cos \frac{\pi}{4}, 0(\sin \frac{\pi}{4}), 0(\sin \frac{\pi}{4}), 1(\sin \frac{\pi}{4})] = [\frac{1}{\sqrt{2}}, 0, 0, \frac{1}{\sqrt{2}}] \\ \mathbf{q}_1 \times \mathbf{q}_2 &= \begin{bmatrix} \frac{1}{\sqrt{2}} \times \frac{1}{\sqrt{2}} - 0 \times 0 - \frac{1}{\sqrt{2}} \times 0 - 0 \times \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \times 0 + \frac{1}{\sqrt{2}} \times 0 + \frac{1}{\sqrt{2}} \times \frac{1}{\sqrt{2}} - 0 \times 0 \\ \frac{1}{\sqrt{2}} \times 0 + \frac{1}{\sqrt{2}} \times \frac{1}{\sqrt{2}} + 0 \times 0 - \frac{1}{\sqrt{2}} \times 0 \\ \frac{1}{\sqrt{2}} \times \frac{1}{\sqrt{2}} + \frac{1}{\sqrt{2}} \times 0 + 0 \times 0 - 0 \times \frac{1}{\sqrt{2}} \end{bmatrix} = [\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}]\end{aligned}$$

The equation can also be written in a simpler form:

$$\begin{aligned}\mathbf{q}_1 &= [w_1, x_1, y_1, z_1] = [w_1, \mathbf{v}_1] \\ \mathbf{q}_2 &= [w_2, x_2, y_2, z_2] = [w_2, \mathbf{v}_2] \\ \mathbf{q}_1 \times \mathbf{q}_2 &= [w_1 w_2 - \mathbf{v}_1 \cdot \mathbf{v}_2, w_1 \mathbf{v}_2 + w_2 \mathbf{v}_1 + \mathbf{v}_1 \times \mathbf{v}_2]\end{aligned}$$

Using this calculation for our previous example, we get:

$$\begin{aligned}\mathbf{q}_1 &= [\cos \frac{\pi}{4}, 0(\sin \frac{\pi}{4}), 1(\sin \frac{\pi}{4}), 0(\sin \frac{\pi}{4})] = [\frac{1}{\sqrt{2}}, 0, \frac{1}{\sqrt{2}}, 0] \\ \mathbf{q}_2 &= [\cos \frac{\pi}{4}, 0(\sin \frac{\pi}{4}), 0(\sin \frac{\pi}{4}), 1(\sin \frac{\pi}{4})] = [\frac{1}{\sqrt{2}}, 0, 0, \frac{1}{\sqrt{2}}] \\ \mathbf{q}_1 \times \mathbf{q}_2 &= [\frac{1}{\sqrt{2}} \times \frac{1}{\sqrt{2}} - 0, (0, 0, \frac{1}{2}) + (0, \frac{1}{2}, 0) + (0, \frac{1}{\sqrt{2}}, 0) \times (0, 0, \frac{1}{\sqrt{2}})] \\ \mathbf{q}_1 \times \mathbf{q}_2 &= [\frac{1}{2}, (0, 0, \frac{1}{2}) + (0, \frac{1}{2}, 0) + (\frac{1}{2}, 0, 0)] \\ \mathbf{q}_1 \times \mathbf{q}_2 &= [\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}]\end{aligned}$$

This is the same result, but with fewer multiplication operations (and hence faster on the CPU).

11.5 Converting a Quaternion to a Rotation Matrix

Once we have a quaternion representing a rotation, we need a method to convert the quaternion into a matrix so that we can use it in our transformations. This is done using the following equation:

$$\mathbf{M}_r = \begin{bmatrix} 1 - 2(y^2 + z^2) & 2(xy - wz) & 2(xz + wy) & 0 \\ 2(xy + wz) & 1 - 2(x^2 + z^2) & 2(yz - wz) & 0 \\ 2(xz - wy) & 2(yz + wx) & 1 - 2(x^2 + y^2) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Using our example from before, we would create a rotation matrix defined as follows:

$$\mathbf{M}_r = \begin{bmatrix} 1 - 2(\frac{1}{4} + \frac{1}{4}) & 2(\frac{1}{4} - \frac{1}{4}) & 2(\frac{1}{4} + \frac{1}{4}) & 0 \\ 2(\frac{1}{4} + \frac{1}{4}) & 1 - 2(\frac{1}{4} + \frac{1}{4}) & 2(\frac{1}{4} - \frac{1}{4}) & 0 \\ 2(\frac{1}{4} - \frac{1}{4}) & 2(\frac{1}{4} + \frac{1}{4}) & 1 - 2(\frac{1}{4} + \frac{1}{4}) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

If we create the two rotation matrices we would normally develop using standard transformations, we get the following:

$$\mathbf{M}_r = \mathbf{M}_1 \times \mathbf{M}_2 = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Which is the same answer as above (but with fewer machine operations).

11.6 GLM to the Rescue

Knowing about quaternions is one thing, but using them in our code is another. Thankfully, we do not have to worry ourselves with the inner workings of quaternions as GLM (the mathematics library we use) has quaternion functionality built in for us. We will simply exploit this in our code in the future. It is handy to know what a quaternion is, and roughly how they operate. If you want to know more (including the theoretical underpinnings) then read Section 4.6 of Mathematics for 3D Game Programming and Computer Graphics [2].

Lesson 12

Working with GLM

In this lesson you are going to have to do some work with GLM. We will look at the different methods we require as used in the previous two lessons, and then it is up to you to write the code specified and check the values in the debugger to check and understand what you have been doing.

Most of the functionality we use in this lesson comes from the main GLM header file - `GLM\glm.hpp`.

12.1 Vectors

First off, we will look at vectors. GLM provides a number of vector types to suit different uses. Let us look first at how we define some vector types, and then the operations we can perform.

12.1.1 Defining Vectors

Declaring a vector type in GLM is easy. There are three types - 2D, 3D and 4D vectors. We declare them in the following manner.

```
1 vec2 u(x, y, z);  
2 vec3 v(x, y, z);  
3 vec4 w(x, y, z, w); // or RGBA for colours
```

Exercise Add six new variables to the main application for this lesson. You want two of each vector type. You can use whatever types you want, but it is advisable to use vectors that point down particular axis (e.g. (1.0, 0.0, 0.0)).

It is also possible to convert between the different vector types. We can either remove components from a vector as follows:

```
1 vec2 u(vec3(x, y, z)); // z component is dropped  
2 vec2 v(vec4(x, y, z, w)); // z and w components dropped  
3 vec3 w(vec4(x, y, z, w)); // w component dropped
```

Or we can add a component to the vector as follows:

```

1 vec3 u(vec2(x, y), z);
2 vec4 v(vec2(x, y), z, w);
3 vec4 w(vec3(x, y, z), w);

```

Exercise Create another six variables. Convert one of the 2D vectors to a 3D and one to a 4D. Then convert one of the 3D vectors to a 2D and one to a 4D. Finally, convert one of the 4D vectors to a 2D and one to a 3D.

You can also access individual components of a vector, either getting them or setting them. An example is shown below.

```

1 float x = u.x;
2 u.y = 10.0f;
3 // Can use x, y, z, w or r, g, b, a.

```

12.1.2 Vector Addition and Subtractions

Adding and subtracting vectors is easy in GLM as the basic vector types have these operators overloaded. An example is shown below.

```

1 vec3 w = u + v;
2 // Same for other vector types
3 // Subtraction is just a change of symbol

```

Exercise Again, you want to create six more variables. Add two of your 2D vectors together storing the result, and subtract one from another again storing the result. Do the same for 3D and 4D vectors.

12.1.3 Vector Scaling

Multiplication operators are also overloaded in GLM for vectors (multiplying two vectors however is component-wise –handy for colours later but not useful here). As such, we can scale our vectors by multiplying them by floating point numbers. We can also divide vectors by floating point numbers. Some examples are shown below.

```

1 vec3 v = 5.0f * u;
2 vec3 w = u / 5.0f;
3 // Same for other vector types

```

Exercise Six more vector variables need to be added to your application again. This time, multiply a 2D vector by a scalar value (float) storing the result, and the same for division. Do the same for 3D and 4D vectors.

12.1.4 Length of a Vector

We now move onto some of the functions provided by GLM. The first of which is the `length` function. This is used to get the length of a vector. An example is shown below.

```
1 float l = length(v);
```

Exercise Now create three float values and get the length of three of your defined vectors - a 2D one, a 3D one and a 4D one.

12.1.5 Normalizing a Vector

Normalizing a vector also involves using a defined function in GLM. The `normalize` function will return a normalized version of a vector. An example is below.

```
1 vec3 n = normalize(u);
2 // Same for other vector types
```

Exercise Create three variables and store the result of normalizing a 2D, 3D, and 4D vector.

12.1.6 Dot Product

The dot product of two vectors is also provided as a function. The use of the `dot` function is shown below.

```
1 float d = dot(u, v);
2 // Same for other vector types
```

Exercise Again you need three new float variables. Get the result of the dot product of a 2D, 3D, and 4D vector and store the results.

Vector Projection

As a little aside, we will look at how we can calculate the projection of one vector onto another. The `proj` function is declared in the header `glm\gtx\projection.hpp`.

```
1 vec3 p = proj(u, v);
2 // Same for other vector types
```

Exercise Perform three projection operations, storing the result. Project a 2D vector onto another 2D vector, and likewise for 3D and 4D vectors.

12.1.7 Cross Product

Calculating the cross product of two vectors is also just a function call. The use of the `cross` function is shown below.

```
1 vec3 c = cross(u, v);
```

Exercise The cross product of 3D vectors is the only interesting one in our work. Perform a cross product on two of the 3D vectors you have previously defined, storing the result.

12.2 Matrices

Let us now move onto matrices and see how we work with these using GLM.

12.2.1 Defining a Matrix

GLM provides a number of different matrix types - ranging from square matrices (`mat3`, `mat4`) to other types (`mat2x4`, etc.). In general, we will only be working with `mat4` types. To declare one we use the following:

```
1 mat4 m(1.0f); // Identity matrix
```

Note the use of a floating point number in the constructor. This sets the major diagonal values of the matrix. In this case we set them as 1 to create an identity matrix.

As with vectors, we can convert between matrix types by explicit construction. An example is shown below.

```
1 mat3 n(mat4(1.0f));
2 // Last column and row are dropped
```

Exercise Create a few matrices. You can take a look at the attributes of the matrix and see how you can access (get and set) individual components - just as vectors. Do this as well to get a feel for the individual types.

12.2.2 Matrix Addition

As with vectors, matrix types have overridden operators. We can add and subtract matrices, just as follows:

```
1 mat4 m = n + o;
```

Exercise Add and subtract some matrices, storing the results.

12.2.3 Matrix Scaling

Matrix scaling also works as vector scaling. An example is shown below.

```
1 mat4 m = 5.0f * n;
2 mat4 o = n / 5.0f;
```

Exercise Scale some of your defined matrices, storing the results.

12.2.4 Matrix Multiplication

The most important part of working with matrices is the ability to multiply them together. We can do this just using the multiply operation as shown below.

```
1 mat4 m = n * o;
```

Exercise Multiply some matrices together. Remember that only certain sizes of matrices can be multiplied together. Try and multiply some of the different types together and see what GLM can do.

Matrix–Vector Multiplication

One of the most important capabilities when working with matrices and vectors is the ability to multiply one by the other - for transformation purposes. An example of this is shown below.

```
1 mat4 T;
2 vec3 u;
3 // Can't multiply a 3D vector by a 4x4 matrix
4 // Have to convert to a 4D vector
5 vec4 v = T * vec4(u, 1.0f);
6 // We can always get the 3D version if explicit
7 vec3 w = vec3(T * vec4(u, 1.0f));
```

Exercise Now try and multiply some of your initial vectors using your matrices, storing the results in new vectors. We will do this some more as we start generating transformation matrices.

12.3 Transformations

We are now going to move on to creating transformation matrices. The functions defined below are declared in the `GLM\gtc\matrix_transform.hpp`. This is because transformation matrices are not part of GLSL (GL Shading Language), which GLM is meant to emulate. We will get to GLSL when we move onto shaders.

12.3.1 Translation Matrix

First we will create a translation matrix. The `translate` function is shown below:

```
1 mat4 T = translate(mat4(1.0f), vec3(x, y, z));
2 // We have to transform an initial matrix - use identity
```

Note that we need to provide a matrix to perform the translate upon. Typically, we will just use the identity matrix for this.

Exercise Create a translation matrix. Then transform one of your 3D vectors (using the technique shown to allow the multiplication), storing the result in another 3D vector. Check your result to ensure that you have done this correctly.

12.3.2 Rotation Matrix

Let us now look at how we create rotation transformation matrices. We can create a rotation matrix for any axis using the same technique - simply defining the axis we wish to rotate around. The following subsections show how we can create X, Y, and Z rotations.

X Rotation

```
1 mat4 Rx = rotate(mat4(1.0f), angle, vec3(1.0f, 0.0f, 0.0f));
```

Y Rotation

```
1 mat4 Ry = rotate(mat4(1.0f), angle, vec3(0.0f, 1.0f, 0.0f));
```

Z Rotation

```
1 mat4 Rz = rotate(mat4(1.0f), angle, vec3(0.0f, 0.0f, 1.0f));
```

Exercise Create three rotation matrices - one for each rotation type. Sensible rotations here are for quarter-rotations (e.g. $\frac{\pi}{2}$ radians). Then transform some 3D vectors - preferably ones pointing down an axis (e.g. $(1, 0, 0)$) so you can check the result. Finally, combine the rotations, by multiplying them together, then perform a transformation and again check the result.

Euler Angles

We can also create a rotation transformation matrix which uses all three axes in one operation by using the `eulerAngleYXZ` function. This is defined in the `glm\gtc\angle.hpp` header.

```
1 mat4 R = eulerAngleYXZ(yaw, pitch, roll);
2 // Other eulerAngle functions exist depending on available values
```

12.3.3 Scale Matrix

Our final transformation matrix type we are interested in is the scale transform. This is created using the `scale` function. An example is shown below.

```
1 mat4 S = scale(mat4(1.0f), vec3(x, y, z));
```

Exercise Create a scale matrix and then transform a 3D vector with this matrix.

12.3.4 Combining Matrices

As indicated, we can combine our transformation matrices together into one single matrix by multiplication. The following provides an example.

```
1 mat4 trans = T * (R * S);
```

Exercise Combine three of your transformation matrices together. Then transform a 3D vector by this combination, storing your result. You should be able to check that you have the right result.

12.4 Quaternions

The final data type we are interested in is the quaternion (`quat`) type. Quaternions are also not a part of GLSL, and therefore we need a header (`GLM\gtc\quaternion.hpp`) to use quaternions in our code.

12.4.1 Defining Quaternions

Creating a quaternion is simple - we just declare one as below.

```
1 quat q;
```

Exercise Create a quaternion in the project for this lesson.

12.4.2 Quaternions for Rotations

A more useful use of quaternions is creating a rotation. We do this using the `rotate` function, as shown below.

```
1 quat q = rotate(quat(), angle, vec3(x, y, z));
2 // The vec3 represents the axis to rotate around
```

We have to define the axis we are rotating around.

Exercise Create quaternions that represent rotations around the three main axes.

12.4.3 Quaternion Multiplication

Multiplying quaternions is also easy - we just use the multiplication operation:

```
1 quat rx = rotate(quat(), angle, vec3(1.0f, 0.0f, 0.0f));
2 quat ry = rotate(quat(), angle, vec3(0.0f, 1.0f, 0.0f));
3 quat rz = rotate(quat(), angle, vec3(0.0f, 0.0f, 1.0f));
4 quat R = rz * rx * ry;
```

Exercise Multiply your three quaternions together.

12.4.4 Conversion to a Matrix

Finally, we can convert a quaternion to a `mat4` by using the `mat4_cast` function. An example of this is shown below.

```
1 mat4 R = mat4_cast(q);
```

Exercise Convert your combined rotation quaternion to a matrix, and then use it to transform a 3D vector, storing the result.

Lesson 13

Rotating a Triangle

In this lesson we will apply a rotation to our triangle. We have already been applying a transformation of sorts to our geometry - what we call a model-view-projection transform. The part we are interested in at the moment is the model transform - this is the transform responsible for rotations, translations and scales.

13.1 Updating Lesson

In the starting code for this lesson you will see a section in update where you need to define the R matrix using a Z rotation transformation. From the previous lesson you know how to do this. The application itself will apply this to our shader which will perform the actual transformation. All you need to do is create R .

The output for this lesson is shown in Figure 13.1.

13.2 Exercise

Modify the programme for this lesson so that you rotate the triangle on the X axis. Then modify it to perform a rotation on the Y axis.

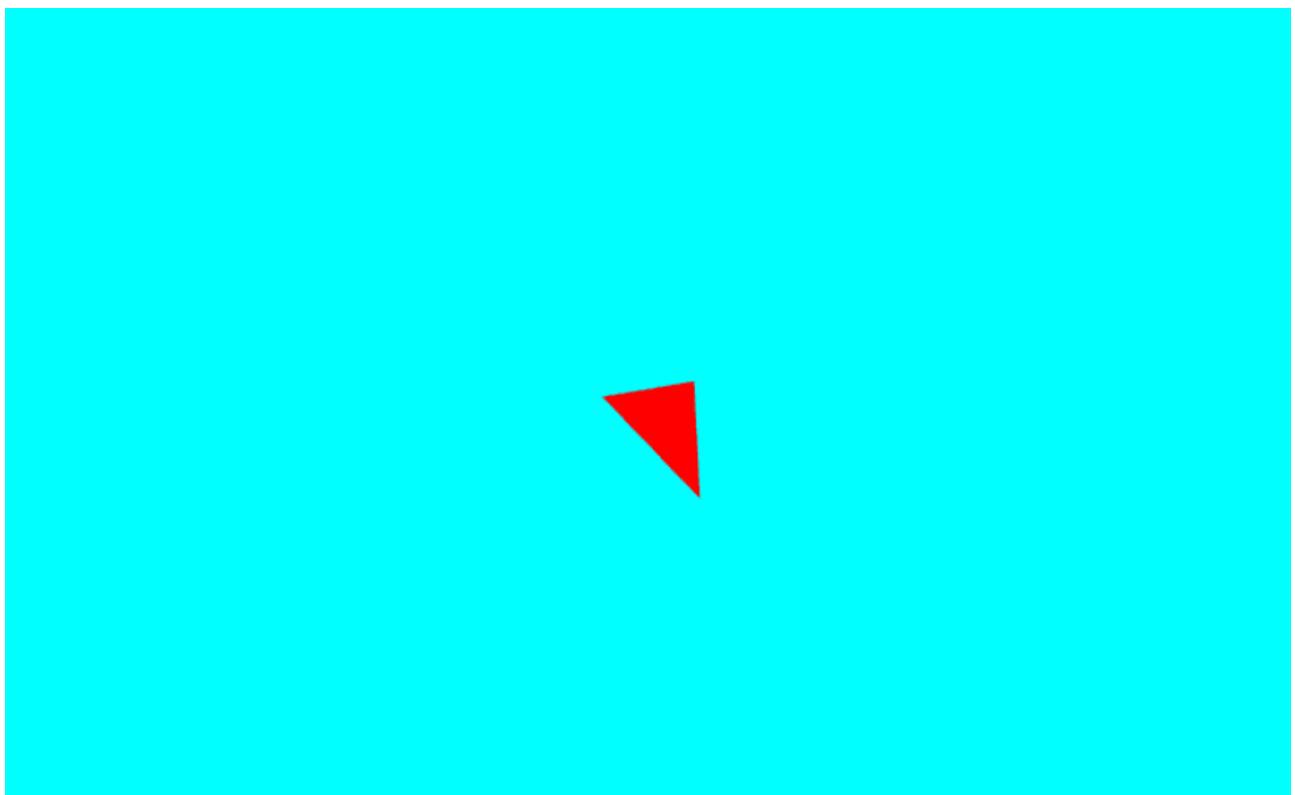


Figure 13.1: Rotating Triangle Output

Lesson 14

Scaling a Triangle

Now that we have seen how to undertake a basic transformation for rotation, let us perform the same action but this time implementing a scaling transformation. Again, the basic lesson structure has been provided, it is up to you to implement the defined change. This time, you have to define the `mat4` value S using the variable s to set a uniform scale on all axes.

When you run the application, you should get an output similar to Figure 14.1.

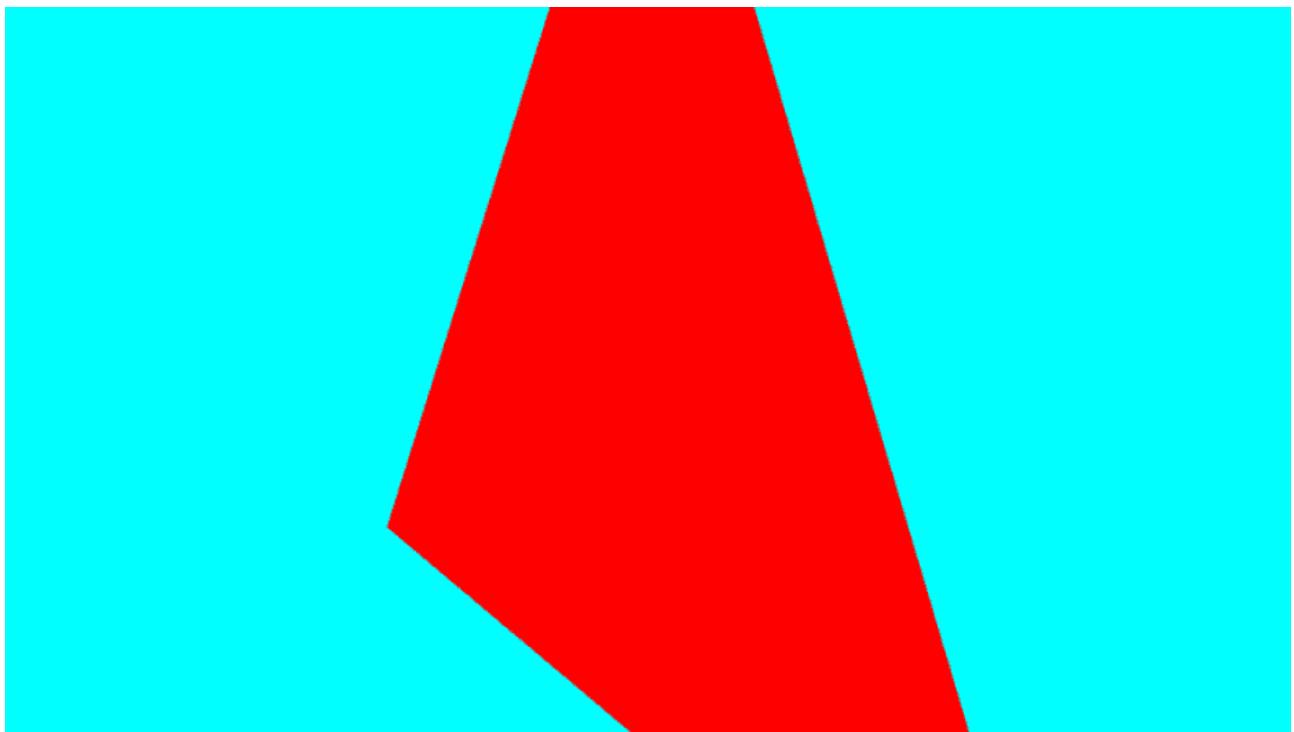


Figure 14.1: Scaling Triangle

14.1 Exercise

Take a look at the code in `update` which defines the scale value s . The code sets s based on a sine wave. This is quite a common technique for computer controlled animation. Being able to think in this manner is quite important for future work you will undertake.

Lesson 15

Scaling and Rotating a Triangle

Our next task is to combine the two transformations to create a triangle that both rotates and scales. We will combine the two transformations by multiplying them together. As always, the project has the necessary code for you to get started. All you have to do is ensure that you create the model transformation matrix accordingly.

The output from this lesson is shown in Figure 15.1.

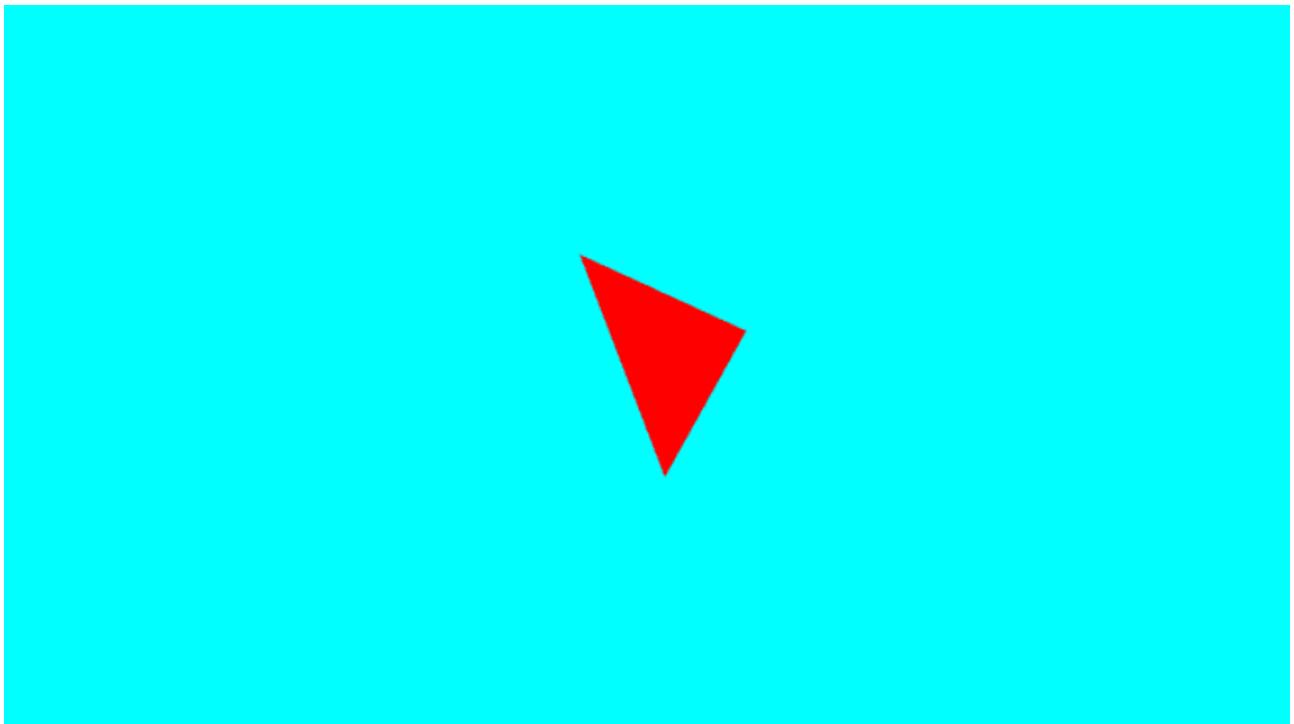


Figure 15.1: Scaling and Rotating Triangle

Lesson 16

Moving Quad

In this lesson your task is to use the translation function to create a translation matrix, T. Remember that the translation function in GLM is as follows:

```
1 mat4 T = translate(mat4(1.0f), v);
```

where v is the offset from the origin we want to move by.

The starting code for this project has a 3-dimensional vector value pos which is updated by the user interacting with the application using the cursor keys. Your task is to use the pos value to define the transformation matrix T.

When you run this application, use the cursor keys to move the quad around the scene. An example output from this application is shown in Figure 16.1.

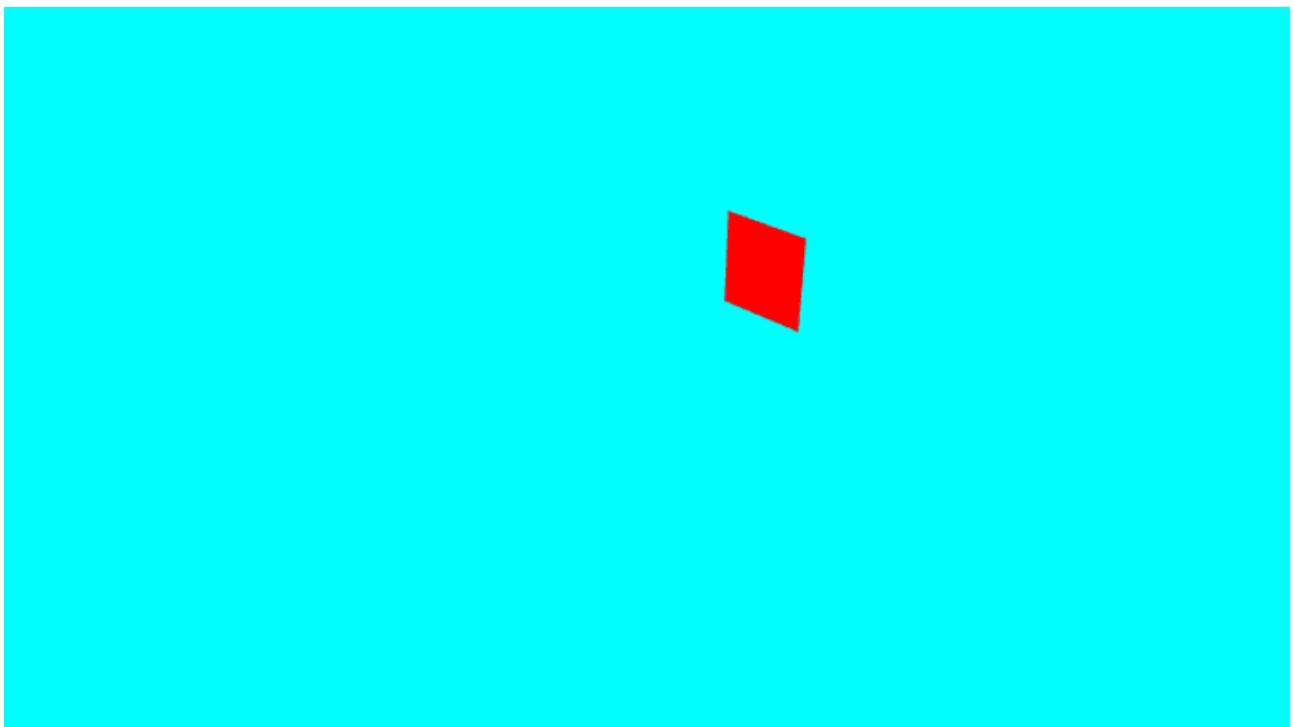


Figure 16.1: Output from Moving Quad Lesson

16.1 Exercise

Examine the code in `update` to see how keyboard input is handled in the render framework. This is useful code to understand. To see if you do, add the extra code necessary to allow translation on the Z axis. You can pick whichever keys you like to achieve this.

Lesson 17

Full Transformation

In this lesson we will combine the three transformation matrices together to create a full transformation for our object. You know how to create the individual transformation matrices, and now the same values exist from the previous few lessons. Your task is to create one single matrix M . Remember that we combine the matrices in the following manner:

$$M = T \times (R \times S)$$

The project code has the place where you need to perform this update. You have all the knowledge you need. An example output from this project can be seen in Figure 17.1.

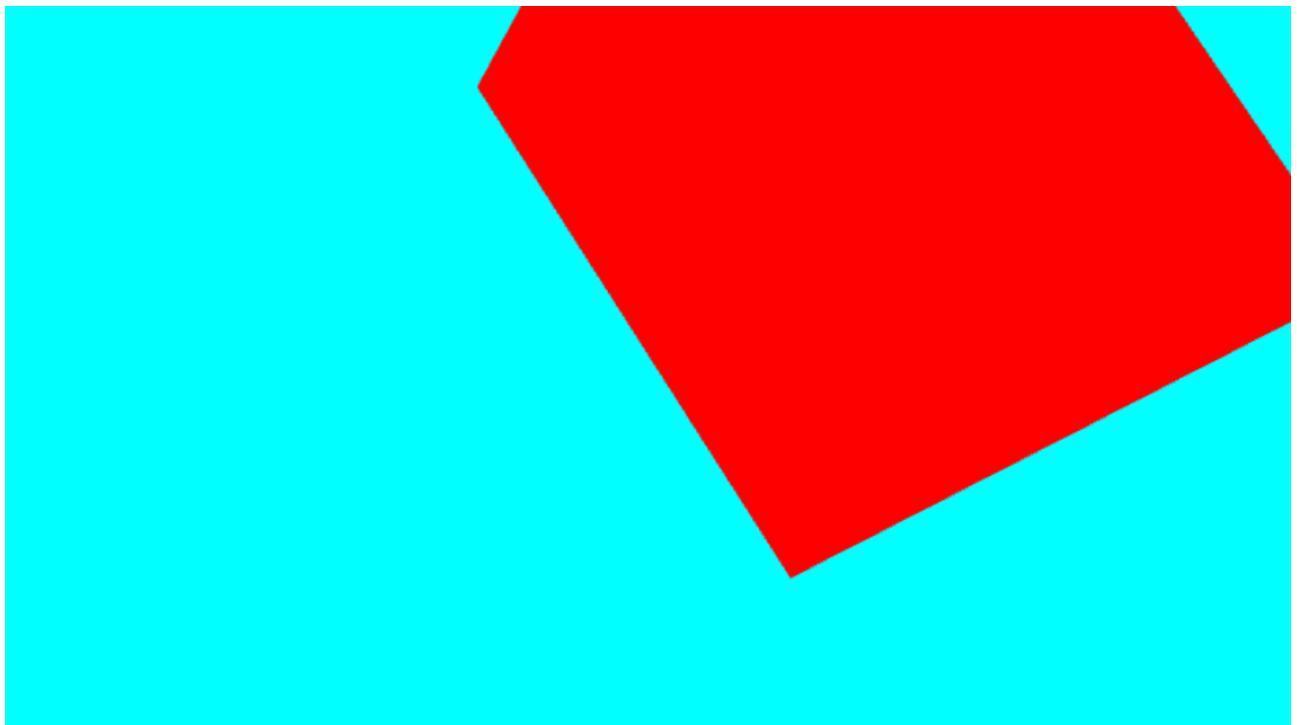


Figure 17.1: **Full Transformation on a Quad**

Lesson 18

Point Based Sierpinski Gasket

We have come quite a long way already in our understanding of fundamental graphics programming. Let us now try and create something a bit more exciting. This example is taken from the book Interactive Computer Graphics: A Top-Down Approach by Edward Angel and David Shreiner [3]. We have adapted it slightly to suit our style of development.

18.1 What is the Sierpinski Gasket?

The Sierpinski Gasket is a form of fractal. The Sierpinski Gasket can be defined recursively and randomly. Basically, the shape is defined by sub-dividing triangles as shown in Figure 18.1.

The first algorithm we are going to use for this is a simple random operation where we create an array of random points which will form our Gasket. Let us see how we do this.

18.2 Getting Started

The code for this lesson requires you to complete the `create_sierpinski` method defined in the main file. There is already some of the method defined. Your task will be to complete the method.

Firstly, we have defined two vectors at the start of the method to store our position and colour data:

```
1 vector<vec3> points;
2 vector<vec4> colours;
```

We have also defined three starting points - these are the three corners of the Sierpinski Gasket. These points are defined as follows:

```
1 array<vec3, 3> v =
2 {
3     vec3(-1.0f, -1.0f, 0.0f),
4     vec3(0.0f, 1.0f, 0.0f),
5     vec3(1.0f, -1.0f, 0.0f)
6 };
```

The next piece of code is about generating random numbers using C++. This is an important piece of code to familiarise yourself with.

18.2.1 Random Number Generation in C++

C++11 works on the principal of random number generation engines. There are a few different generation engines and schemes available depending on application requirements - you can read about them if you are interested. However, for most general purpose random number generation situations we can just use the default engine. This is defined as follows:

```
1 default_random_engine e;
```

We can then get a random number from this engine by making a call such as:

```
1 auto n = e();
```

However, it is often the case that we need to generate a random number within a defined range. To do this, C++11 also provides distribution objects. These allow us to define a range and a method of distribution on the random numbers generated by an engine. As before, if you are interested in the different techniques you can research these further.

For our current purposes we need to randomly select one of the points defined in the array `v`. Therefore, we need a random number in the range 0 to 2. We also want this number to be an `int`. To create such a distribution we use the following:

```
1 uniform_int_distribution<int> dist(0, 2);
```

This has all been set up for you in the `create_sierpinski` method. The important part you need to know is how to combine these two objects together to generate a random number between 0 and 2. This is done using the following piece of code:

```
1 auto n = dist(e);
```

This is an important piece of code that you will require to complete this challenge. The challenge itself relies on getting random numbers to randomly select one of the points in the array. We will look at the equation you need to implement shortly.

18.3 Generating Points

The `create_sierpinski` method generates a starting point and adds it to the `vector` of positions. This initial position is the centre of the gasket as defined by the three corners. An initial colour value is also added to the colour `vector`.

18.3.1 Challenge

The algorithm to generate the Gasket is as follows:

$$\text{point}(k) = (\text{point}(k - 1) + v(\text{rand}))/2.0$$

To get a random number, we use the random engine and distribution (between 0 and 2) we created. You will need to write a for loop that runs from 1 (not 0) to `num_points` (a constant value defined in the application). The for loop will add a new point based on the above algorithm. You will also need to add a colour value with every point.

Finally, you need to ensure that you are adding the buffers to the `geom` object passed into the `create_sierpinski` method. You should know how to do this by now as well. Everything else in the application has been taken care of.

The output from this application is shown in Figure 18.2.

18.4 Exercises

1. Try manipulating the number of points generated. Try 5000, 500000, etc. What is the result?
2. Try and also implement the transformation concepts we discussed in the last few lessons. Can you make the Gasket rotate, scale, transform, etc.? What does the result tell you about point rendering in OpenGL.

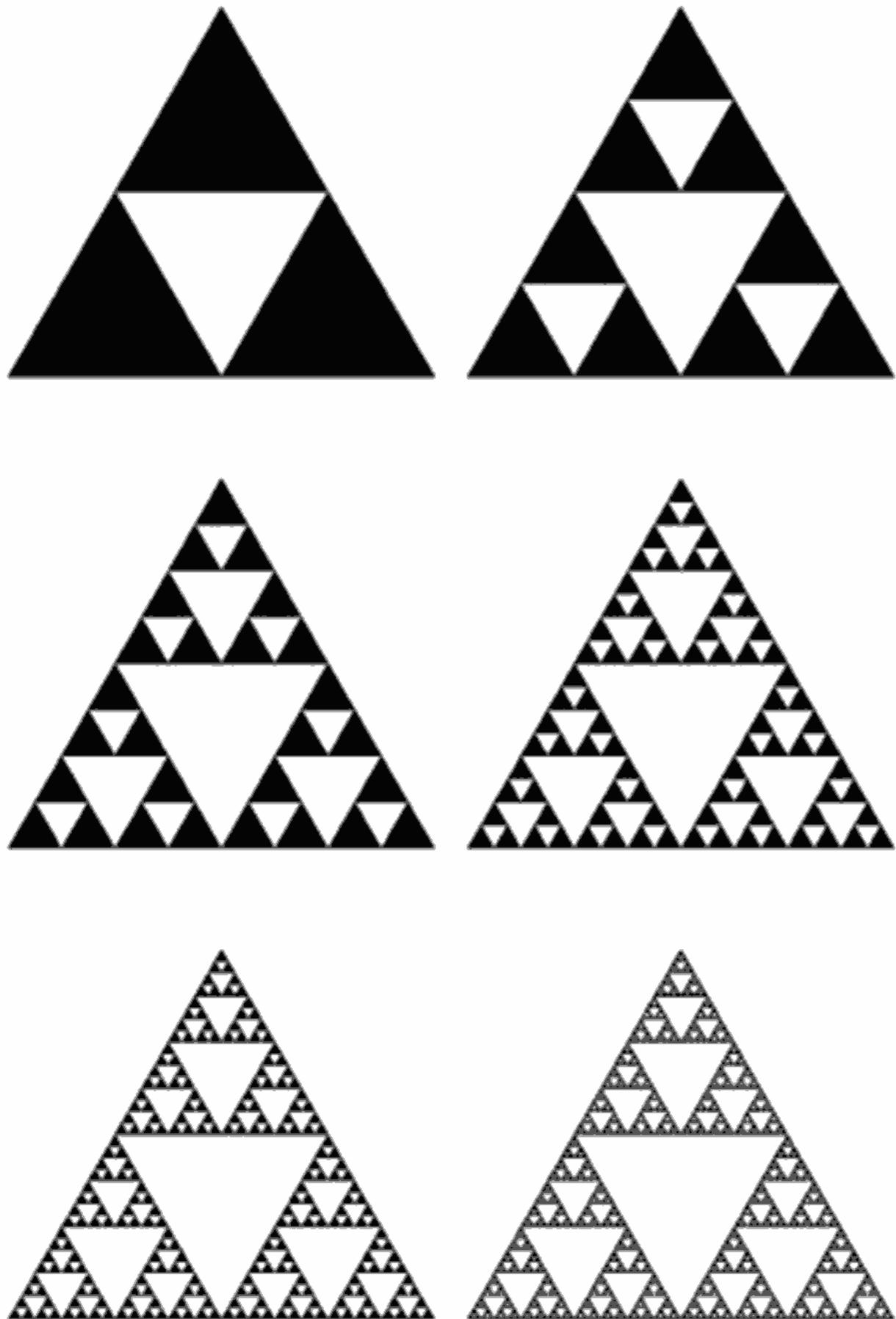


Figure 18.1: Sierpinski Gaskets at Different Division Levels

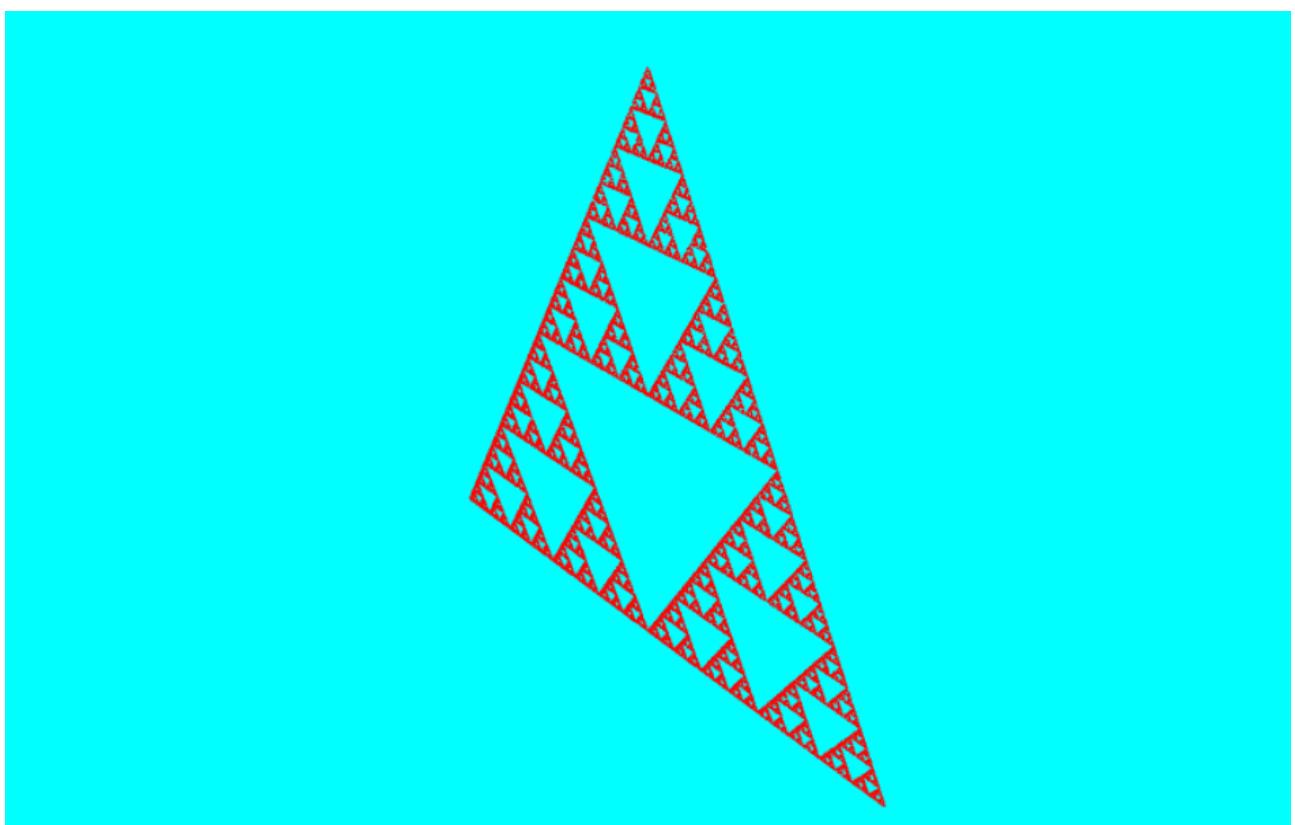


Figure 18.2: Output from Point Based Sierpinski Gasket

Part II

Working with Meshes

Lesson 19

Rendering a Cube

We have so far kept our drawing code to 2D shapes. Of course, this is great if we only want to do 2D work, but really we want to work in 3D. To do this, we just need to add more triangles. We can define a cube for example by just defining the triangles for the six faces (12 triangles in all).

19.1 Updating the Project

For this project you only need to update the geometry generation code as we did in the first few lessons. It is the number of triangles you have to add now that has changed. Previously we were working with only a few points. Now you will need 36 (3 for each triangle, 2 triangles per face, 6 faces). You have to work out how to define these points - you will have to think in three dimensions. Things to remember:

- Face direction - which way is the face pointing. Remember that we have back-face culling active. You don't want to render a face that points into the cube.
- Winding order - make sure you are defining the triangle in a counter-clockwise manner based on how you view the side

The starting code for this project has keyboard input defined. This will allow you to rotate the cube around to check the faces are visible. An example output from this application is shown in Figure 19.1.

19.2 Exercise

At the end of your initialise code, add the following line:

```
1 glDisable(GL_DEPTH_TEST);
```

What is the result? Can you determine what is happening without depth testing turned on? Think about the order that you are rendering the triangles and which ones might be overriding others.

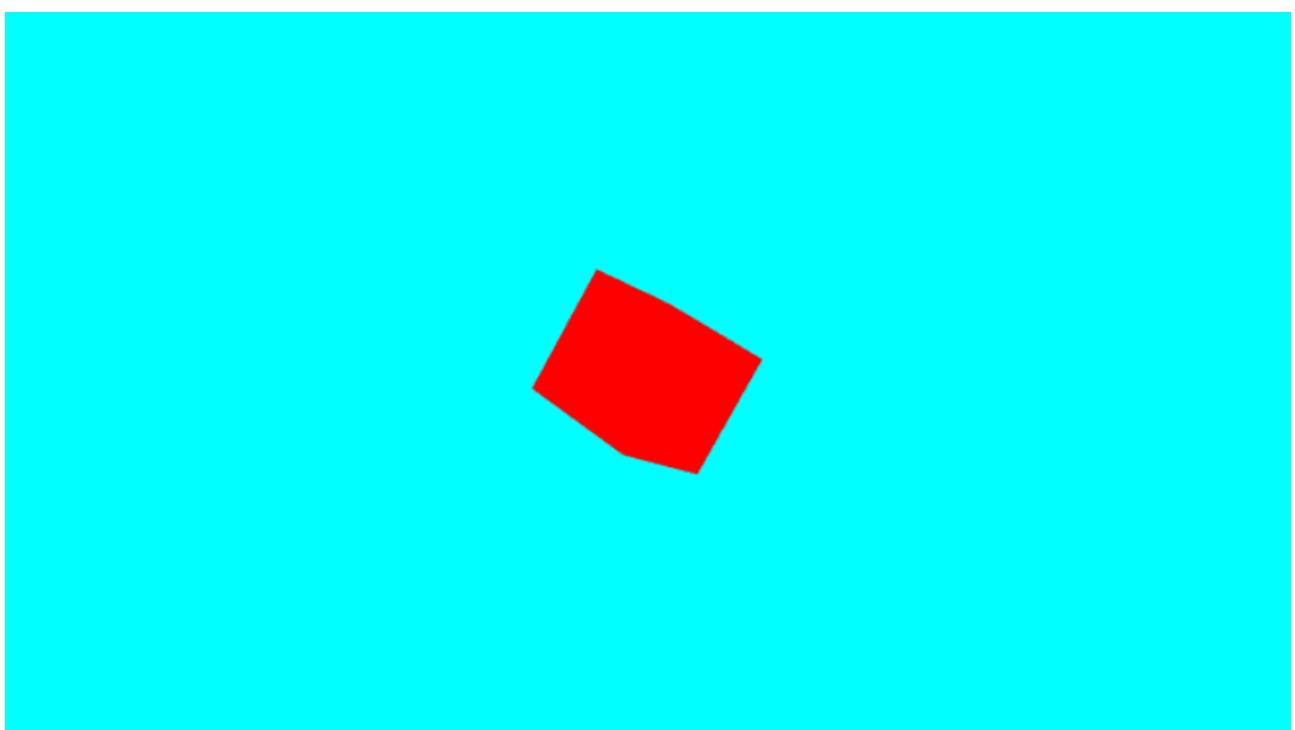


Figure 19.1: **Cube Render Output**

Lesson 20

Transforming a Cube

Now that you have a cube rendered, the next step is to utilise the transformation work we did previously. The code for this project has all the framework you need to undertake this. You just have to apply the knowledge gained from the last few lessons to complete the project code.

An example of the output from this lesson is shown in Figure 20.1.

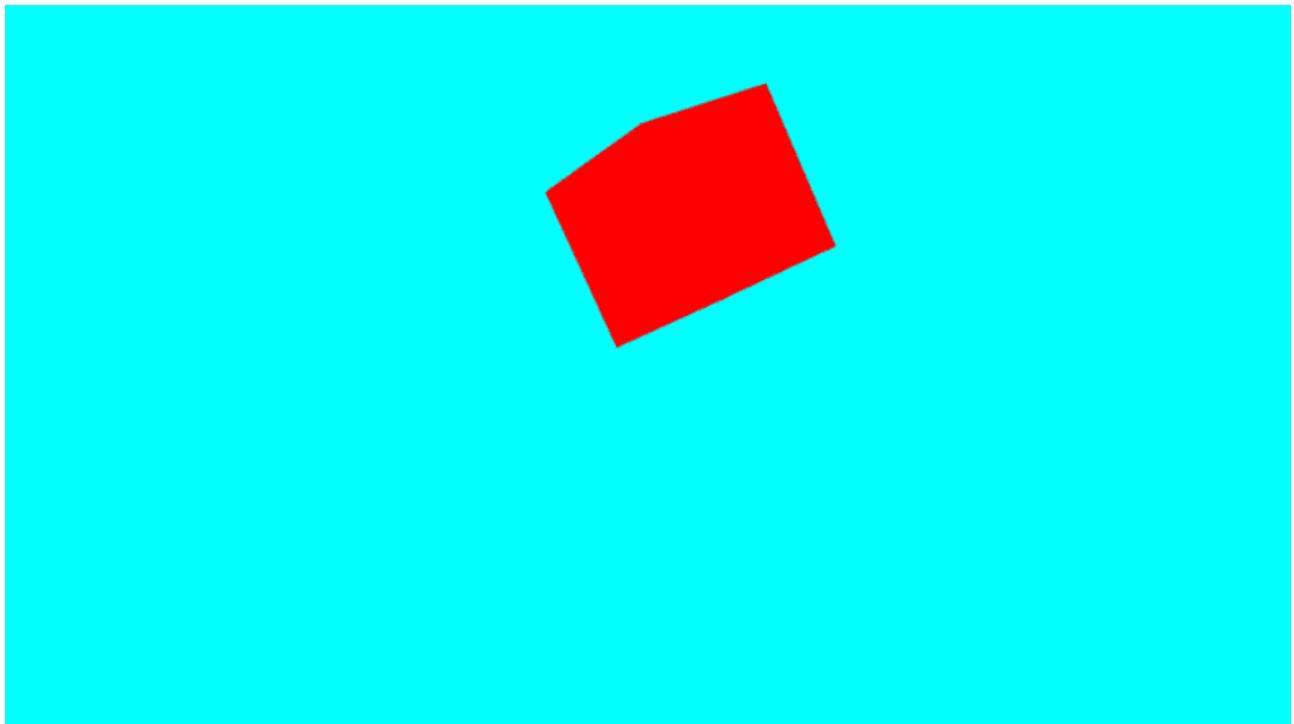


Figure 20.1: **Transformed Cube Output**

Lesson 21

Working with Indices

Up until now we have been writing quite a lot of code to draw very simple shapes. As you can imagine, we do not really want to be writing a line of code for every single vertex of our shapes (imagine doing this for a 3D character model with thousands of individual vertices). As we work through the lessons, we will explore a few different methods to remove the amount of code we have to write just to handle the geometry. The first approach we will use is indexing.

21.1 What is Indexing?

Let us consider how we draw a quad using triangles at the moment. Basically, we are defining two separate triangles as shown in Figure 21.1.

Obviously the triangles are connected in our actual application, but this is for illustrative purposes. If you look at what we are doing, we are defining six separate vertices to define a shape that only has four separate vertices. This is not very efficient. So what if we could reuse vertices. Well we can. Consider Figure 21.2.

We can consider each vertex of the quad as numbered above. In this instance, we can define our two triangles as follows:

$$\begin{aligned} &\{0, 2, 1\} \\ &\{0, 3, 2\} \end{aligned}$$

This is how indexing works. We can define an array of vertices, and then define shapes as collections of these vertices. For example, a cube is just eight vertices, but defined by twelve triangles. If we did not use indexing, then we would have to declare thirty-six vertices. Using indexing, we declare eight, and then define our twelve triangles as collections of three indexes in the array of eight vertices. Let us now see how we use indexing in OpenGL.

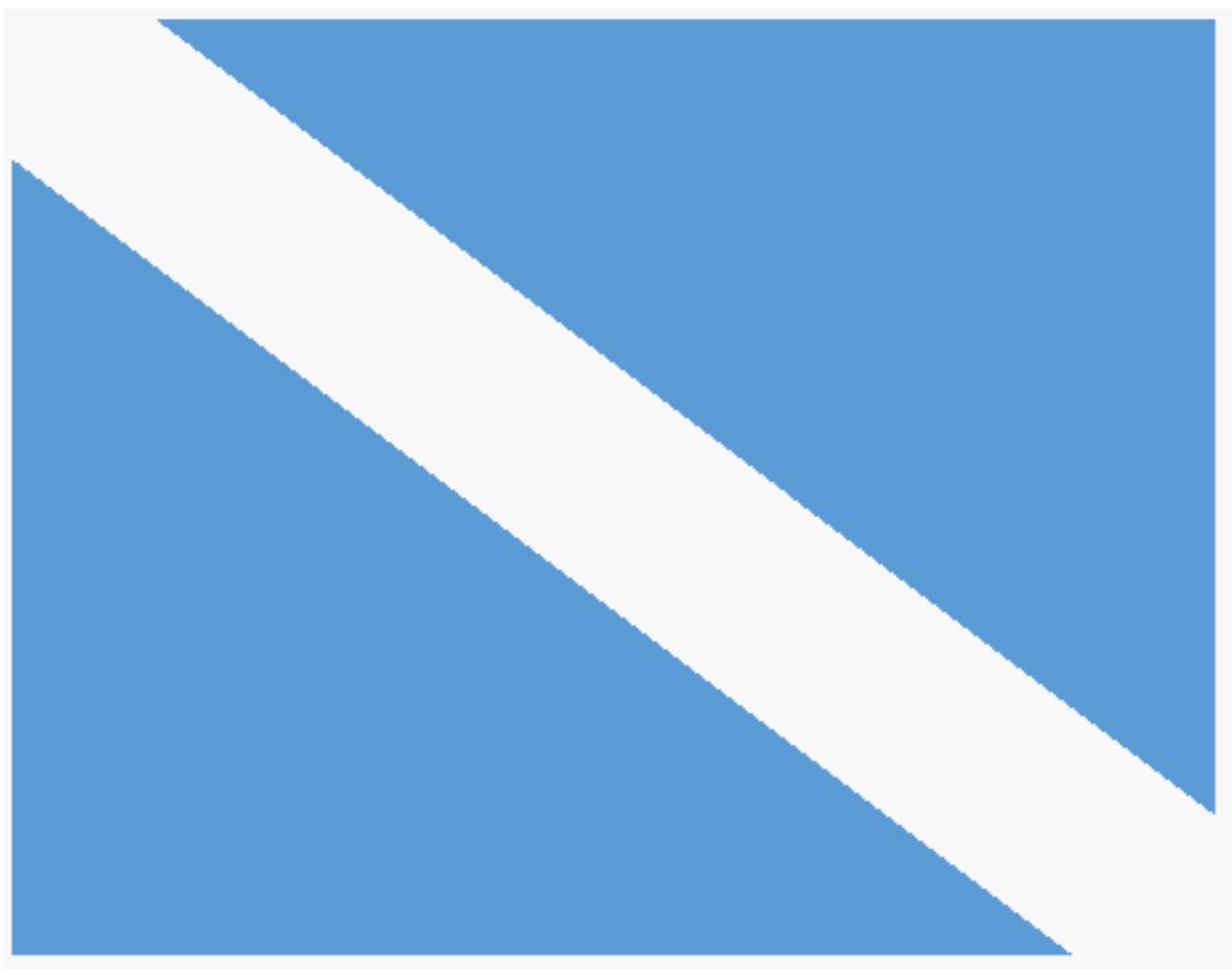


Figure 21.1: **Quad Built Using Triangles**

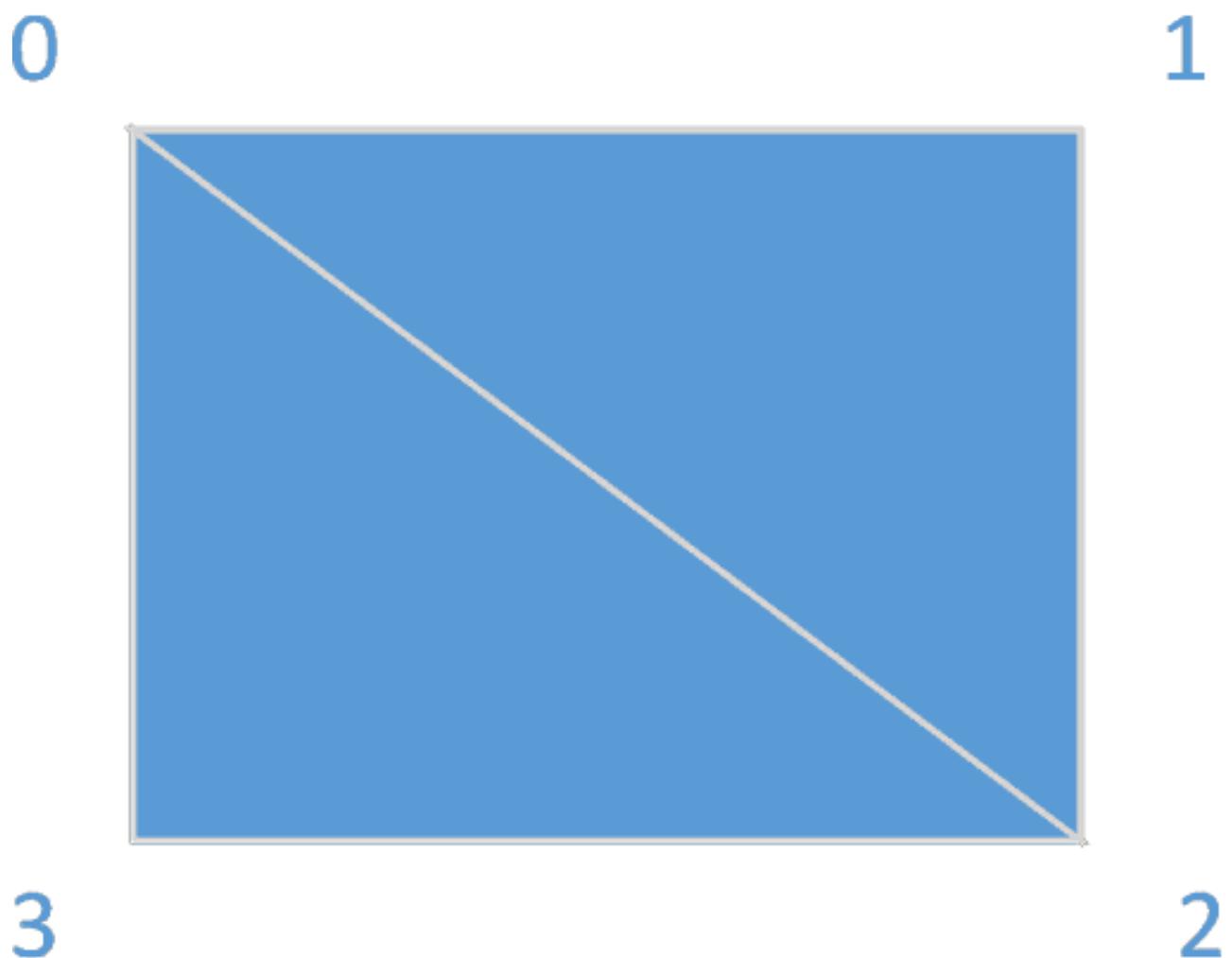


Figure 21.2: **Indexed Quad**

Lesson 22

Indexed Cube

Now that we have seen how to use indices, it is your turn to try and implement a version of our cube rendering application, but this time with indices. What this means is that you only require eight vertices now (one for each corner of the cube), but you require 36 indices (3 for each triangle, 2 triangles per face, 6 faces). You will have to work out which vertices to use for which triangles, and again remembering that winding order is important.

The initial project code for this lesson has declared the two necessary vectors (one of type `vec3` the other of `GLuint` or unsigned integers). You simply have to provide the values. You also need to add the indices to the geometry object. This is done using the `add_index_buffer` method on the geometry:

```
1 geom.add_index_buffer(indices);
```

Once run, you will should get an output similar to that shown in Figure 22.1. Remember to rotate around your cube to check that you have defined it correctly.

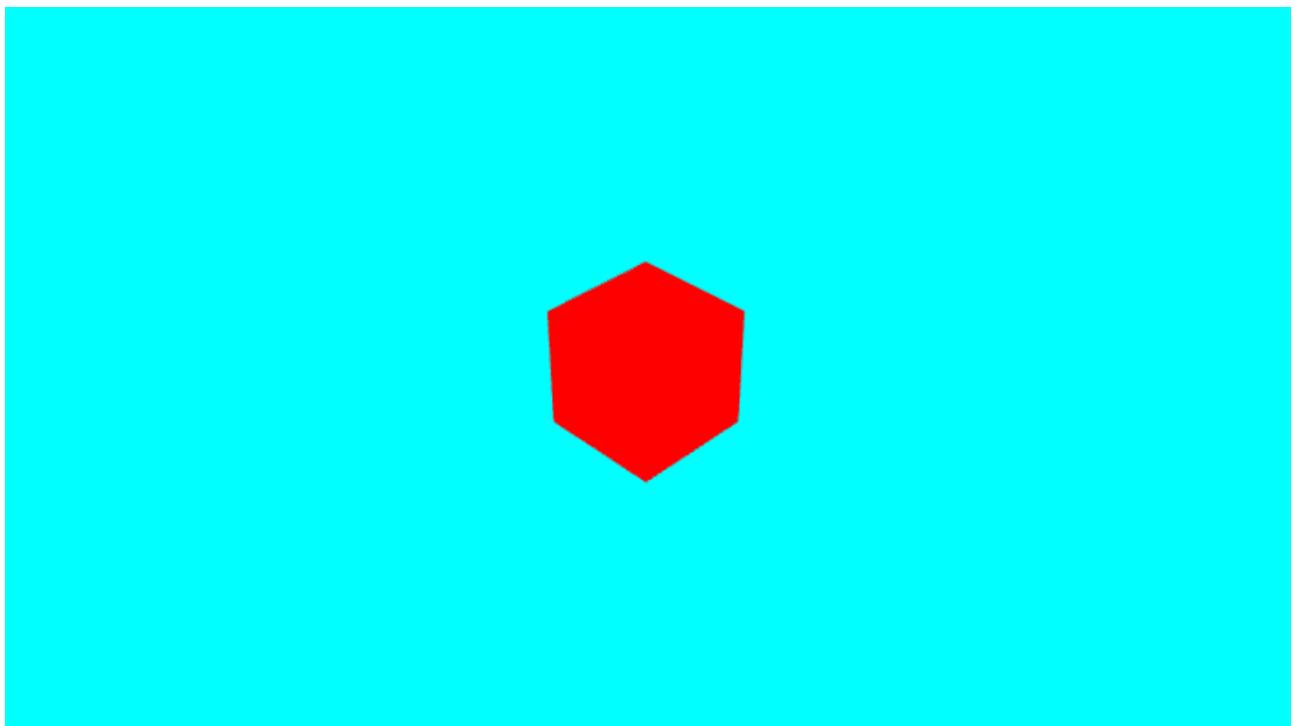


Figure 22.1: Rendered Cube using Index Data

Lesson 23

Sierpinski Gasket

We now know enough to do the Sierpinski Gasket example using a recursive algorithm and triangle sub-division. For the challenge, you will have to implement the division function.

23.1 Getting Started

The project for this lesson contains two operations that you will need to use or implement to get this application running. The first is a operation called `triangle`. This operation will add three vertices to our position and colour vectors. It has been implemented for you, and looks as follows:

```
1 void triangle(const vector<vec3> &points, vector<vec3> &positions, ←
  vector<vec4> &colours)
2 {
3     positions.insert(positions.end(), points.begin(), points.end());
4     for (auto i = 0; i < 3; ++i)
5         colours.push_back(vec4(1.0f, 0.0f, 0.0f, 1.0f));
6 }
```

Your job will be to call this operation when required. We will build up from a single triangle to a few other triangles as we work. The other operation - the one you have to implement - is `divide_triangle`. It currently does nothing.

23.2 Challenge Part 1

For 0 subdivisions, our output is just the triangle defined by the three vectors as shown in Figure 23.1.

Create an initial version of `divide_triangle` which checks if the count is 0, and if so uses the incoming vertices as the triangle definition. Therefore, the pseudocode for this version of the code is very simple, as shown in Algorithm 1.

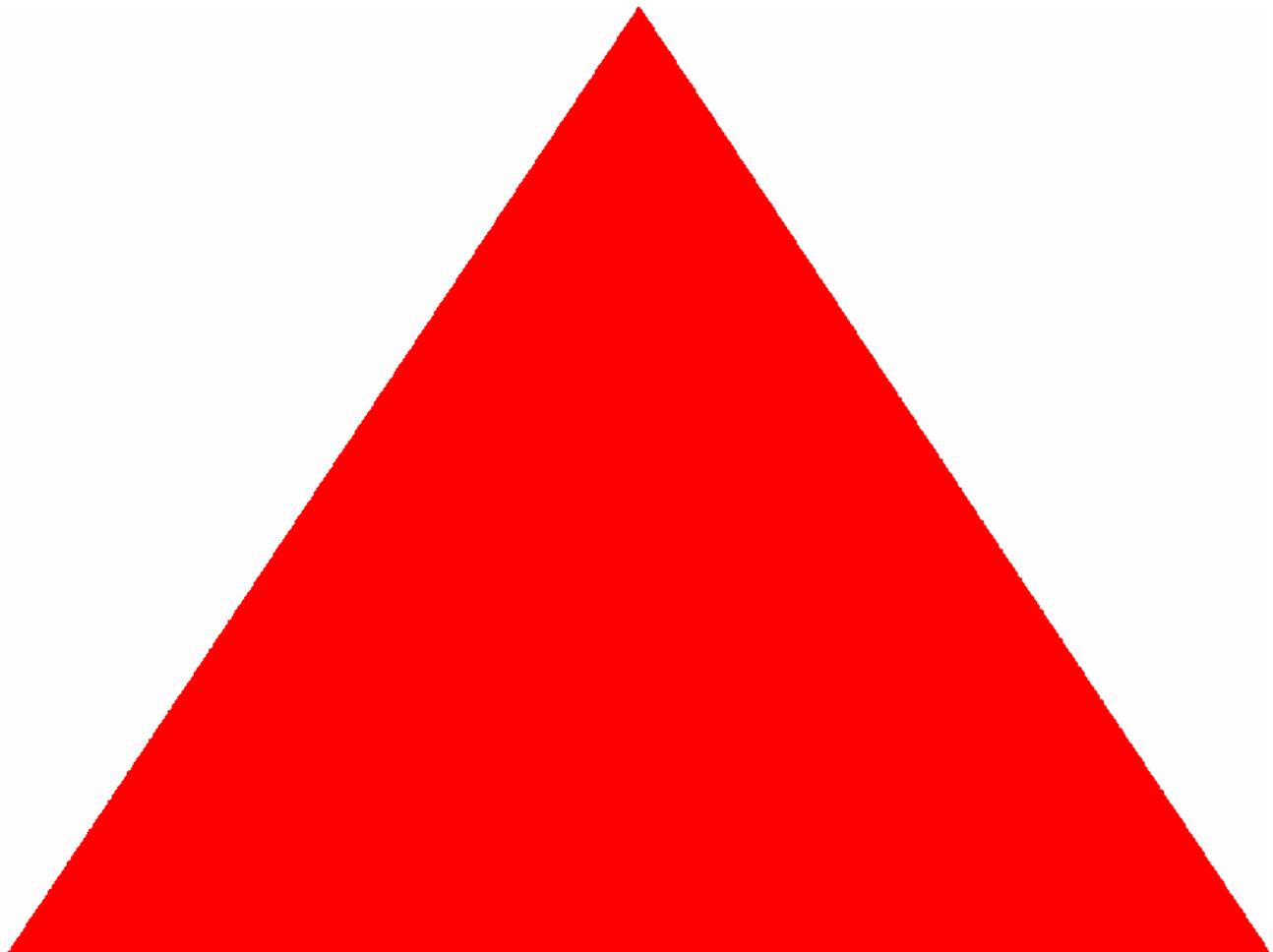


Figure 23.1: Sierpinski at 0 Divisions

Algorithm 1 Divide Triangle at 0 Subdivisions

Require: Size of *points* is 3

```

1: procedure SIERPINSKI(points, positions, colours)
2:   TRIANGLE(points, positions, colours)

```

23.3 Challenge Part 2

To subdivide the triangle for the Sierpinski Gasket, we need to use the idea shown in Figure 78.2 for the first subdivision.

From our first triangle we simply work out the midpoints on each side, and from these midpoints and the original points we define three new triangles.

Your challenge now is to implement the version that works for one subdivision of the Gasket. Do not worry about the recursive version yet. The pseudocode (assuming the vertices are numbered clockwise from 0 to 2 starting from the top) is given in Algorithm 2.

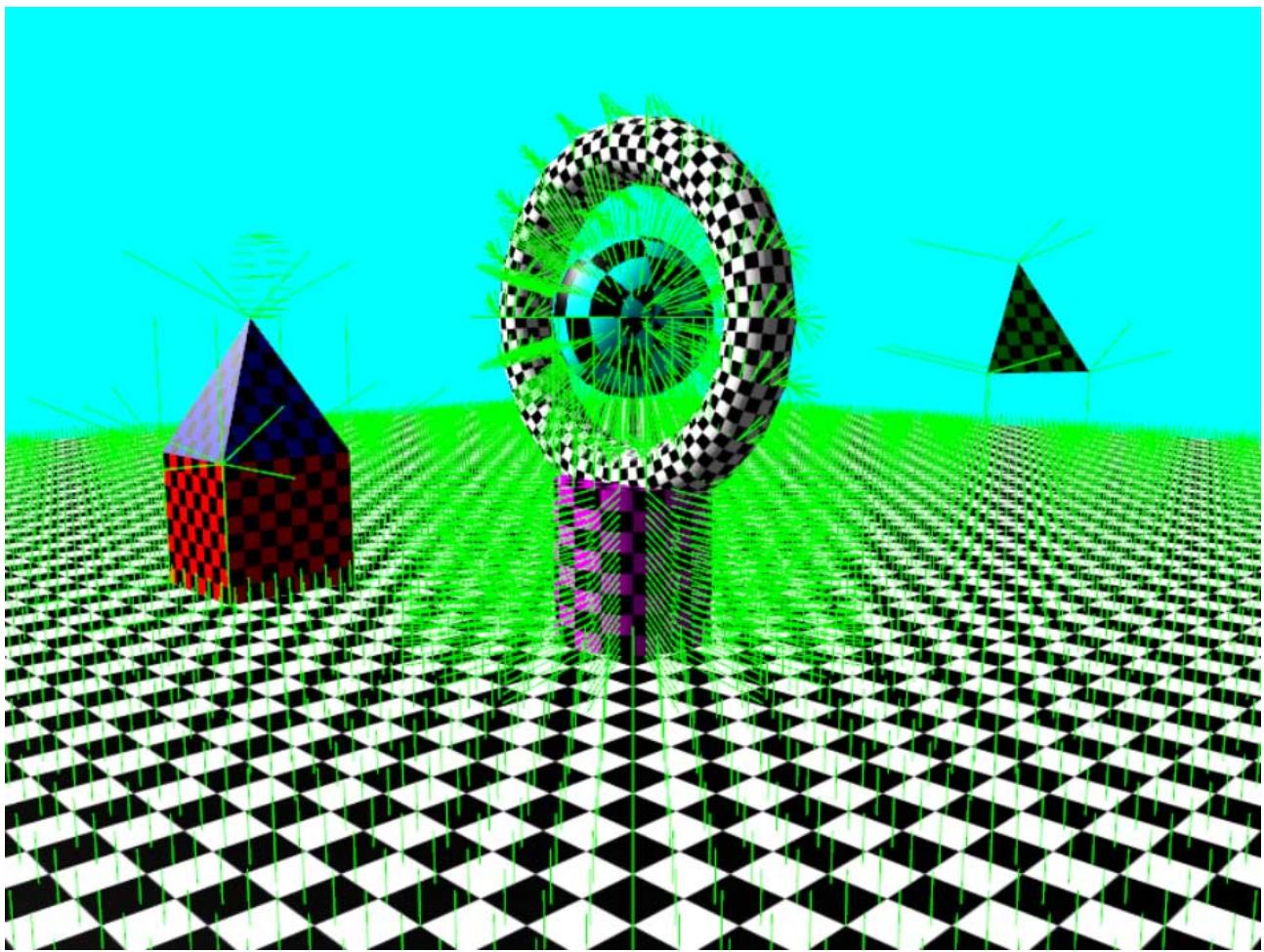


Figure 23.2: Sierpinski at 1 Division

23.4 Challenge Part 3

The next subdivision follows the same rule for each of its triangles. The second subdivision is shown in Figure 23.3.

Your goal now is to modify your current version of the `divide_triangle` operation to provide a generalised recursive solution which will work for any number of subdivisions. The pseudocode for this version is shown in Algorithm 3.

When you manage to get this running, you should get an output similar to that in Figure 23.4.

Algorithm 2 Divide Triangle for 1 Subdivision**Require:** Size of *points* is 3

```

1: procedure SIERPINSKI(points, positions, colours)
2:    $m_0 \leftarrow (points_0 + points_1)/2$ 
3:    $m_1 \leftarrow (points_1 + points_2)/2$ 
4:    $m_2 \leftarrow (points_2 + points_0)/2$ 
5:   TRIANGLE([points0, m1, m0], positions, colours)
6:   TRIANGLE([m1, points2, m2], positions, colours)
7:   TRIANGLE([m0, m2, points1], positions, colours)

```

Algorithm 3 Recursive Sierpinski Gasket**Require:** Size of *points* is 3

```

1: procedure SIERPINSKI(points, positions, colours)
2:   if count > 0 then
3:      $m_0 \leftarrow (points_0 + points_1)/2$ 
4:      $m_1 \leftarrow (points_1 + points_2)/2$ 
5:      $m_2 \leftarrow (points_2 + points_0)/2$ 
6:     DIVIDE_TRIANGLE([points0, m1, m0], positions, colours)
7:     DIVIDE_TRIANGLE([m1, points2, m2], positions, colours)
8:     DIVIDE_TRIANGLE([m0, m2, points1], positions, colours)
9:   else
10:    TRIANGLE(points, positions, colours)

```

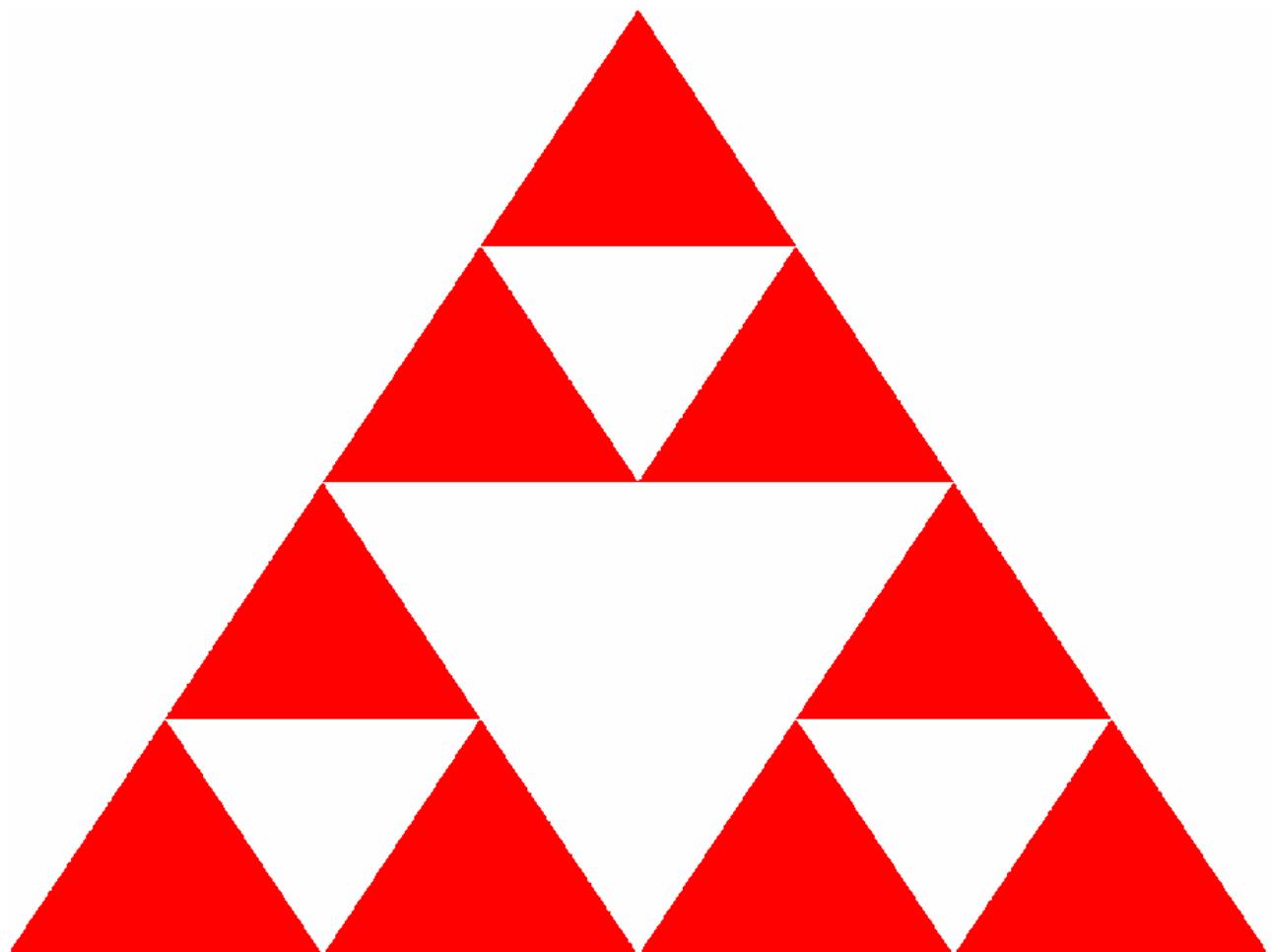


Figure 23.3: Sierpinski at 2 Divisions

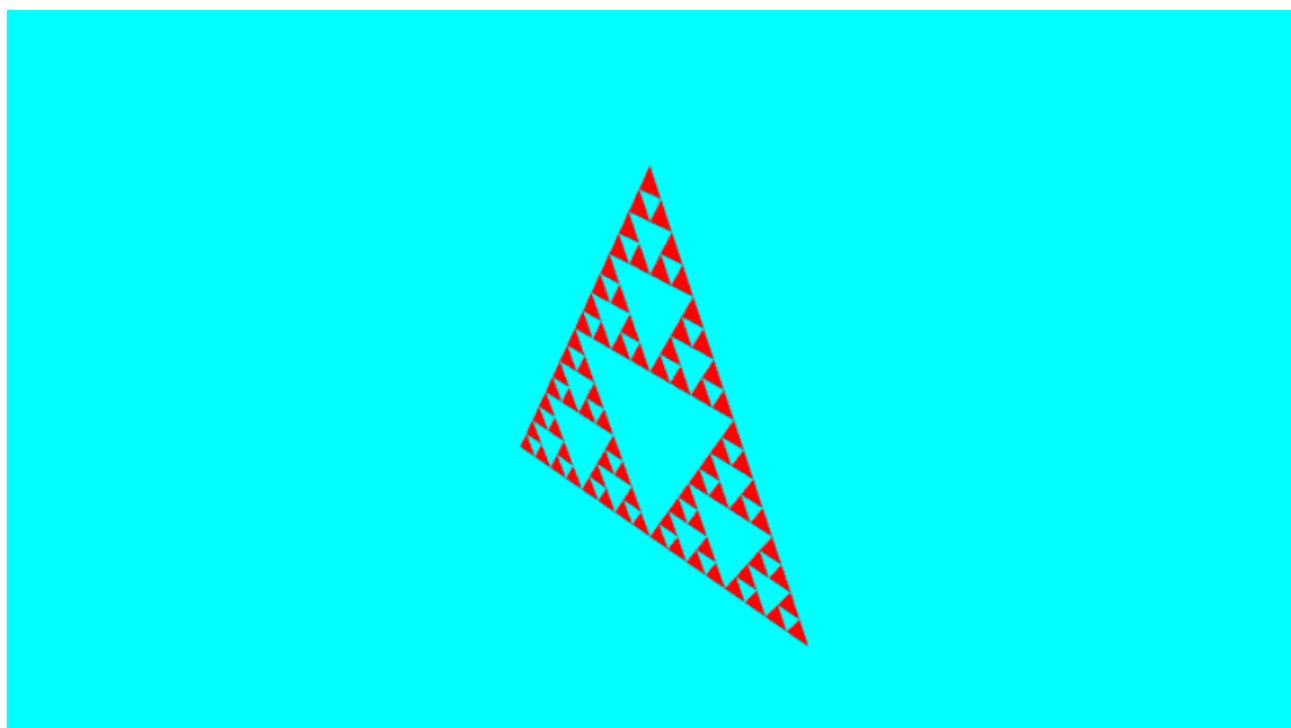


Figure 23.4: Sierpinski Gasket Output

Lesson 24

Sphere by Subdivision

Let us try and develop another approach to generating geometry using a recursive method by generating a sphere by using a sub-division technique.

24.1 Technique

The simple way to show how we are going to achieve a sphere using sub-division techniques is by looking at the example of a circle. We start initially with a triangle that fits inside a unit circle:

As the circle is unit size, each vertex of the triangle is exactly one unit away from the centre. We then find the midpoint in all the lines of the triangle:

And then use these points to subdivide the individual lines, and then push the midpoints out so they are unit length from the centre:

We continue this approach until we gain a rough approximation of a sphere. Figure 24.1 illustrates the general idea.

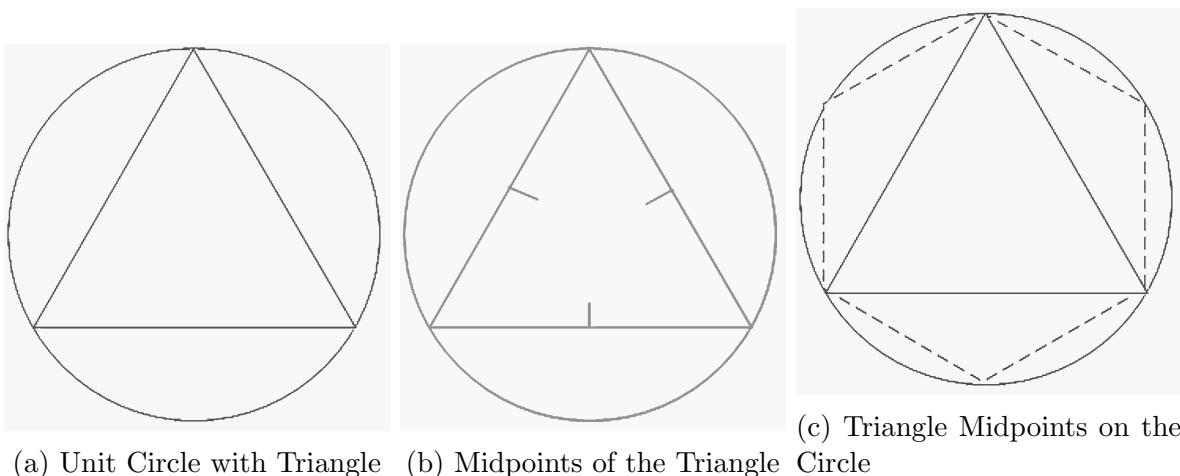


Figure 24.1: Circle by Subdivision

24.1.1 Getting Started

In the project for this lesson you will find a similar set up to the Sierpinski Gasket approach. The main operation you need to be working on is the `divide_triangle` one. This has been set up so that you just need to evolve the algorithm. We begin the application with a tetrahedron (a four sided shape with triangle on each face - like a pyramid with a triangle base). We will undertake the same process as the Sierpinski Gasket, but evolve from our tetrahedron to a sphere.

24.1.2 Implementation

Your task is to develop the `divide_triangle` operation. Remember how we approached things in the Sierpinski Gasket, and consider trying to do 0 sub-divisions, then 1, and then finally the general case.

Challenge 1 - 0 Subdivisions

As before, at 0 subdivisions we don't have to do anything. The pseudocode for this version is given in Algorithm 4.

Algorithm 4 Sphere Subdivision for 0

Require: Size of *points* is 3, *divisions* ≥ 0

```

1: procedure DIVIDE_TRIANGLE(points, divisions positions, colours)
2:   positions := positions + points            $\triangleright$  Insert the points into the positions vector
3:   for i  $\leftarrow 1$  to 3 do
4:     colours := colours + red

```

The code to insert a vector at the end of an existing vector in C++ is shown below. For the colour data you will have to use a for loop to add the 3 colour values accordingly. Or you could create a new version of the `triangle` operation from the Sierpinski Gasket lesson.

```
1 data.insert(data.end(), new_data.begin(), new_data.end());
```

The output for this will be a simple tetrahedron. You can rotate the shape to check if there are any gaps.

Challenge 2 - 1 Subdivision

Now we need code to perform 1 subdivision. This is quite trivial, and follows the same technique as the Sierpinski Gasket, except this time we have four triangles to create. The pseudocode is shown in Algorithm 5. Remember that the hat (\hat{n}) indicates that the resulting vector is normalized - so you will have to normalize the result of the calculation.

Again test the application. It will be a weird blob shaped object at 1 subdivision, but we are almost ready to undertake the full algorithm.

Algorithm 5 Sphere Subdivision for 1**Require:** Size of *points* is 3, *divisions* ≥ 0

```

1: procedure DIVIDE_TRIANGLE(points, divisions positions, colours)
2:    $v_0 \leftarrow \widehat{p_0 + p_1}$ 
3:    $v_1 \leftarrow \widehat{p_0 + p_2}$ 
4:    $v_2 \leftarrow \widehat{p_1 + p_2}$ 
5:   ▷ Triangle for  $p_0, v_0, v_1$ 
6:   ▷ Triangle for  $p_2, v_1, v_2$ 
7:   ▷ Triangle for  $p_1, v_2, v_0$ 
8:   ▷ Triangle for  $v_0, v_2, v_1$ 

```

Challenge 3 - Recursive Subdivisions

We are now in a place to write the recursive version of this algorithm. This is shown in Algorithm 6.

Algorithm 6 Recursive Sphere Subdivision**Require:** Size of *points* is 3, *divisions* ≥ 0

```

1: procedure DIVIDE_TRIANGLE(points, divisions positions, colours)
2:   if divisions  $> 0$  then
3:      $v_0 \leftarrow \widehat{p_0 + p_1}$ 
4:      $v_1 \leftarrow \widehat{p_0 + p_2}$ 
5:      $v_2 \leftarrow \widehat{p_1 + p_2}$ 
6:     DIVIDE_TRIANGLE( $[p_0, v_0, v_1]$ , divisions − 1, positions, colours)
7:     DIVIDE_TRIANGLE( $[p_2, v_1, v_2]$ , divisions − 1, positions, colours)
8:     DIVIDE_TRIANGLE( $[p_1, v_2, v_0]$ , divisions − 1, positions, colours)
9:     DIVIDE_TRIANGLE( $[v_0, v_2, v_1]$ , divisions − 1, positions, colours)
10:   else
11:     positions := positions + points           ▷ Insert the points into the positions vector
12:     for i  $\leftarrow 1$  to 3 do
13:       colours := colours + red

```

When you get this working, you should have an output similar to that illustrated in Figure 24.2.

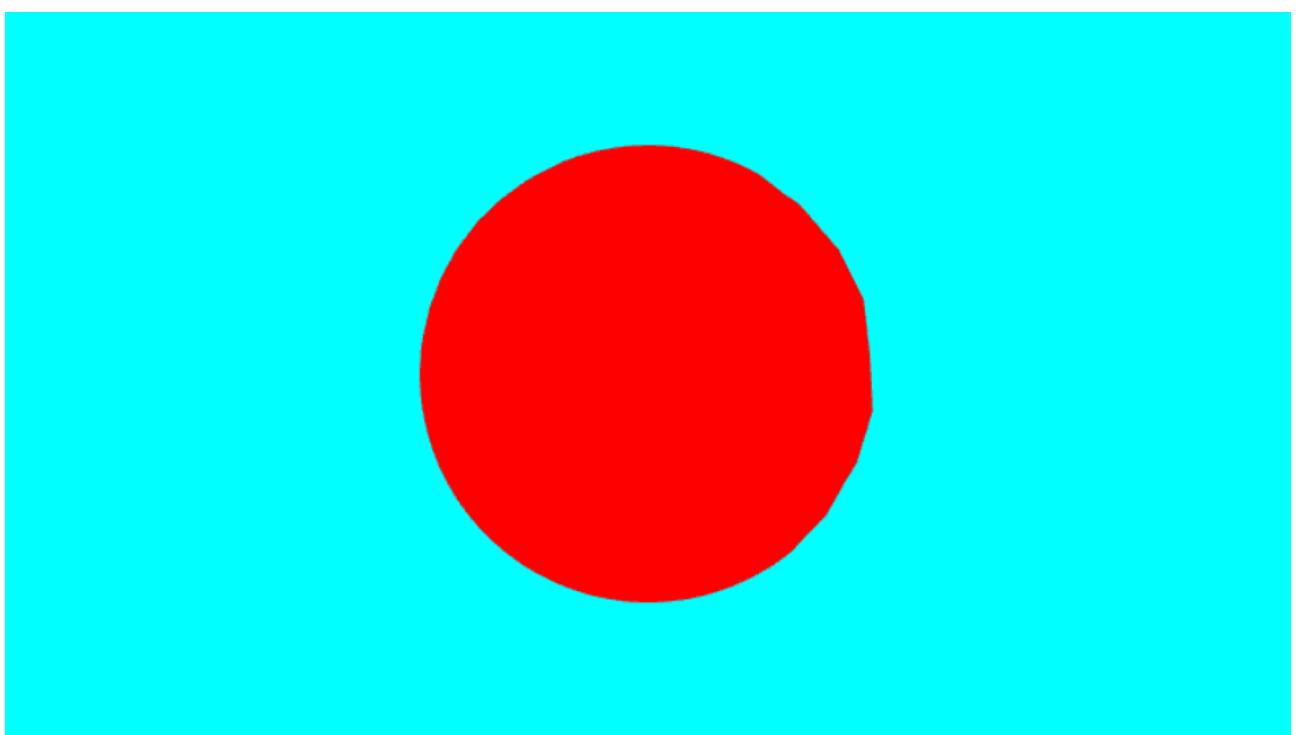


Figure 24.2: Sphere by Subdivision Output

Lesson 25

Meshes

Rendering basic geometry is one thing, but really what we want to do is combine geometry with a transformation and also attach any graphical properties (more on this later). We will call this combination of components a mesh - a quite standard name used for a 3D graphical object in a scene.

25.1 What is a Mesh?

A mesh is just a collection of data. In the render framework, the code is as follows:

```
1 /*  
2  An object combining a transform, geometry and material to create a←  
   renderable  
3  mesh object  
4 */  
5 class mesh  
6 {  
7 private:  
8     // The transform object of the mesh  
9     transform _transform;  
10    // The geometry object of the mesh  
11    geometry _geometry;  
12    // The material object of the mesh  
13    material _material;  
14 public:  
15     // Creates a mesh object  
16     mesh()  
17     {  
18     }  
19     // Creates a mesh object with the provided geometry  
20     mesh(geometry &geom)  
21     {  
22     }  
23     // Creates a mesh object with the provided geometry and material  
24     mesh(geometry &geom, material &mat)  
25     {  
26     }  
27     // Default copy constructor and assignment operator
```

```

28     mesh(const mesh &other) = default;
29     mesh& operator=(const mesh &rhs) = default;
30     // Destroys the mesh object
31     ~mesh() { }
32     // Gets the transform object for the mesh
33     transform& get_transform() { return _transform; }
34     // Gets the geometry object for the mesh
35     const geometry& get_geometry() const { return _geometry; }
36     // Sets the geometry object for the mesh
37     void set_geometry(const geometry &value) { _geometry = value; }
38     // Gets the material object for the mesh
39     material& get_material() { return _material; }
40     // Sets the material object for the mesh
41     void set_material(const material &value) { _material = value; }
42 };

```

You don't need to type this code in - it is provided as part of the render framework. The structure of the `mesh` class can be illustrated as shown in Figure 25.1.

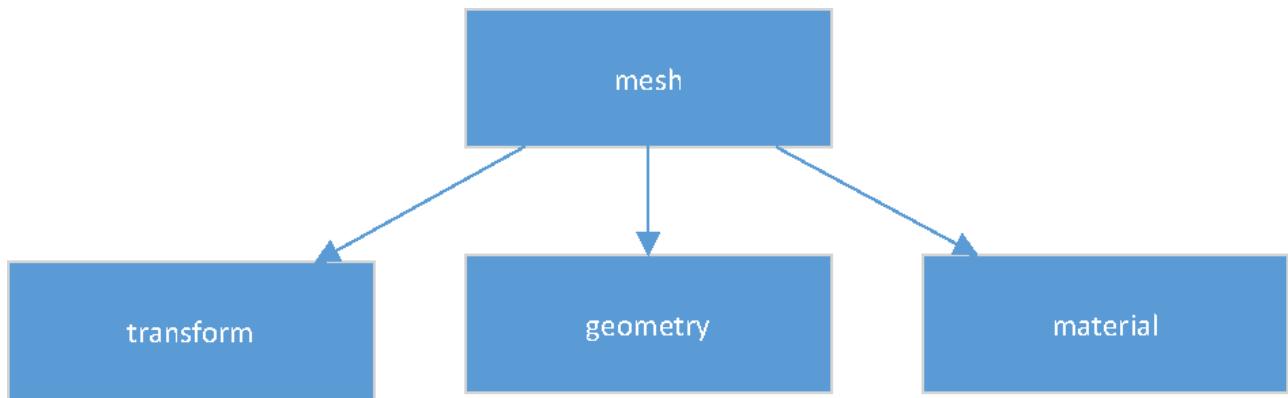


Figure 25.1: Mesh Class Structure

A `mesh` is made of three subclasses:

`transform` Data used to generate a transformation matrix for the mesh. Contains values such as position and rotation.

`geometry` The data required to render the geometry to the scene based on the `transform`.

`material` Data used to provide graphical effects to our rendered geometry. We will come to this later.

We will now look at how we create a mesh.

25.2 Creating a Mesh

Creating a mesh is relatively easy. The following code will illustrate.

```

1 m = mesh(geom);

```

We use the constructor for `mesh` that takes in a `geometry` object. This is the typical approach we will take, although you can look at the structure of the `mesh` class to see other techniques we

could use. The main point of a `mesh` is that you should at a minimum provide some `geometry` to render. We will look at the other parts of the `mesh` object over time.

25.3 Rendering a Mesh

Rendering our `mesh` is as simple as when we rendered `geometry`. We just call `render` on the `renderer` as before:

```
1 renderer::render(m);
```

Lesson 26

Transforming a Mesh

Now that we are using meshes, let us look at how we go about moving our objects around the scene by manipulating the `transform` of the mesh. First, let us look at what is contained in our `transform` object.

26.1 transform

The definition of our `transform` object is shown below.

```
1 /*  
2 Utility class used to store transformation data  
3 */  
4 struct transform  
5 {  
6     // The position of the 3D object  
7     glm::vec3 position;  
8     // The orientation of the 3D object  
9     glm::quat orientation;  
10    // The scale of the 3D object  
11    glm::vec3 scale;  
12  
13    // Creates a transform object  
14    transform() : scale(glm::vec3(1.0f, 1.0f, 1.0f)) {}  
15  
16    // Translates the 3D object  
17    void translate(const glm::vec3& translation)  
18    {  
19        position += translation;  
20    }  
21  
22    // Rotates the 3D object using Euler angles  
23    void rotate(const glm::vec3 &rotation)  
24    {  
25        glm::quat rot(rotation);  
26        rotate(rot);  
27    }  
28  
29    // Rotate the 3D object using the given quaternion
```

```

30 void rotate(const glm::quat &q)
31 {
32     orientation = orientation * q;
33     orientation = glm::normalize(orientation);
34 }
35
36 // Gets the transformation matrix representing the defined ←
37 // transform
38 glm::mat4 get_transform_matrix()
39 {
40     auto T = glm::translate(glm::mat4(1.0f), position);
41     auto S = glm::scale(glm::mat4(1.0f), scale);
42     auto R = glm::mat4_cast(orientation);
43     auto matrix = T * R * S;
44     return matrix;
45 }
46
47 // Gets the normal matrix representing the defined transform
48 glm::mat3 get_normal_matrix()
49 {
50     return glm::mat3_cast(orientation);
51 }

```

There are three values stored in the `transform`:

`position` The position of the `mesh` within the scene.

`orientation` The orientation of the `mesh` within the scene. We store this using a quaternion.

`scale` The scale of the `mesh` within the scene.

These values can be manipulated directly, or can be manipulated using the helper methods for translation and rotation. The `transform` also has methods to generate the transformation matrix for us. It is up to you how you decide to work with the `transform`, but you should probably use the helper methods.

26.2 Using the Transformation

There are two tasks that you have to undertake in this lesson - apart from adding the code to create and render the matrix. The first is to add controls to manipulate the position, orientation and scale of the mesh. The controls you need to implement are given in following table:

WSAD	move the mesh
Up & down arrows	rotate on X-axis
Right & left arrows	rotate on Z-axis
O & P	scale

26.3 Getting the Transform Matrix

The other task you have to do in this lesson is getting the transform matrix from the `mesh` and then using it to create our MVP matrix. For a hint, the code below should help:

```
1 auto M = m.get_transform().get_transform_matrix();  
2 // ... calculate the rest of the MVP matrix
```

Running this application will give the same result as the previous one, except you can now manipulate the mesh using the controls defined.

Part III

Shaders and Texturing Effects

Lesson 27

Shaders

We have raced through the basic use of the mesh object since you should find this familiar. Next we are going to look at some more advanced concepts, and in particular some modern techniques to graphics rendering now that the rendering framework is in place. The first technique we are going to look at is graphics card programming, using what is called shaders. Graphics card programming is the technique that allows us to create far more realistically rendered scenes with just some simple modifications to our existing approach.

27.1 What is a Shader?

To put it simply, a shader is just a program that runs on the GPU, allowing us to create a particular effect based on input values, geometry, textures, etc. A shader is programmed in a separate language to the main application. Typically the language is C-like, and there are a number to choose from based on the platform you are working on:

High Level Shader Language (HLSL) A DirectX language.

GL Shader Language (GLSL) An OpenGL language.

Cg An Nvidia proprietary language.

For our work we will be using GLSL, for obvious reasons.

A shader integrates into what is called the programmable pipeline, which is shown in Figure 27.1.

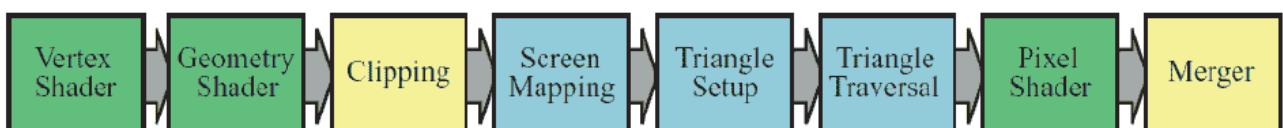


Figure 27.1: Graphics Pipeline

This image is taken from Real-Time Rendering [1]. The parts of the pipeline we are most interested in are coloured green. These are the programmable parts of the graphics pipeline. Each serves a different purpose, and we will look at these shortly.

27.2 How do Shaders Work?

A shader is just one part of the process that converts our geometric data and other information into the image you finally see on the screen. As Figure 27.1 illustrated, there are actually three different types of shader programs that operate on the graphics pipeline. Each of these has a different purpose. At the start, the vertex shader is responsible for taking in raw geometric data, and positioning it world space (along with some other possible functions). Its main responsibility is to take a single vertex (a vertex shader is written to work on one piece of vertex information at a time), and convert the vertex's position into world space (using the transformation matrices we have already discussed). At the opposite end of the pipeline, the pixel shader (or fragment shader as it is called in OpenGL terms) is responsible for determining the individual pixel colours associated with a triangle when represented on screen. This can include texture information, information from the vertex shader, or other lighting considerations.

27.2.1 Why Shaders are Fast

A good question to ask now is why shaders are fast at producing rendering effects. The main advantage shaders have is that although you develop a single program, multiple copies of this program are running on the GPU at one time. This is possible because each individual vertex can be transformed independently, and each individual triangle can also be worked on independently. This allows the GPU to run many versions of your programs, each operating on one piece of geometry information at a time. Once the GPU has processed all information for a single piece of geometry (e.g. a cube, a 3D character, etc.), then other information and programs can be run through the GPU, producing a different part of the screen.

27.3 Shader Types

As was illustrated in Section 27.2, there are three shader types (actually there are now four but we won't concern ourselves with the new tessellation shader at the moment). We will look at these in the following sub-sections.

27.3.1 Vertex Shader

A piece of geometry sent to the graphics card could be considered just a triangle mesh (a collection of triangles making up a shape). Each triangle is made up of vertices (which it is the vertex shader's responsibility to manipulate). These vertices can contain more than positional information - and in fact it is more common that they do contain more than just positional data. Colour, texture coordinates, normals, and any other information relative to a single vertex can be sent into the vertex shader. It is the job of the vertex shader to use this information to output the world position of the vertex, and also output any other generated information which might be useful to the later programmable stages.

27.3.2 Geometry Shader

Whereas the vertex shader deals with an individual vertex, the geometry shader works on a single primitive piece of geometry (e.g. a triangle, a line, a point, etc.). This input primitive is then used to generate zero or more output primitives. This allows the geometry shader to add more geometry to a scene if required by the shader program. This allows such effects as fur and other concepts.

The geometry shader also allows data to be output from the GPU at this stage. What this means is that we can process a collection of vertex data into other geometry data, and then use this data for some other effect later in the render (in other words we don't output our draw to screen, but rather to memory).

We won't look at the geometry shader till much later in the module.

27.3.3 Fragment (or Pixel) Shader

The pixel shader is responsible for determining the individual pixel colours represented on screen. This involves determining what colour the triangle is, using such techniques as normal colour, lighting, or texturing.

As with the geometry shader, it is also possible to output to memory rather than the screen at this stage. However, this time rather than raw data, we are creating a texture in memory which we can use for some other part of the render process.

27.4 GLSL

As we are working in OpenGL, we will be using GLSL as our shader language. The goal here is not to describe all the fine details of GLSL, but rather build up your understanding by working through examples.

GLSL is a C-like language, and in general you should be familiar enough with the syntax we will discuss. To give you a brief example of what some shader code may look like, examine the following:

```

1 #version 440
2
3 uniform mat4 MVP;
4
5 layout (location = 0) in vec3 position;
6
7 void main()
8 {
9     gl_Position = MVP * vec4(position, 1.0);
10 }
```

We will not examine this code now, but hopefully you can see that there is nothing unusual about GLSL as a language. If you are interested in having code highlighting in Visual Studio for shaders, then take a look at NShader.

27.5 Using Shaders

There are actually a number of stages to loading and using a shader, which we will elaborate upon as we work through the rest of the module. Essentially we have to go through the following stages:

1. Read in the file containing the shader code
2. Create a shader object
3. Attach the read in text from the file to the shader object
4. Compile the shader object
5. Repeat 1 - 4 for all the shaders you want to load
6. Create a program object
7. Attach the necessary shader objects to the program object (can be many)
8. Link the program object
9. Repeat 6 - 8 for all the programs you want to create

We have used the word object here, but this isn't the same idea as we use when developing object-oriented systems. OpenGL keeps track of the shader and program objects for us, assigning each a number when we create it. For example, we declare a variable for our shader and program as follows:

```
1 GLuint shader;
2 GLuint program;
```

(A `GLuint` is just an `unsigned int`). We can then create an object using an OpenGL command. For example:

```
1 GLuint shader = glCreateShader(type);
```

The `type` parameter is the type of shader we want to create (vertex shader, fragment shader, etc.). The `glCreateShader` command returns a value that is essentially an index for the created shader object (for example, the first created object returns a value of 1, the second 2, etc.). This means OpenGL manages the actual object for us; we just need the shader index number to call commands upon it. This is a common approach in OpenGL, and you will see us using this technique in many other places.

You may ask the reason why we use `GLuint` for our shader value. The answer is that we will never have values in the negative range, so there is no point having a signed value. Also, we can check if the object has been initialised correctly by checking that the value of shader is not equal to 0 (the same as if we were checking that an object was not equal to null).

27.5.1 Compiling Shaders

When it comes to loading and compiling shader objects, we go through the steps illustrated by the code below

```

1 auto file_content = read_file(filename);
2 auto shader = glCreateShader(type);
3 auto source = file_content.c_str();
4 glShaderSource(shader, 1, &source, 0);
5 glCompileShader(shader);

```

We read in the text from the file (line 1). We then create a shader object of the necessary type (line 2). We then get the underlying character array of the string (line 3) and use this to attach as the shader source for the `glShaderSource` command on line 4. OpenGL relies on character arrays rather than C++ strings, as it is essentially C based rather than C++. Hence we have to get the character array as above. The `glShaderSource` command takes four parameters:

1. The shader object index to use with this source
2. The number of character arrays that are to be used as source for the shader
3. A pointer to an array of character arrays used for our source
4. A pointer to an array of `uint` that contains the length of each character array. We use 0 (`null`) as this means OpenGL assumes the character arrays are null terminated. In our case they are as we have used a string to read in the file in the first place.

27.5.2 Compilation Problems

It may be the case that we have a problem when we compile the shader. From our application point of view, we will get an error that wouldn't really reflect the problem in our shader code. Because of this, we have to get the compilation log from OpenGL. We do this as follows:

```

1 GLint compiled;
2 glGetShaderiv(shader, GL_COMPILE_STATUS, &compiled);
3 if (!compiled)
4 {
5     GLsizei length;
6     char log[1024];
7     glGetShaderInfoLog(shader, 1024, &length, log);
8     cout << log << endl;
9     glDeleteShader(shader);
10 }

```

We get the state of the compilation by calling the `glGetShaderiv` command on line 2. We then check to see if the shader compiled OK. If not, we create a small buffer (technically we should get the length of the log and then create the correct sized buffer - we are being a bit lazy here), and then get the compilation log by calling `glShaderInfoLog`. We then print out the log accordingly. This can be a very useful piece of information when our applications don't run.

27.5.3 Linking Programs

Creating and linking programs follows a similar approach, although now we are attaching shader objects to a program rather than source code. The following shows the general process:

```

1 auto program = glCreateProgram();
2 for (int i = 0; i < count; ++i)
3     glAttachShader(program, shaders[i]);
4 glLinkProgram(program);

```

We create a program object (line 1), and then attach all the necessary shader objects to the program (assuming an array of shader objects, then we use the for loop defined on line 2) using the `glAttachShader` command. Finally, once all the necessary shader objects are attached we call `glLinkProgram` on the program object.

27.5.4 Linking Problems

As with compiling shaders, we may hit a problem when linking. The following code is similar to checking for compilation problems:

```

1 GLint linked;
2 glGetProgramiv(program, GL_LINK_STATUS, &linked);
3 if (!linked)
4 {
5     GLsizei length;
6     char log[1024];
7     glGetProgramInfoLog(program, 1024, &length, log);
8     cout << log << endl;
9     for (int i = 0; i < count; ++i)
10        glDetachShader(program, shaders[i]);
11     glDeleteProgram(program);
12 }

```

You should be able to determine what we are doing from the previous description.

27.6 Shaders in the Graphics Framework

Thankfully, you don't have to worry too much about what is happening when loading and compiling shaders as the render framework will take care of most of the work for you.

27.6.1 Creating Effects

The first thing we need to do in our graphics framework is to create an effect object:

```

1 effect eff;

```

An `effect` object is just a collection of shaders - typically a vertex and fragment shader. We use this object to load in shader files. Consider an effect object as the particular combination of shaders that we wish to apply to an object. The more interesting part is the adding shaders to the effect object.

27.6.2 Loading Shaders

To load a shader into an effect object we just need to tell it the file to load and the type of shader it is:

```
1 eff.add_shader("filename", type);
```

For the type value, the following are used:

GL_VERTEX_SHADER a shader used to control the manipulation of vertex data. This is the first stage of the programmable pipeline

GL_FRAGMENT_SHADER a shader used to control the final pixel colour rendered. This is the final stage of the programmable pipeline

GL_GEOMETRY_SHADER a shader used to manipulate individual geometric primitives (e.g. triangles, lines, etc.). This is the final vertex processing stage (the next programmable stage is the fragment shader). This shader is optional.

GL_TESS_CONTROL_SHADER tessellation shader stage

GL_TESS_EVALUATION_SHADER tessellation shader stage

GL_COMPUTE_SHADER compute (general purpose) shader stage

If when a shader is added to the effect there is a compilation problem, the output will be shown on the console window.

27.6.3 Building Effects

Once you have added all the shaders you require to your `effect` you need to build it. This is done using the `build` method on the effect:

```
1 eff.build();
```

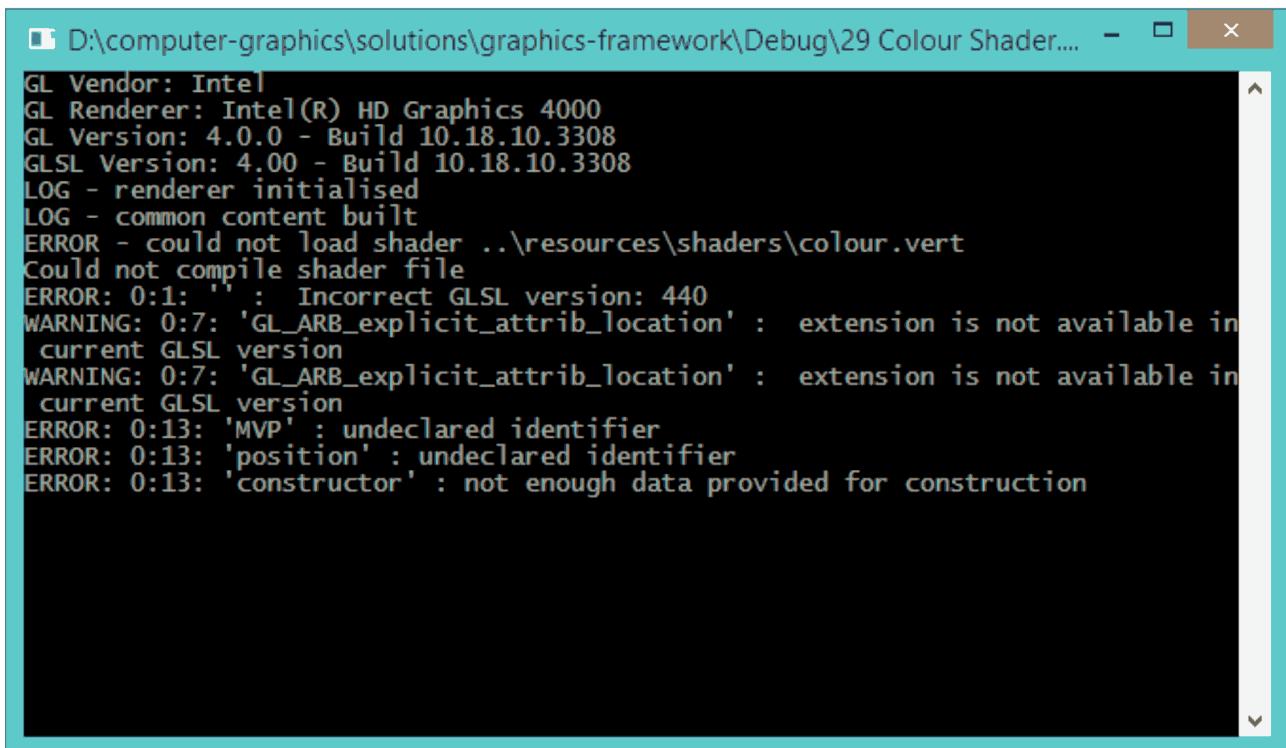
At this stage, any linking problems combining the relevant shaders into an effect will be displayed on the console window. For example, we may get the output shown in Figure 27.2.

The output does depend on your hardware. However, you will get the file name and the line number of the error in the console window. From that you should be able to work out the problem.

27.6.4 Binding Effects

The `renderer` requires an `effect` to be bound to it for rendering to actually occur. To bind (i.e. use) an `effect` we use the following command:

```
1 renderer::bind(eff);
```

A screenshot of a Windows-style console window titled "D:\computer-graphics\solutions\graphics-framework\Debug\29 Colour Shader...". The window contains the following text:

```
GL Vendor: Intel
GL Renderer: Intel(R) HD Graphics 4000
GL Version: 4.0.0 - Build 10.18.10.3308
GLSL Version: 4.00 - Build 10.18.10.3308
LOG - renderer initialised
LOG - common content built
ERROR - could not load shader ..\resources\shaders\colour.vert
Could not compile shader file
ERROR: 0:1: '' : Incorrect GLSL version: 440
WARNING: 0:7: 'GL_ARB_explicit_attrib_location' : extension is not available in
current GLSL version
WARNING: 0:7: 'GL_ARB_explicit_attrib_location' : extension is not available in
current GLSL version
ERROR: 0:13: 'MVP' : undeclared identifier
ERROR: 0:13: 'position' : undeclared identifier
ERROR: 0:13: 'constructor' : not enough data provided for construction
```

Figure 27.2: Console Window with Shader Error

27.7 Further Reading

The two main texts for the module [1, 3] should help you in exploring shader concepts further. In particular, chapters 2 and 3 of Real-Time Rendering are a good starting point.

Lesson 28

Colour Shader

Now that we have introduced shaders, it is time to take a bit more of an in-depth look at how shaders work. We are going to do this by building an example application that applies a particular coloured effect to our object. One feature of the shader is the ability for us to set the colour to be used within our main C++ application.

28.1 A Closer Look at Shaders

So far, we have seen shaders and just ignored how they work. Now we are going to examine these in more depth. We will look at our previous simple shader and explain what is happening in the code. However, first it is worth explaining how shaders work on the GPU.

28.1.1 How Shaders Run on the GPU

Hopefully you know that a GPU is just a massive parallel processor. The advantage of the types of data that make up a 3D scene is that it can be worked on independently. For example, to create a cube we need at least 8 vertices. Instead of the GPU processing each vertex one at a time, it can process all 8 vertices simultaneously. In fact, it can do many hundreds or thousands simultaneously (depending on the GPU). This means that when you run a shader on the GPU, you are actually running many copies of the shader.

If you recall the pipeline model from Section 27.2, you can hopefully understand that data is streamed from one end of the pipeline to the other. This is the core idea of a shader - it takes in a number of inputs, processes them, and sends out a number of outputs. It is the classic input-process-output model.

$$INPUT \Rightarrow PROCESS \Rightarrow OUTPUT$$

Once you understand that a shader is working on just a small piece of incoming data, and passes this data onto the next stage of the pipeline, everything will become clear.

28.1.2 Previous Vertex Shader Example

Let us now dive back into our previous simple shader example. This is shown below:

```

1 #version 440
2
3 uniform mat4 MVP;
4
5 layout (location = 0) in vec3 position;
6
7 void main()
8 {
9     gl_Position = MVP * vec4(position, 1.0);
10}

```

Let us look at the this code line by line. Line 1 is as follows:

```
1 #version 440
```

As with C/C++, a GLSL shader can have some pre-processor declarations to tell the compiler how to treat the code it sees. Here, we are telling the compiler which version of GLSL we are targeting (4.4.0). This is the minimal version that your graphics card must support to undertake this module. Anything new will support 4.0 or above.

The next line is:

```
1 uniform mat4 MVP;
```

This line needs a little more explanation. The term `uniform` is used to describe a variable that is the same for *all* the running versions of this shader. In other words, it is a global value. Also, it is a value that the user of the shader can set.

The type of the `uniform` value is next - `mat4`. One of the reasons we are using GLM is that it uses the same types (more or less) as GLSL. Hopefully, you recognise that a `mat4` is just a 4×4 matrix - a transformation matrix.

Finally we have the name of the variable - `MVP`. `MVP` stands for *Model-View-Projection* - it is the combined model, view and projection matrices we were using before. Our render framework bundles these up and places them in the `MVP` variable. When working with shaders from now on, always remember to include the `MVP` uniform - or you won't get the output you expect.

Our next line is:

```
1 layout (location = 0) in vec3 position;
```

This line will also need a bit of explanation. We are now defining the values that are being streamed into the shader - the values used to define the geometry. The first part of the line of interest is `layout (location= 0)`. This is telling the program what position in the stream of data to expect the value in on - `position` 0. These locations are based on the buffer indices we have been using in the main application. The indices the render framework uses are shown in Table 28.1.

The next part of the declaration - `in` - tells us that the value is coming into the shader stage. The opposite of this - `out` - is for values we pass out of the shader stage and down the pipeline.

Finally we have the type of the incoming value - `vec3` for a 3-dimensional vector - and its name, `position`.

0	Positions
1	Colours
2	Normals
3	Binormals
4	Tangents
10	Texture Coordinates 0
11	Texture Coordinates 1
12	Texture Coordinates 2
13	Texture Coordinates 3
14	Texture Coordinates 4
15	Texture Coordinates 5

Table 28.1: Render Framework Default Vertex Locations

Shaders are made up of functions just as a standard C/C++ program is. Each stage of the programmable pipeline (e.g. vertex, geometry, fragment) requires a **main** function. The **main** function calls any other sub-functions accordingly.

The only operation that occurs in our simple vertex shader is the setting of the screen position of the incoming vertex. This is performed by transforming the incoming model position into the screen position, which is represented by **gl_Position**. The line

```
1 gl_Position = MVP * vec4(position, 1.0);
```

sets the **gl_Position** value by multiplying the MVP matrix by the incoming position. **gl_Position** is a value we have to set to tell OpenGL where the vertex should be positioned in screen space. Notice as well that we have to convert the position vector to a **vec4** by setting the *w* component to 1. This is because we are using a 4×4 matrix for our transform, and require a **vec4** to transform. We will talk more about homogenous coordinates in later lectures.

28.1.3 Fragment Shader Example

The previous example was for a vertex shader, which only provides one part of our current effects. Let us take a look at the simplest form of fragment shader, as shown below

```
1 #version 440
2
3 layout (location = 0) in vec4 colour_in;
4 layout (location = 0) out vec4 colour_out;
5
6 void main()
7 {
8     col = colour;
9 }
```

You might notice that this looks very familiar. This is the point of the GLSL - shaders are very similar. We have an **in** value - **colour_in** - and an **out** value - **colour_out**. All that our application does is set the **out** value to the **in** value. We are passing the incoming colour of the pixel to the outgoing colour of the pixel. We use **vec4** to represent a pixel.

28.2 Coloured Shader

We are now ready to build our coloured shader. We will look at each of the new parts in turn. First of all we have to pass in a colour for our effect, and we need to output the colour value from the vertex shader to the fragment shader. The structure of this effect is given in Figure 28.1.

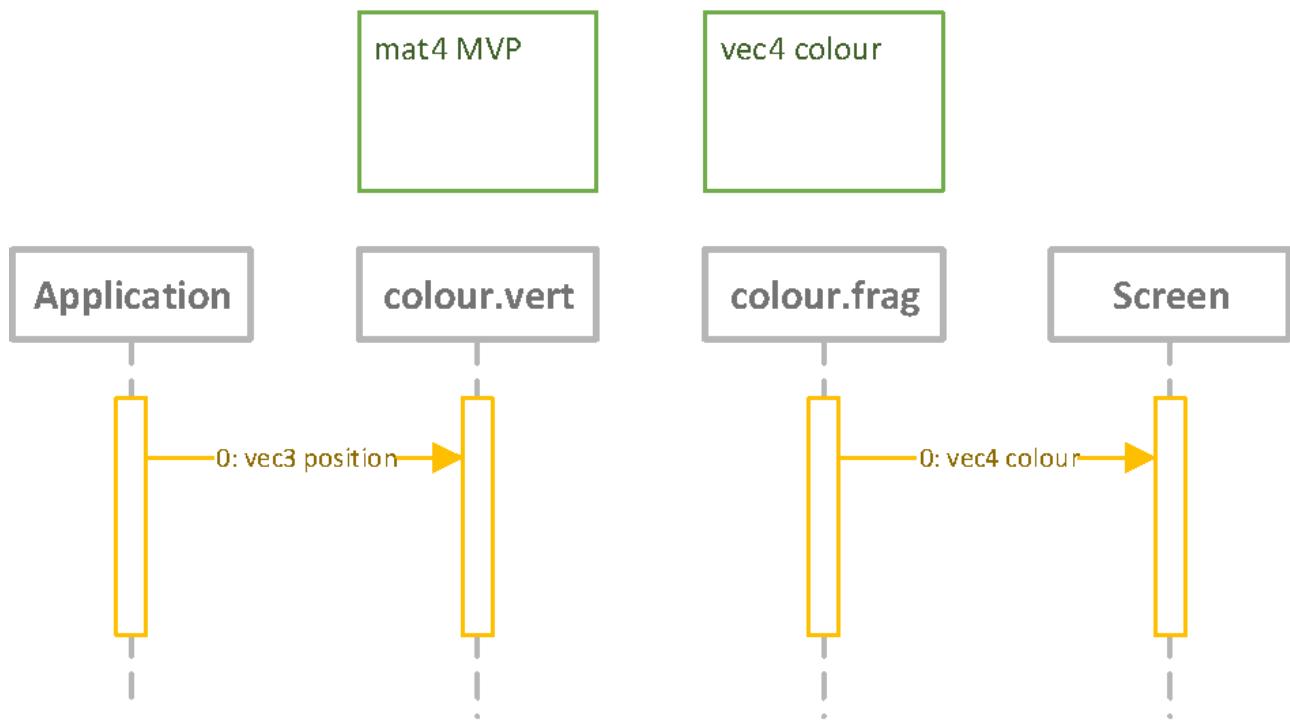


Figure 28.1: Colour Shader

Figure 28.1 is how we will present shader structures throughout the workbook. We indicate the flow of vertex data by arrows transferring from one shader stage to the next. Their location, type, and name are provided. In the green text boxes the type and name of any uniforms are provided. The shader stages are denoted by their file names.

28.2.1 Passing Values to Shaders

We already saw how we could declare `uniform` value in our shader. For our coloured effect we need to add a new `uniform` to represent the colour we want to add the following to the `colour.frag` file:

```

1 // The colour we want for our object to be
2 uniform vec4 colour;
  
```

It is as simple as that to add a new value we can set. We will use this value to calculate the colour of the vertex:

```

1 void main()
2 {
3     // Set outgoing colour
4     out_colour = colour;
5 }
  
```

28.2.2 Setting the Value in our Main Application

Setting uniform values requires us to use a number of OpenGL calls. All these calls start with `glUniform`. After this, we define the size of the value we are setting and the type. For example, to set a single `float` value we use the following call:

```
1 glUniform1f(<location>, <value>);
```

We will come to the `location` value shortly. For a 4-dimensional vector of type `float`, we would use the following:

```
1 glUniform4f(<location>, <x>, <y>, <z>, <w>);
```

This second call becomes a little bit cumbersome considering that we are using `vec4` objects. OpenGL also provides a pointer method of uniform setting. For a 4-dimensional vector we use the following:

```
1 glUniform4fv(<location>, <size>, <value>);
```

The `size` value is the number of 4-dimensional vectors we are setting. This will typically be 1, but may be larger if you want to set an array of values on the shader. The `value` has to be a pointer to the type we are using. GLM provides a helper function to get this for us - `value_ptr`. Therefore our call should look something like this:

```
1 glUniform4fv(<location>, 1, value_ptr(vec4(1.0f, 0.0f, 0.0f, 1.0f)))←  
;
```

For location, we need to discover the location that OpenGL has allocated the uniform. There are OpenGL calls to do this, but the render framework has provided a helper method to do this. The call is as follows:

```
1 eff.get_uniform_location("name");
```

`name` is the uniform name as declared in the shader. The value returned from this call is an `unsigned int` that we can use in the `glUniform` call. Therefore, our final call to set a colour uniform is as follows:

```
1 glUniform4fv(  
2     eff.get_uniform_location("colour"),  
3     1,  
4     value_ptr(vec4(1.0f, 0.0f, 0.0f, 1.0f));
```

To set a matrix value in a shader we use a similar call but with matrix added. For example, to set our MVP uniform we use the following call:

```
1 glUniformMatrix4fv(  
2     eff.get_uniform_location("MVP"), // Location of uniform  
3     1, // Number of values - 1 mat4  
4     GL_FALSE, // Transpose the matrix?  
5     value_ptr(MVP)); // Pointer to matrix data
```

You are now ready to set the colour uniform value in the shader. The `colour.frag` file needs to be updated to have a `colour` uniform. You will then need to update your main file to set the `colour` uniform value. The colour you want to set it to is green.

The output from this application is given in Figure 28.2.

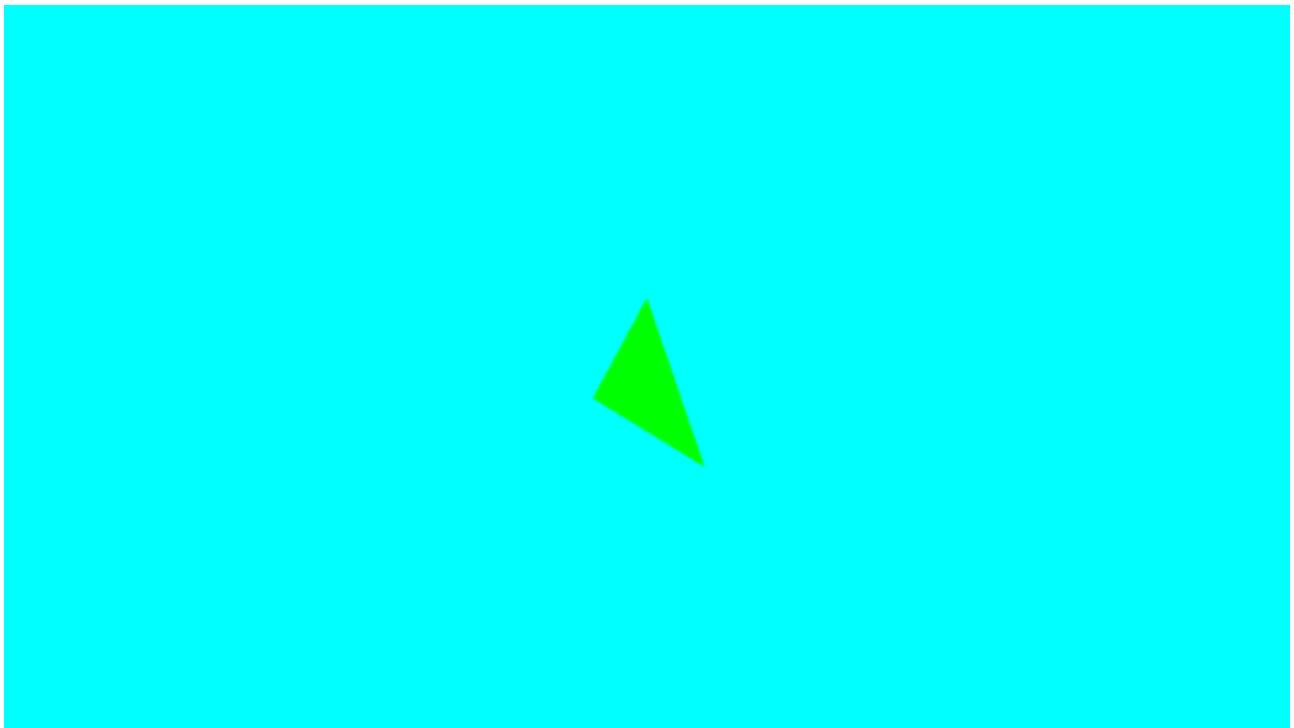


Figure 28.2: Output from Colour Shader Application

28.3 Exercises

1. Show that you understand how the value is being set by experimenting with different colour values.
2. Modify your program so that you pass in a second colour value into the shader and add this to the calculated colour value before passing it to the fragment shader. Use sensible values to test the shader code - for example red and green.

Lesson 29

Texturing

Before moving onto how we implement texturing, we will first look at how texturing works.

29.1 What is Texturing?

Texturing is the process by which we apply some form of image data to our surfaces to provide more detail. For example, we can consider a wall as just a simple geometric plane. If we attach a brick texture to the wall, all of a sudden the wall looks far more realistic. If we change the texture to that of a wooden wall, we completely change the look of the wall, but at no extra cost.

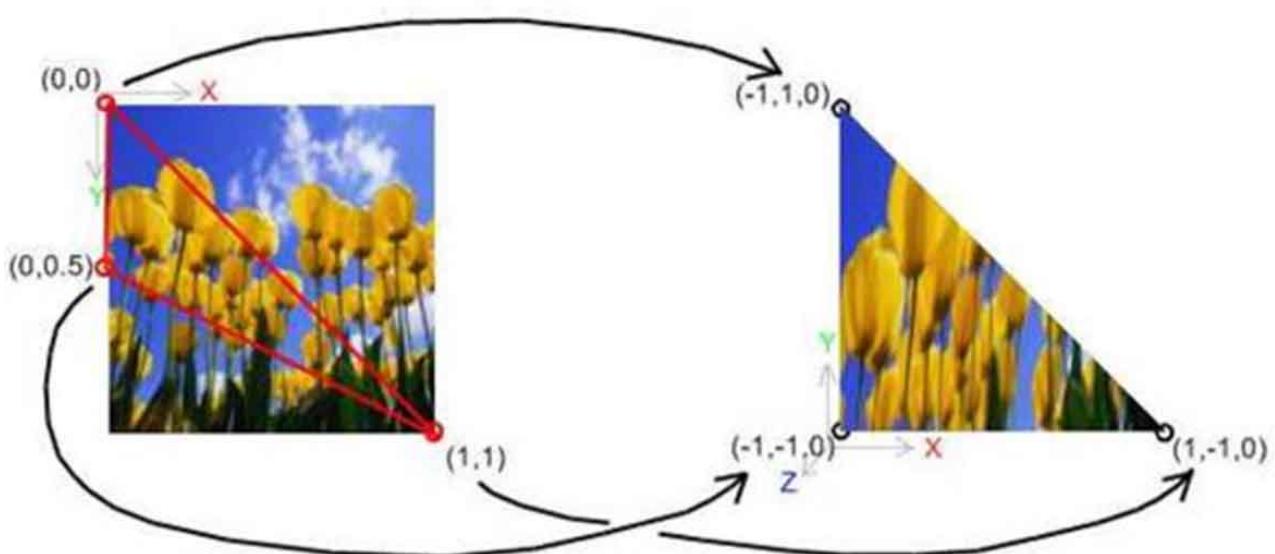
29.2 How do we Texture?

Texturing a piece of geometry requires more data to be sent to the GPU. What we need to do is send texture coordinates along with our texture data. But what are texture coordinates? Let us take a look at a typical 3D model and the texture used on it. Figure 29.1 provides an example.

We can see that the texture used on the model looks set out in a manner not useful for just displaying an image of a dragon. What we need to do is work out which parts of the texture attach to which parts of the model. This is where texture coordinates come into effect.

Each of our vertices can have texture coordinate information attached to them. These coordinates are 2D vectors which relate to a position on the texture, ranging from $<0,0>$ to $<1,1>$. We use these coordinates to create triangles of the texture to attach to the model's triangle. The fragment shader then uses these coordinates to determine our final pixel colour. Figure 29.2 illustrates this.

Texturing can be thought of as wrapping the particular texture around the defined geometry - almost like wrapping it in paper. This is not quite true, and we are doing mapping rather than mapping. Each defined piece of our geometry will have a piece of the final image attached to it. The GPU can easily work out the required piece of the image and place it on the geometry.

Figure 29.1: **Textured Object**Figure 29.2: **Texture Coordinates**

Lesson 30

Texturing in Shaders

We are now going to look at how we use texturing in shaders and in the render framework. OpenGL does not natively support the loading of textures, so we have to use a library to do so. In the render framework we use FreeImage - although you don't really need to worry about that. The main thing you need to worry about is how to use textures in your shader.

30.1 Texturing in GLSL

Texturing is done in the fragment shader in GLSL. As you should hopefully expect, we require a `uniform` value to set as our texture. For example, we use the following code in GLSL:

```
1 uniform sampler2D tex;
```

The `sampler2D` type indicates that we have a 2D texture (we can have 1D and 3D textures as well). The term sampler is used as we are sampling values from the texture at a particular coordinate.

To get a colour value from the texture we use the following code:

```
1 colour = texture(tex, tex_coord);
```

The `texture` function is used to retrieve a colour value from the texture at a particular coordinate. It takes two parameters:

`tex` The texture to be sampled

`tex_coord` The coordinate of texture to sample

This will return a `vec4` representing the colour of the texture sampled at the coordinate provided.

For the shader to work it needs to input a texture coordinate and output a colour. Visually

$$TEX_COORD \Rightarrow PROCESS \Rightarrow COLOUR$$

The shader is represented in Figure 30.1.

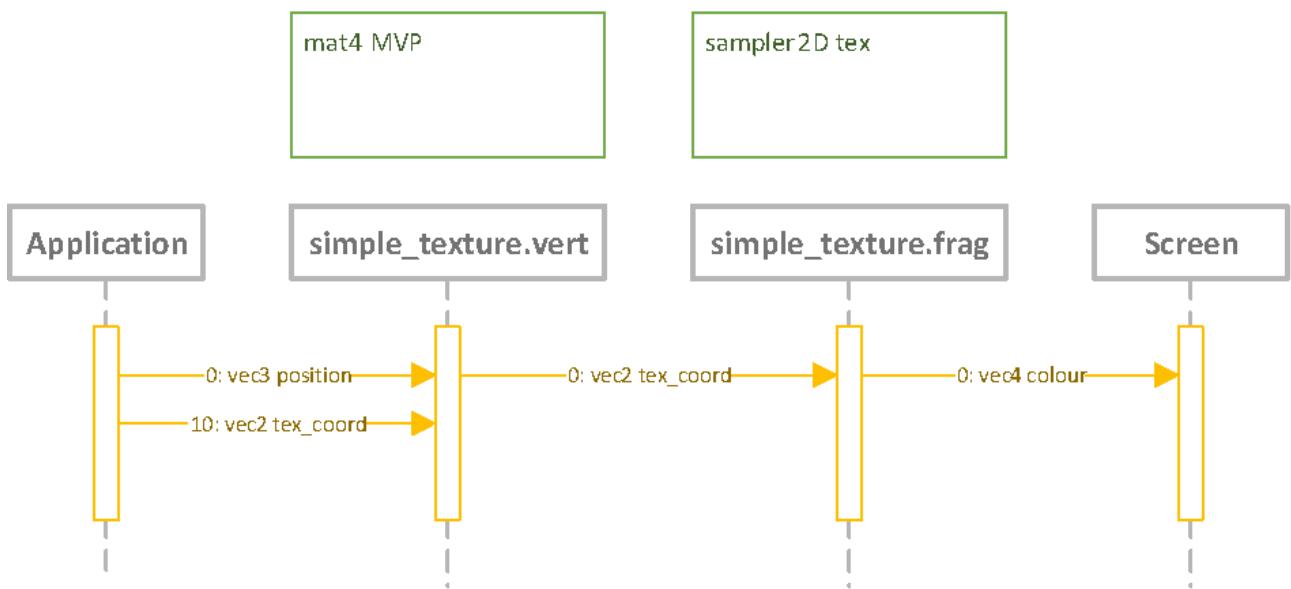


Figure 30.1: Simple Texture Shader

30.2 Using Textures in our Render Framework

The render framework takes care of most texture handling mechanisms for you, and in particular the loading of texture data into OpenGL. There are essentially three parts to the texturing process we need to concern ourselves with when texturing objects in OpenGL:

1. Adding texture coordinates to geometry
2. Loading in texture data
3. Binding and setting textures with our shaders

Let us look at these three areas in turn.

30.2.1 Setting Texture Coordinates

The `geometry` object can have various buffers attached to it as you already know from our use of colour and position data. Texture coordinates are just another buffer type - they are vectors of type `vec2`. You should be able to create a vector of this type in the main application for this project easily enough.

For adding this buffer to the `geometry` object, we use the standard `add_buffer` method, using the index for `TEXTURE_COORDS_0`:

```
1 geom.add_buffer(tex_coords, BUFFER_INDEXES::TEXTURE_COORDS_0);
```

Our `geometry` object will now have texture coordinate data associated with it.

30.2.2 Loading Textures

As mentioned, we are using FreeImage to load our images within the render framework - but you don't really need to worry about that. The render framework provides a `texture` class that can load in textures by filename:

```
1 tex = texture("filename");
```

Everything is handled by default. We will look at some other techniques for creating textures over the next few lessons. For this lesson, you will need to load the `resources\textures\sign.jpg`.

30.2.3 Setting Textures in Shaders

To actually use our textures in shaders we need to go through two processes. First, we need to bind our texture to OpenGL with a given index, and then we use this index to attach to a shader uniform. Let us look at texture binding first.

Binding Textures

Binding textures is done via the `renderer` using the `bind` method:

```
1 renderer::bind(tex, index);
```

The index is used to tell OpenGL which location to bind the texture to. Typically, we start at 0 and work upwards. Each texture that you bind for use in a shader must be given a unique index - so normally 0, 1, 2,

Setting Uniforms

A `sampler2D` has a location just as any other uniform, and has the type `int`. Therefore we use the following to attach a bound texture to a uniform:

```
1 glUniform1i(eff.get_uniform_location("tex"), index);
```

Make sure the `index` from the call to `bind` matches the one to `glUniform1i`. Therefore you should have the following:

```
1 renderer::bind(tex, 0);
2 glUniform1i(eff.get_uniform_location("tex"), 0);
```

If you run this application you will get the output shown in Figure 30.2.

30.3 Exercises

1. Try working with different textures just to get comfortable with loading shaders.
2. Try different geometry types to see how the texturing works for them as well.
3. You should now be ready to add multiple objects to your scene. You will need to set their transforms so they are not in the same location and remember to render them all. Try and have five different objects in your scene.

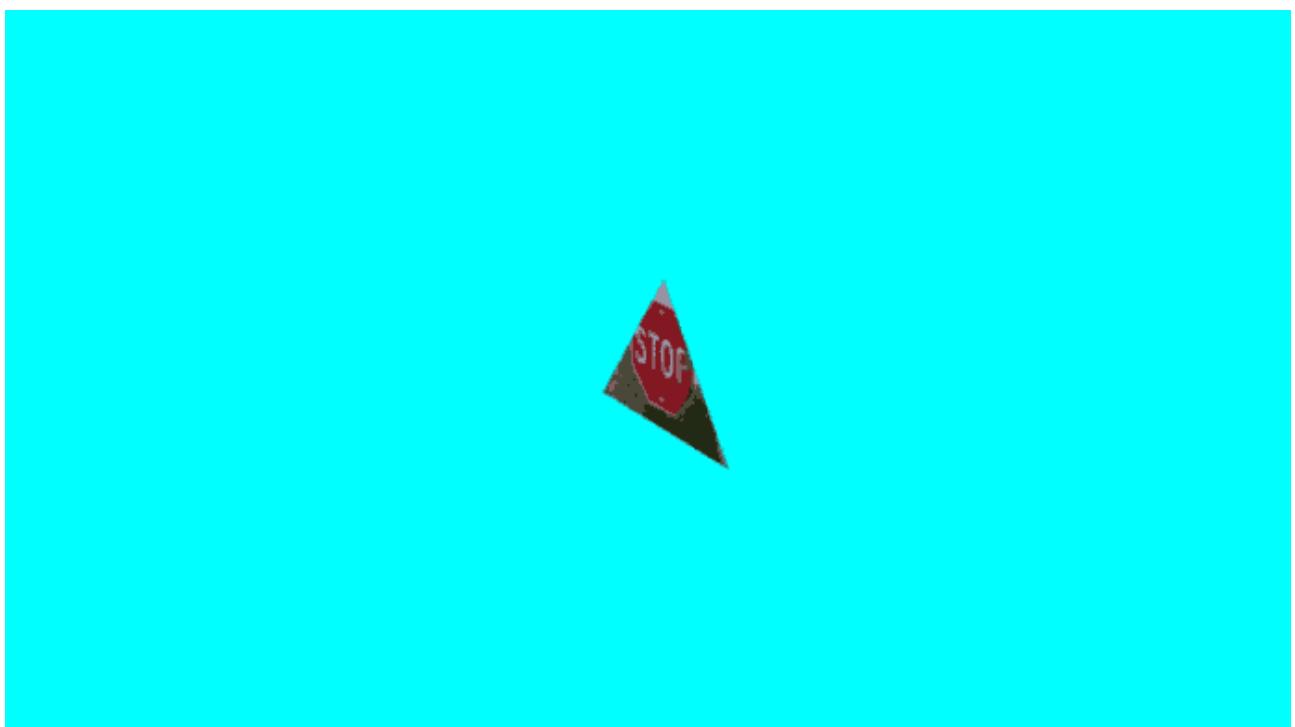


Figure 30.2: **Textured Object**

Lesson 31

Mipmaps

Now let us look at how textures are handled based on their distance from the viewer (scaling). OpenGL does this using concepts called minification and magnification. The concept we are going to use to support this are mipmaps.

31.1 Minification and Magnification

We have two possible scenarios we want to magnify our texture or we want to minify it. If we look at the magnification, we have a particular problem to deal with as shown in Figure 31.1.



Figure 31.1: Magnification

As we magnify an image, we do not gain information. This is why our images can look fuzzy. Magnification is actually unlikely, and we are more likely to see minification occurring. For example, see Figure 31.2.

The top picture shows the minification problem. The second image uses mipmaps. The third anisotropic filtering (we will look at anisotropic filtering in the next lesson). Let us look at mipmaps in more detail. A mipmap basically introduces scaling based on how far away an image is by producing multiple versions of the image and different sizes. Each image is scaled in half to provide a new image (so for example 512×512 produces 256×256 produces 128×128 etc. This is why we need power of two texture sizes). Figure 31.3 should help.

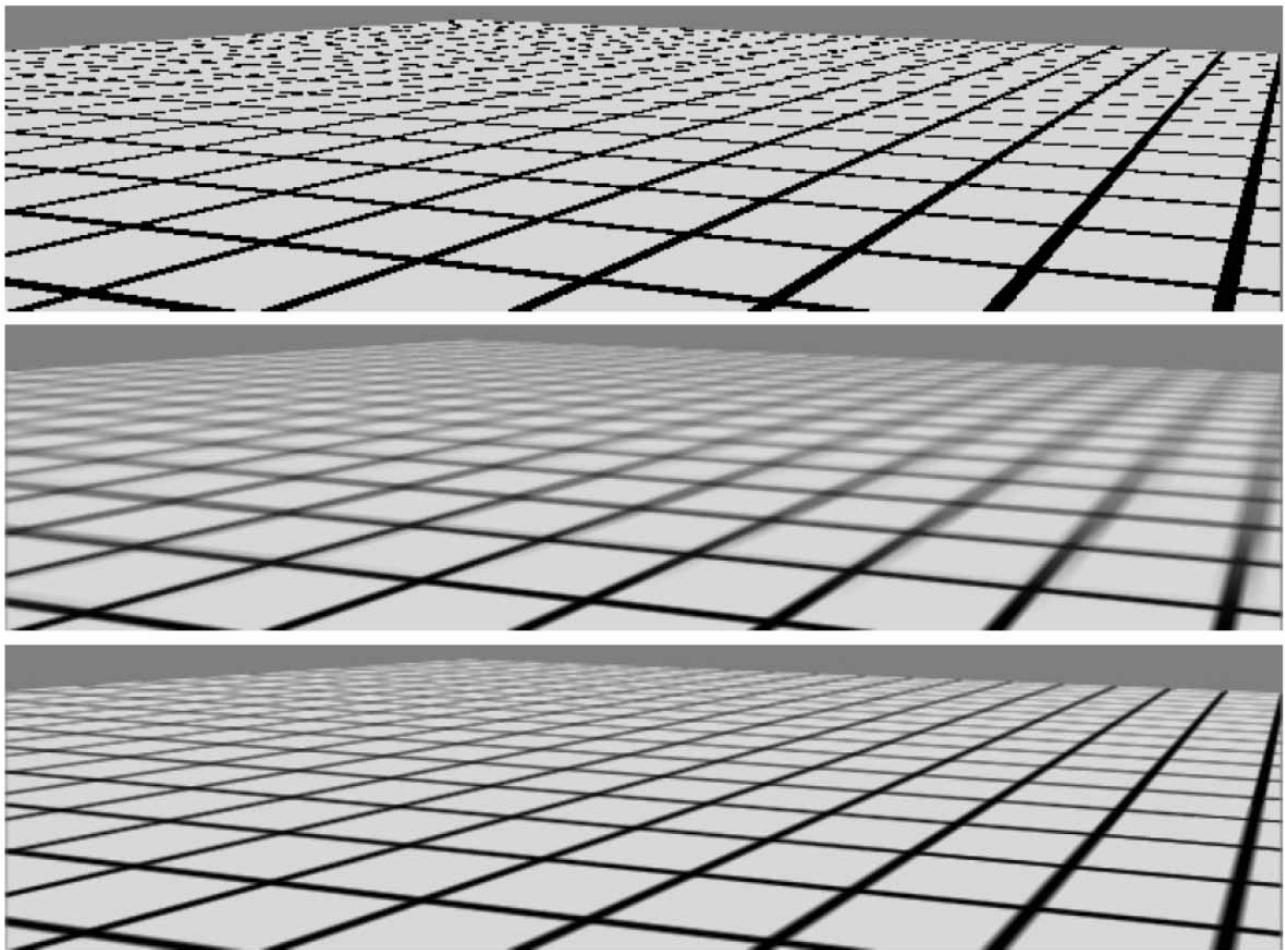


Figure 31.2: Minification

What we need to do is get OpenGL to do this for us. This is actually quite easy - the rendering framework has this built in and you have been using mipmaps all the time. Let's see the difference they make when switched off.

31.2 Mipmap Test Application

This application will require some work from you. The output you are aiming for is shown in Figure 31.4 (use up and down arrows to move forward and back).

You should immediately notice the difference between the image on the left (no mipmaps) and the one on the right (mipmaps on).

Your task is to recreate this application. You only need to use the standard texturing shader. What you do need to know about is how to tell the texture loader to use mipmaps or not. This is done by using the standard load operation but with a extra parameters:

```
1 tex = texture(filename, mipmaps, anisotropic);
```

We are only interested in the `mipmaps` parameter at present. This is a boolean value that you set to `true` for mipmaps and `false` for no mipmaps. In both cases `anisotropic` should be set to false (we will look at that next lesson). Therefore you need to load a texture twice - once with mipmaps off and once with them on.

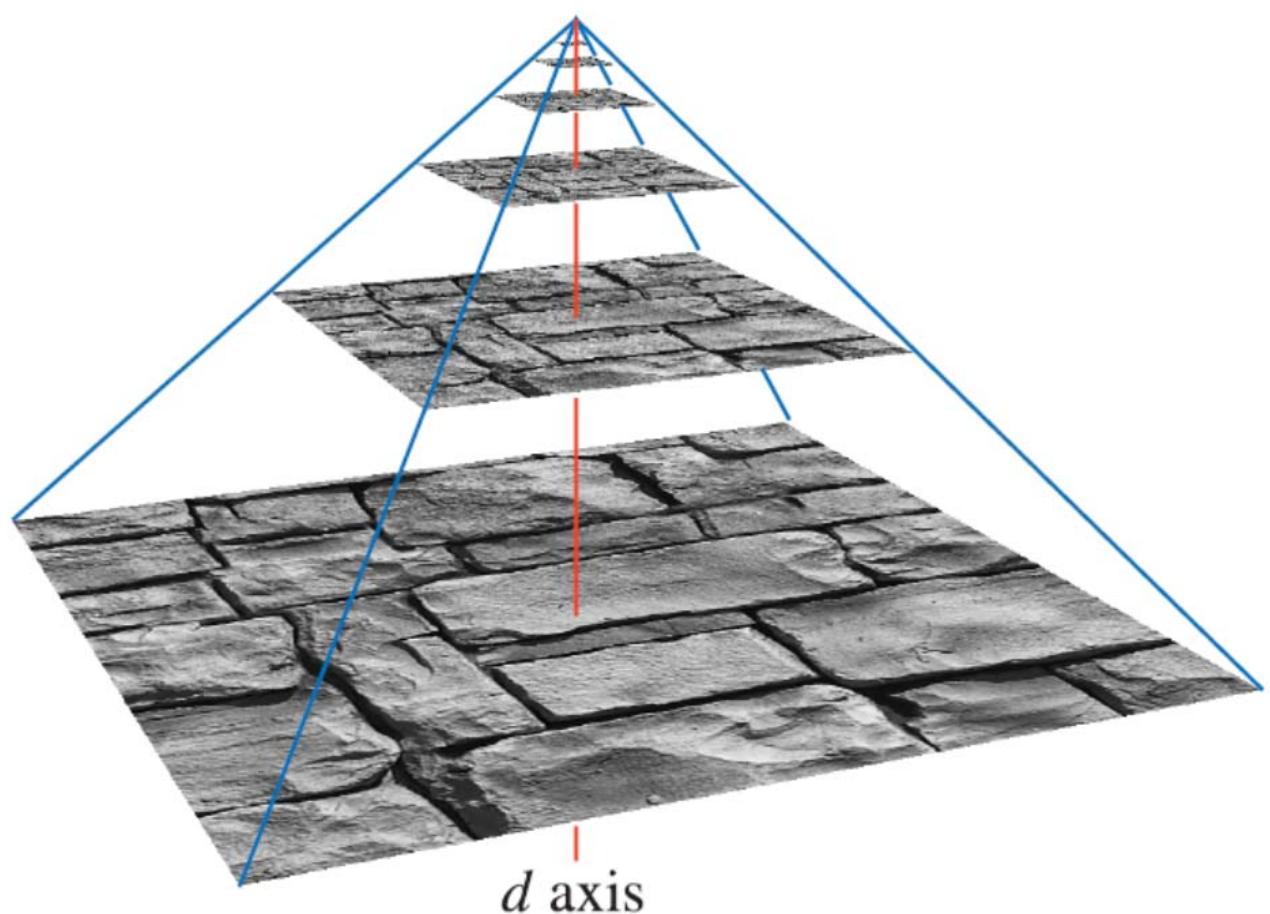


Figure 31.3: **Mipmaps**

Your task is to get the meshes correct to replicate the output. Good luck!

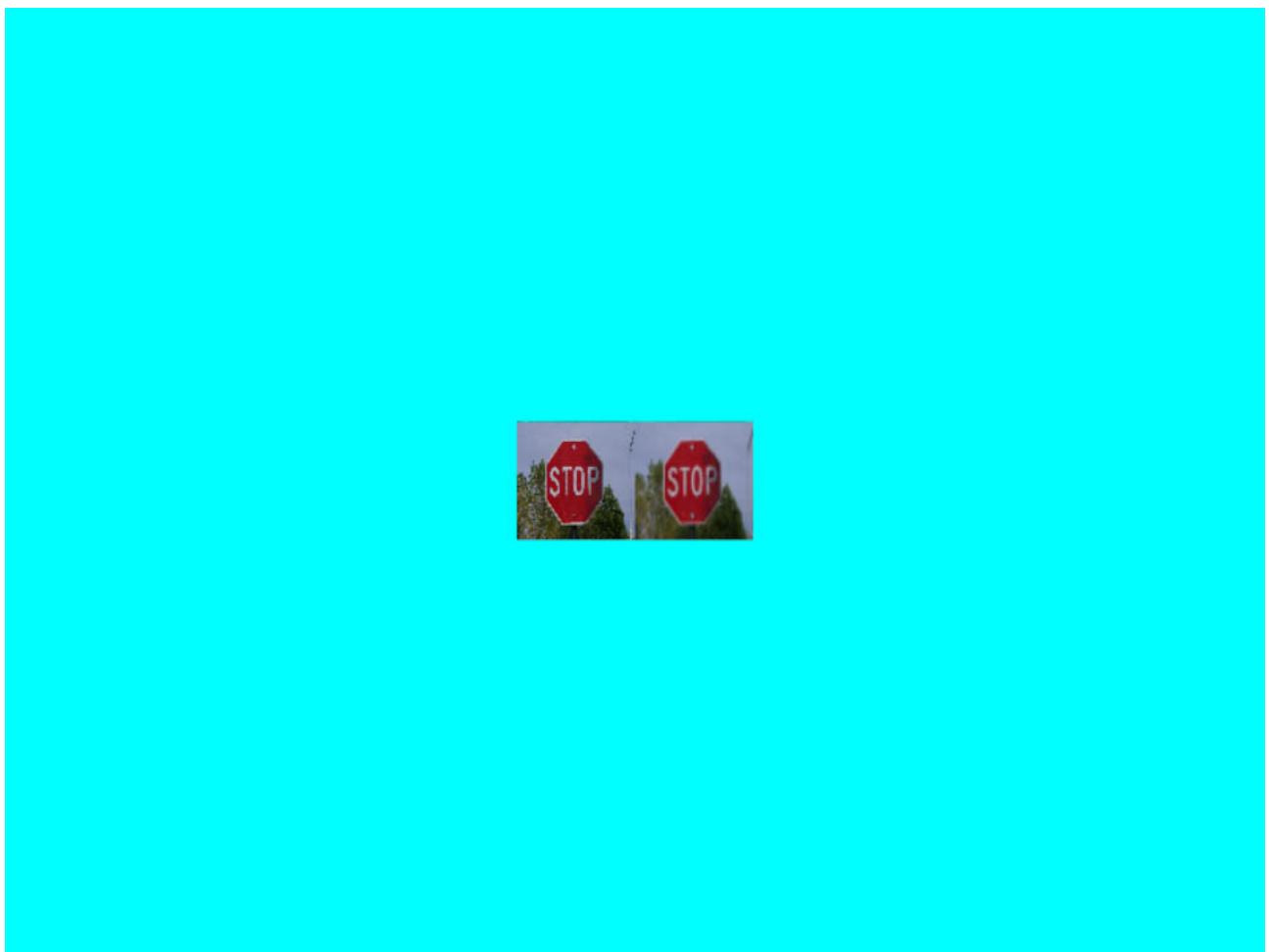


Figure 31.4: Mipmap Test Application Output

Lesson 32

Anisotropic Filtering

Anisotropic filtering allows us to deal with the problems caused by mipmaps when looking at objects at an oblique angle (in other words, not straight on). This is a useful technique to avoid what we saw in the previous lesson, and is quite easy to do - we already covered it in the last lesson.

Your goal this lesson is to recreate the output shown in Figure 32.1.

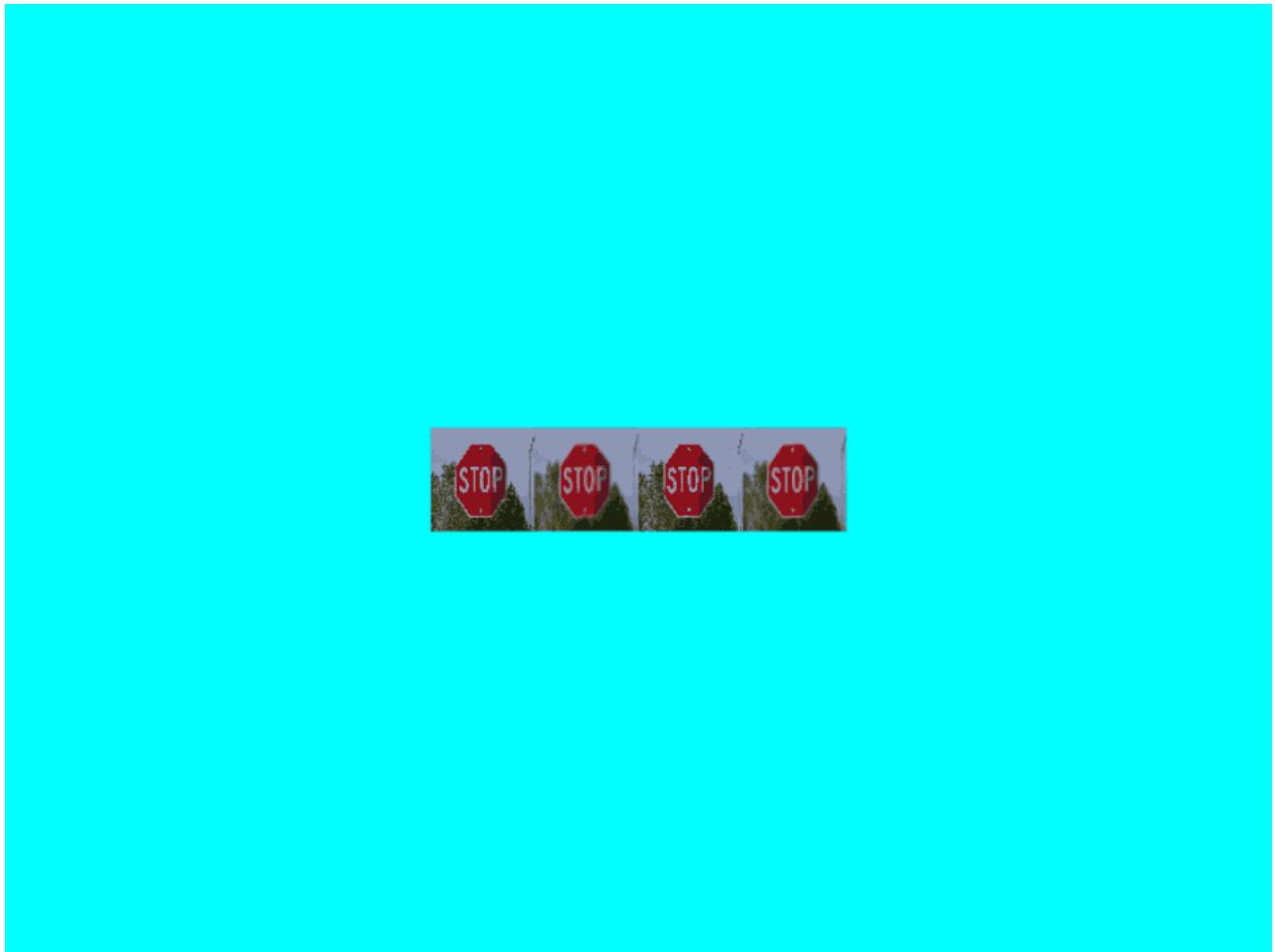


Figure 32.1: Anisotropic Test Application Output

In order from left to right we have:

1. Mipmaps off, anisotropic off

-
- 2. Mipmaps off, anisotropic on
 - 3. Mipmaps on, anisotropic off
 - 4. Mipmaps on, anisotropic on

You probably won't see much difference with having mipmaps and anisotropic - but there is some.

Lesson 33

Multi-texturing

One of the most powerful techniques we can use when texturing is multi-texturing. This involves us taking two or more textures, then using some form of blend weight to determine how much of the texture colour we want to use at that particular vertex. Figure 33.1 should help to illustrate.

Notice that the side nearest the camera has more grass associated with it. This is just applying more of the texture colour from the grass texture at this point rather than the checked texture.

33.1 Blend Maps

Normally, we use another texture to help us determine what percentage of each texture colour to use. This texture is sometimes called a blend map or we can use an alpha map, and can look a little bit like Figure 33.2.

Based on the grayscale or alpha value of the texture, we use the amount from texture 1 or texture 2. It is really that easy, and all we need to do is use three textures in our shader. Let us look at doing this now.

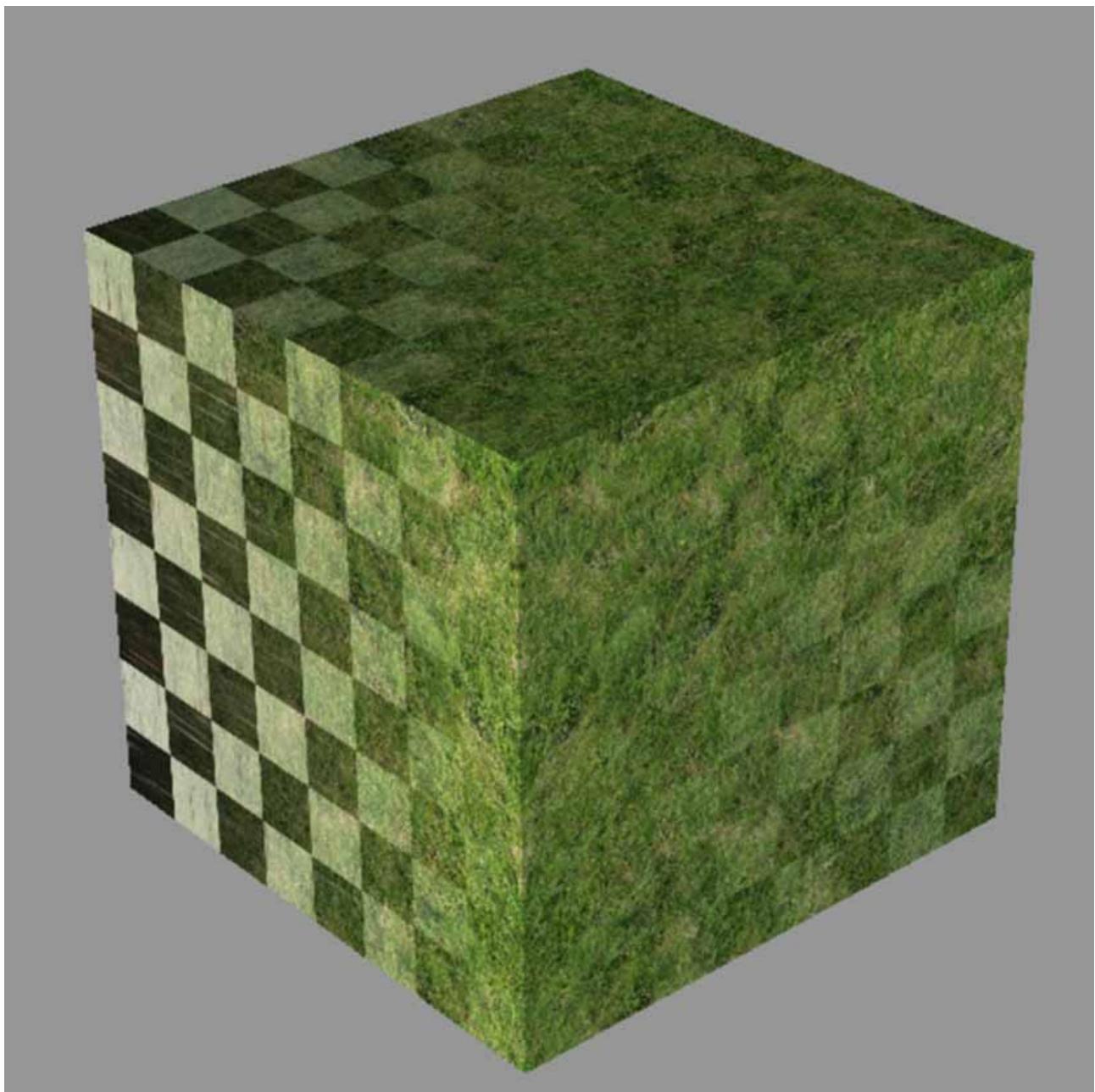


Figure 33.1: Multi-texturing



Figure 33.2: Blend Map

Lesson 34

Blended Textures

We are going to add a new shader file to our collection one to determine the colour of a pixel by blending two textures together using a third blend map. Let us create this shader file first.

34.1 Blend Shader

We already know how we can sample a colour from a texture in GLSL using the `texture` function.

```
1 vec4 colour = texture(sampler, coordinate);
```

Our goal this time is to take samples from two different textures and then blend them together using a floating point value (ranging from 0.0 to 1.0) to determine how the colours are mixed together. To do this we use the `mix` function.

```
1 colour = mix(sample1, sample2, factor);
```

`mix` takes two colour (`vec4`) values and a floating point factor. If the `factor` = 0.0 then `colour` = `sample1`. If `factor` = 1.0 then `colour` = `sample2`. The values between are a mix of the colours accordingly.

The shader for blending is shown in Figure 34.1.

The pseudocode for the algorithm is shown in Algorithm 7.

Algorithm 7 Blending Shader

```
1: procedure BLEND(tex_coord, tex1, tex2, blend_map)
2:   col1 ← TEXTURE(tex1, tex_coord)
3:   col2 ← TEXTURE(tex2, tex_coord)
4:   blend ← TEXTURE(blend_map, tex_coord)
5:   colour ← MIX(col1, col2, blend.r)
```

Notice to mix we are using the red value from the texture. We could have used the alpha, green or blue values instead.

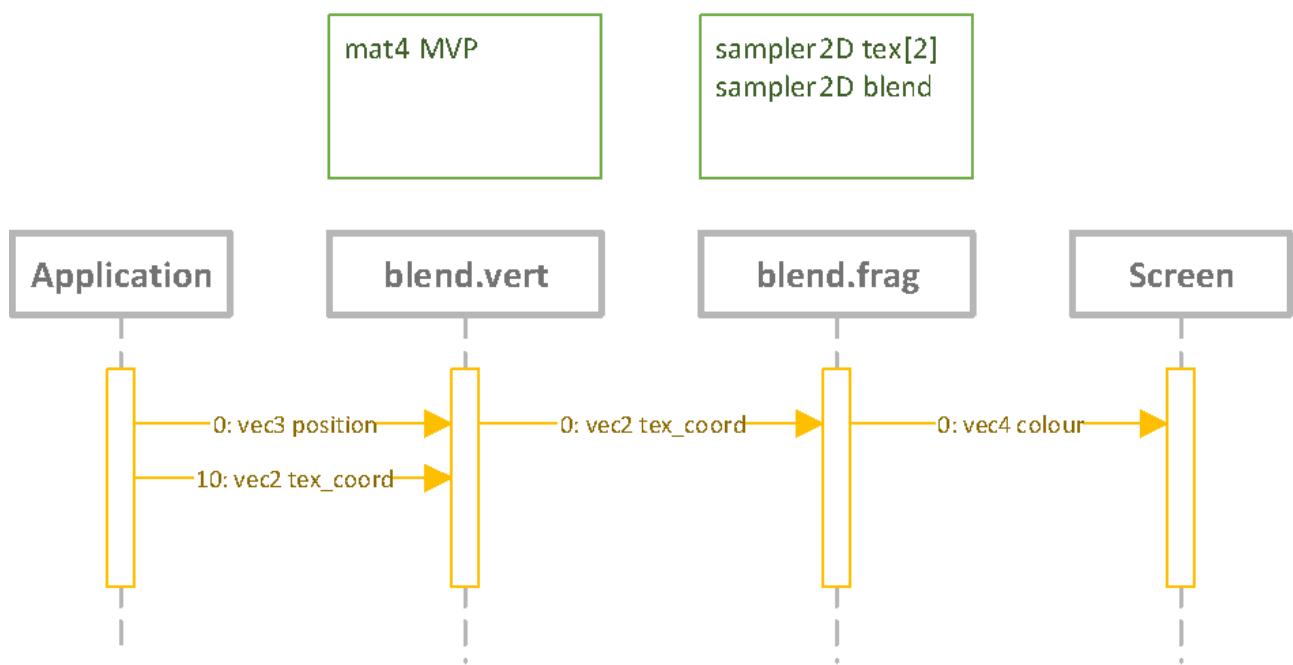


Figure 34.1: Blend Shader

Your task for this lesson is to get the blend shader working. You have enough information to do this now. The files you will need to load are in the resources folder and called `grass.png`, `stonygrass.jpg` and `blend_map.jpg`.

To set the texture array uniform in the shader you have two choices. You can set each texture individually using the uniform names `tex[0]` and `tex[1]`. Or you can set them in one call using an array containing the two index values.

If you run this application you should get the output shown in Figure 34.2 (depending on the textures and camera you have used).

34.2 Excercise

1. Try and use some of the other colour values for your blend factor, and experiment with other blend maps.
2. You have four different colour values from a blend map that you could use as blend factors. Try and create a shader that uses 4 textures as input and blends them together using the four colour values of a texture. A simple blend map (i.e. black and white with transparency) will not do here - you will need one with colours.

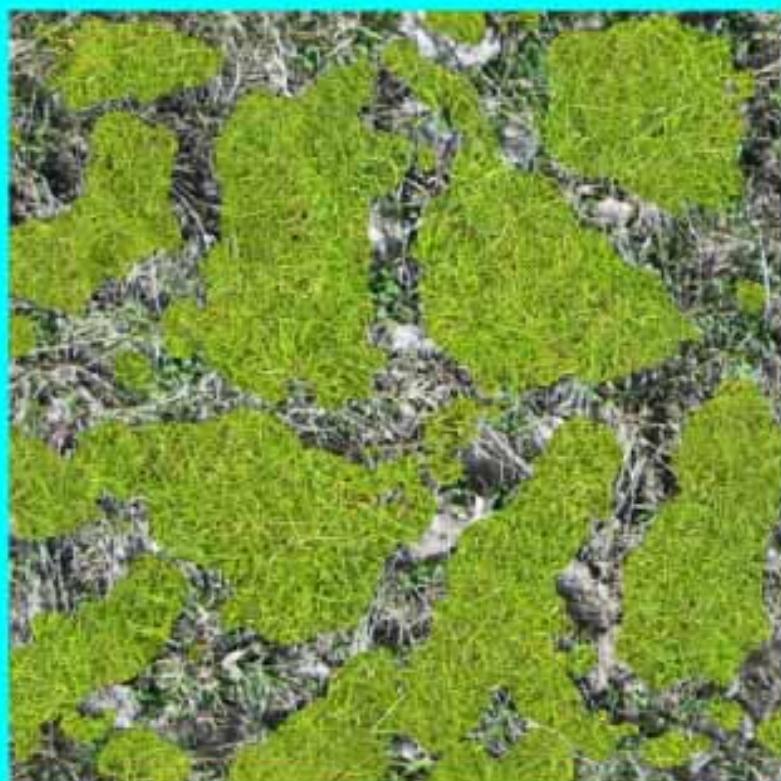


Figure 34.2: Blended Textures on a Cube

Lesson 35

Dissolve Shader

Let us carry on our work with texturing by implementing a dissolve shader. Our dissolve effect will utilise a new command in the fragment shader which allows us to tell the GPU not to draw a fragment.

35.1 Discarding Fragments

We will need to use a new GLSL command to allow us to discard (i.e. not render) a particular fragment. This command is actually very simple.

```
1 discard;
```

If during the execution of our fragment shader we encounter this command the fragment is discarded and the rest of the fragment shader code is not executed. It is that simple. If our fragment shader only had this command, then nothing would be rendered.

35.2 Shader Structure

Figure 35.1 presents the structure for the dissolve shader.

The structure is basically the same as any of our texturing shaders (hopefully you are spotting the pattern). Therefore, the vertex shader is just the same as we have had already. The fragment shader is where we determine if we should be discarding a fragment.

35.2.1 Fragment Shader

Our fragment shader is again where we do our actual effect work. We need to sample a blend map that can act as our dissolve texture and compare it to a dissolve factor. If the red component of the map is greater than our factor, then we discard it. The pseudocode for the algorithm is given in Algorithm 8.

You will need to update the fragment shader and the main application for this lesson. The `dissolve_factor` value is changed using up and down. This will dissolve part of the object accordingly. If you run this application you will get the output shown in Figure 35.2.

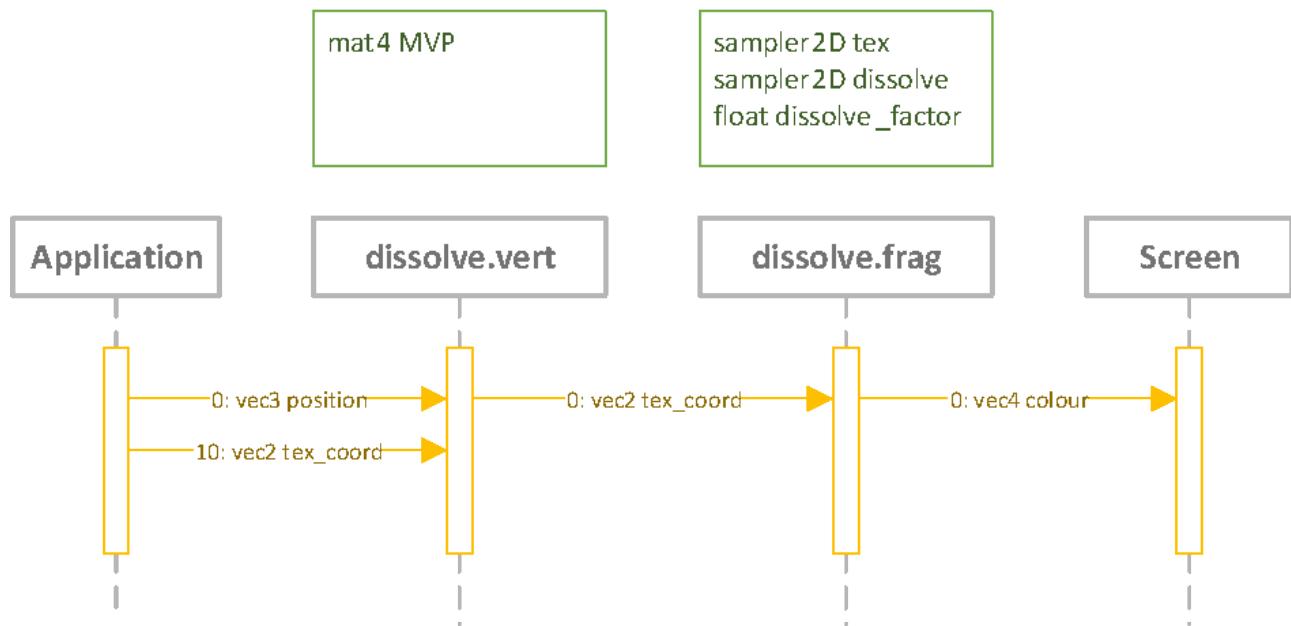


Figure 35.1: Dissolve Shader Structure

Algorithm 8 Dissolve Shader

```

1: procedure DISSOLVE(tex_coord, tex, dissolve, dissolve_factor)
2:   dissolve_factor  $\leftarrow$  TEXTURE(dissolve, tex_coord)
3:   if dissolve_value.r  $>$  dissolve_factor then
4:     discard
5:   colour  $\leftarrow$  TEXTURE(tex, tex_coord)
  
```

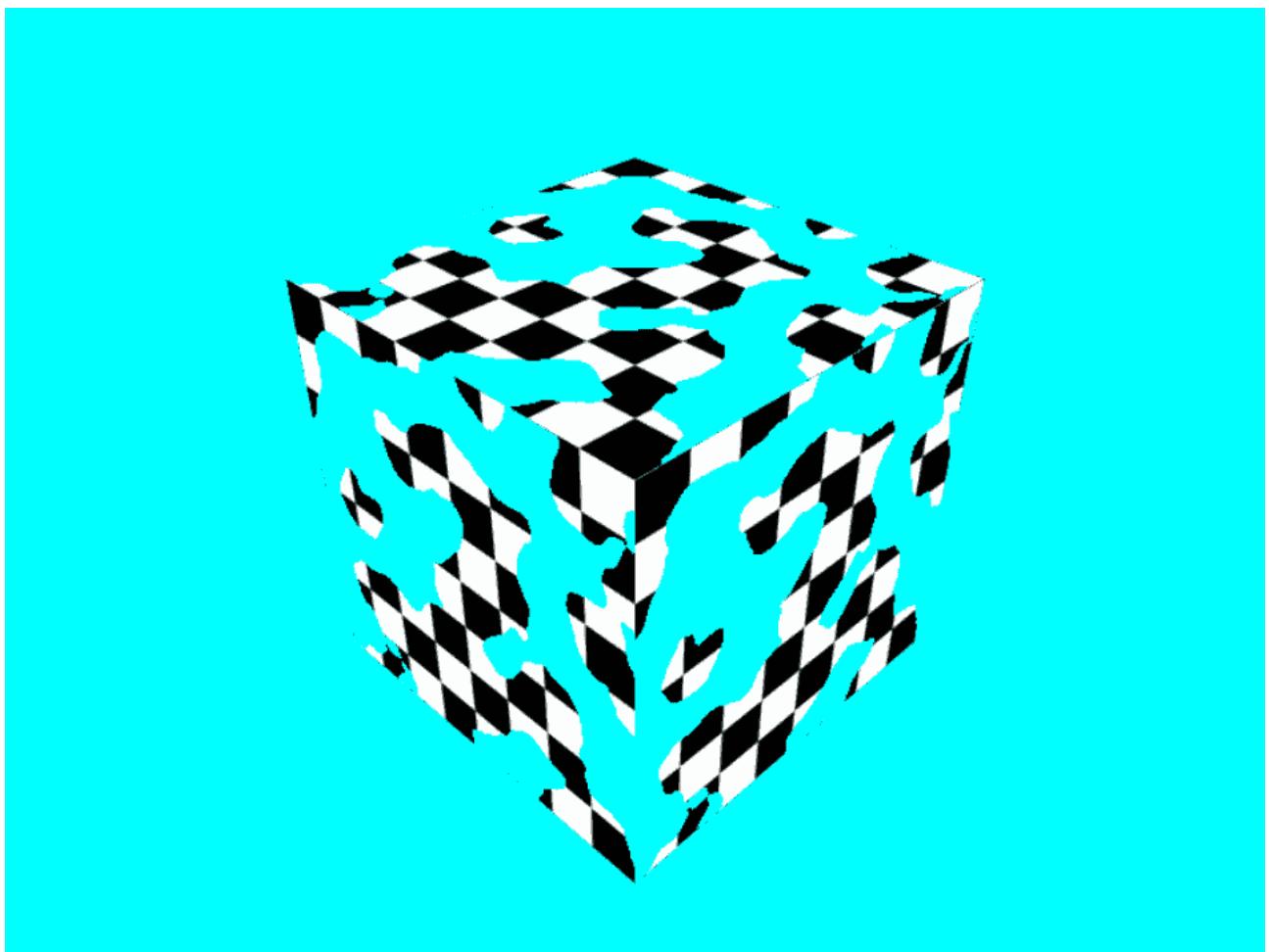


Figure 35.2: Dissolve Shader Output

Lesson 36

Simple Cell Shading

We are now going to use a 1D texture. A 1D texture is essentially just a line of pixels rather than a 2D image. What we are going to use it for is to perform a simple cell shading approach to our render.

36.1 1D Textures

A 1D texture is a texture that only has width - the height is set to 1. This means that we have to treat the data type differently in GLSL - we need a `sampler1D` instead of a `sampler2D`.

```
1 uniform sampler1D tex;
```

We still use the `texture` function to sample the texture. However we now need to use a single floating point value to sample the texture rather than a 2D vector.

36.2 Cell Shader

Our vertex shader is where we will be doing our work this time. You don't have to worry at the moment what the vertex shader is doing - this is a lighting style calculation we will cover soon. For the moment `simple_cell.frag` is all you have to worry about.

The overall structure of the cell shader is provided in Figure 36.1.

$$\text{position, normal} \Rightarrow \text{tex_coord} \Rightarrow \text{colour}$$

The texture coordinate coming into the fragment shader this time is just a floating value. We can use this to get a sample from the 1D texture using the `texture` function. You will need to add a line to do this to the definition of `simple_cell.frag`.

36.3 Creating Textures from Colour Data

We are going to look at how we can generate a texture from a set of colour data. The render framework has a method that will take a collection of colour data and create a texture object

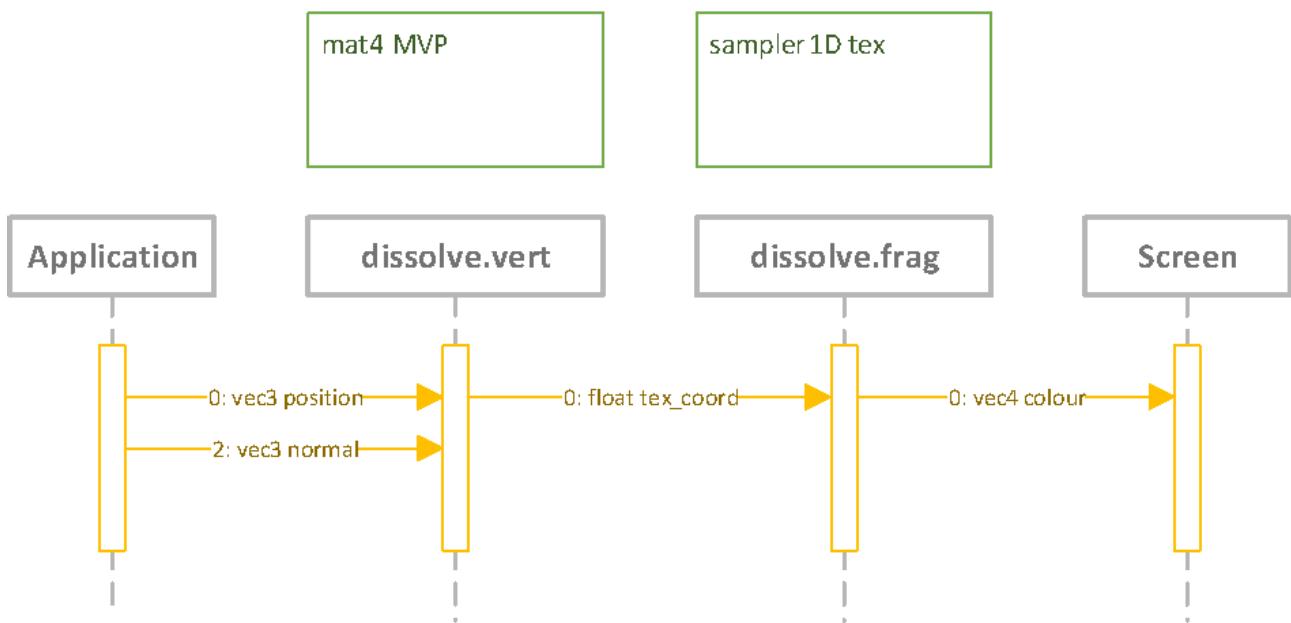


Figure 36.1: Simple Cell Shader

for us. This is how we are going to generate our texture. The line of code we use is:

```
1 tex = texture(colour_data, width, height);
```

In particular, for this particular application you will want to specify that we are not using mipmaps or anisotropic filtering, so your actual call will be:

```
1 tex = texture(colour_data, width, height, false, false);
```

This method takes three values:

`colour_data` the colour data that makes up the texture. This is just a `vector<vec4>`.

`width` the width of the texture to generate

`height` the height of the texture to generate

For this application we will use a texture with four colour values. These values are:

```

< 0.12, 0.0, 0.0, 1.0 >
< 0.25, 0.0, 0.0, 1.0 >
< 0.5, 0.0, 0.0, 1.0 >
< 1.0, 0.0, 0.0, 1.0 >
  
```

If you run this application you will get the output shown in Figure 36.2.

36.4 Exercises

Experiment with the cube to see what it looks like when it is rendered with the cell shader.

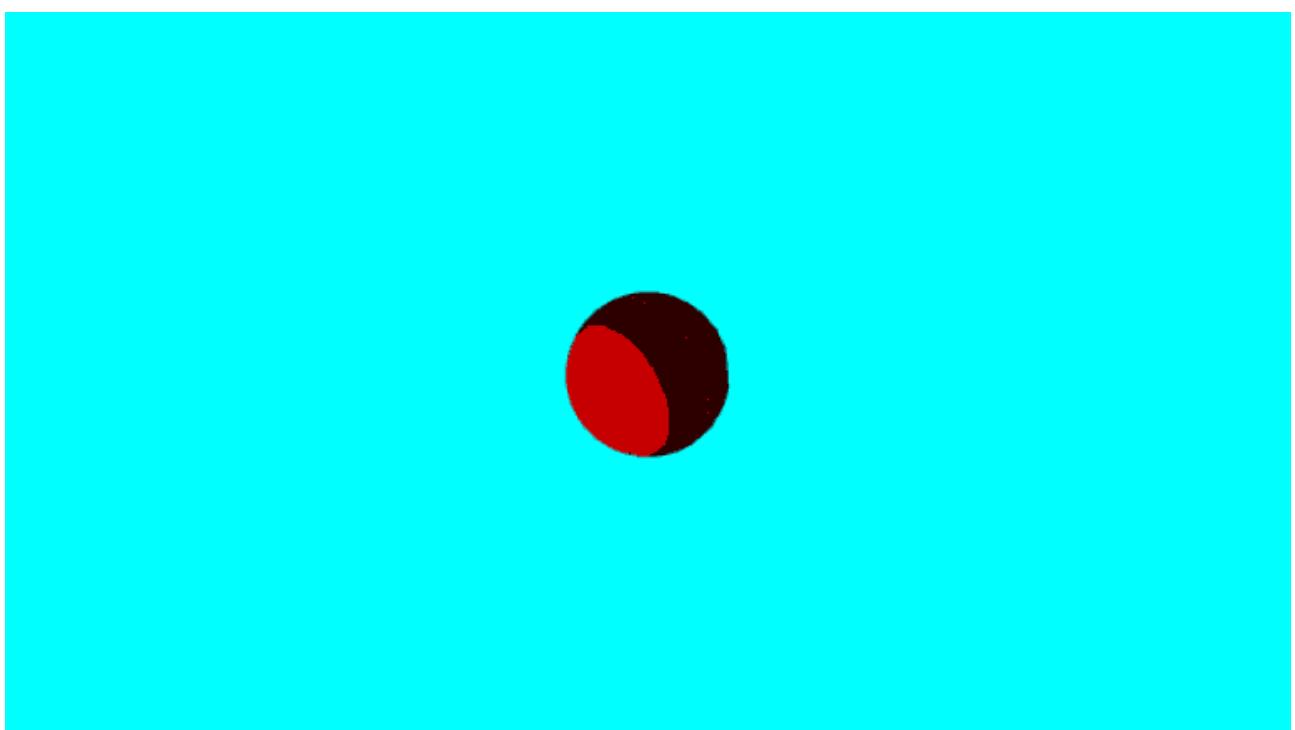


Figure 36.2: **Cell Shading with a 1D Texture**

Lesson 37

Repeat Textures

We are now going to finish our work on texturing by looking at how we can make our textures repeat across a surface. By repeat, we mean that the texture has a tiling like effect, allowing us to reuse a texture across a surface if we wish. This is a useful technique when you wish to texture large surfaces such as floors and walls.

37.1 Non-uniform Texture Coordinates

So far we have only dealt with uniform texture coordinates (those ranging from [0, 0] to [1, 1]). What happens when we have texture coordinates outside these ranges? Well, it depends what we told OpenGL to do when we loaded in the texture. OpenGL provides 4 options:

GL_REPEAT the texture is repeated uniformly with the texture coordinates. This is the default behaviour.

GL_MIRRORED_REPEAT the texture is repeated, but is mirrored on the repeating axis.

GL_CLAMP_TO_EDGE the colour on the edge of the texture is continued along non-uniform coordinates.

GL_CLAMP_TO_BORDER a defined border colour is used for non-uniform coordinates.

Figure 37.1 (taken from *Real-Time Rendering* [1]) illustrates the different approaches to non-uniform texture coordinates.

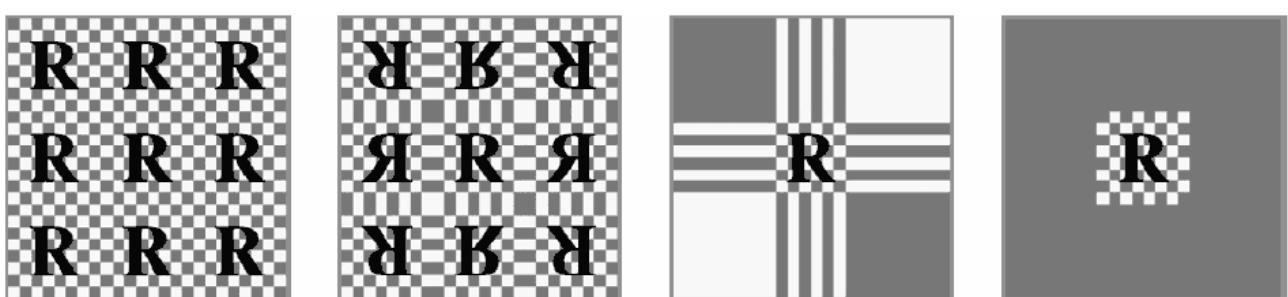


Figure 37.1: Non-uniform Texture Coordinates

In the graphics framework, textures are loaded with the default behaviour (**GL_REPEAT**). You can change this behaviour if you know OpenGL enough.

37.1.1 Defining Non-uniform Texture Coordinates

So now that we know how OpenGL deals with non-uniform textures, how do we go about using non-uniform texture coordinate behaviour. Well, all we need to do is define some non-uniform texture coordinates. Figure 37.2 illustrates (ignore the white spaces - they are for illustration purposes).

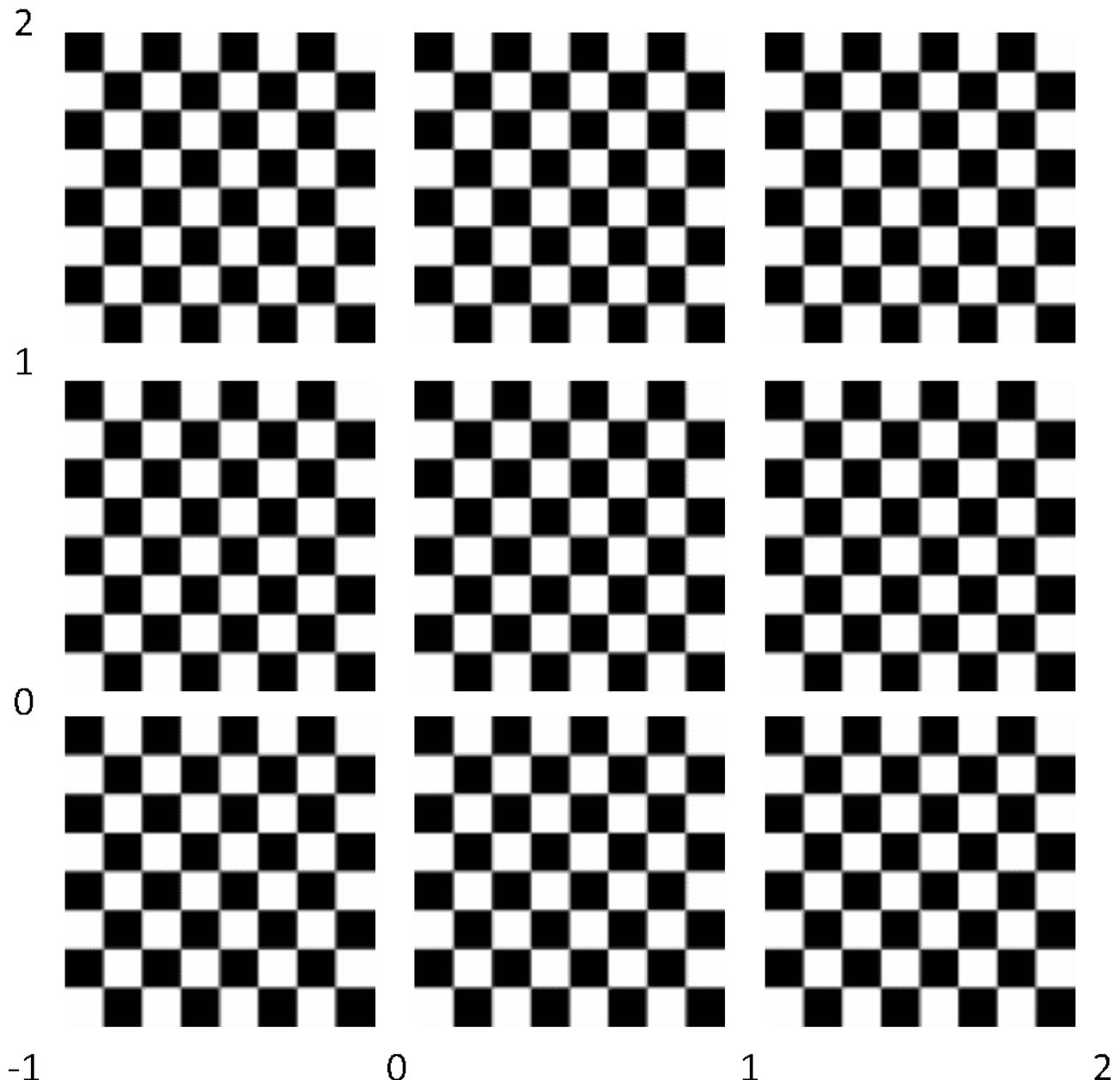


Figure 37.2: **Repeating Texture Coordinates**

Instead of defining our texture coordinates for the four corners as (in counter-clockwise order from the top-right):

1. (1, 1)
2. (0, 1)
3. (0, 0)
4. (1, 0)

We use the following texture coordinates:

1. $(2, 2)$
2. $(-1, 2)$
3. $(-1, -1)$
4. $(2, -1)$

OpenGL will then interpret these as non-uniform texture coordinates and act accordingly.

37.2 Updating Main

The main application in this lesson requires you to define the non-uniform texture coordinates for the defined quad, as well as ensuring you remember how to load and use textures. Once you have completed this lesson, you will get an output similar to that shown in Figure 37.3.

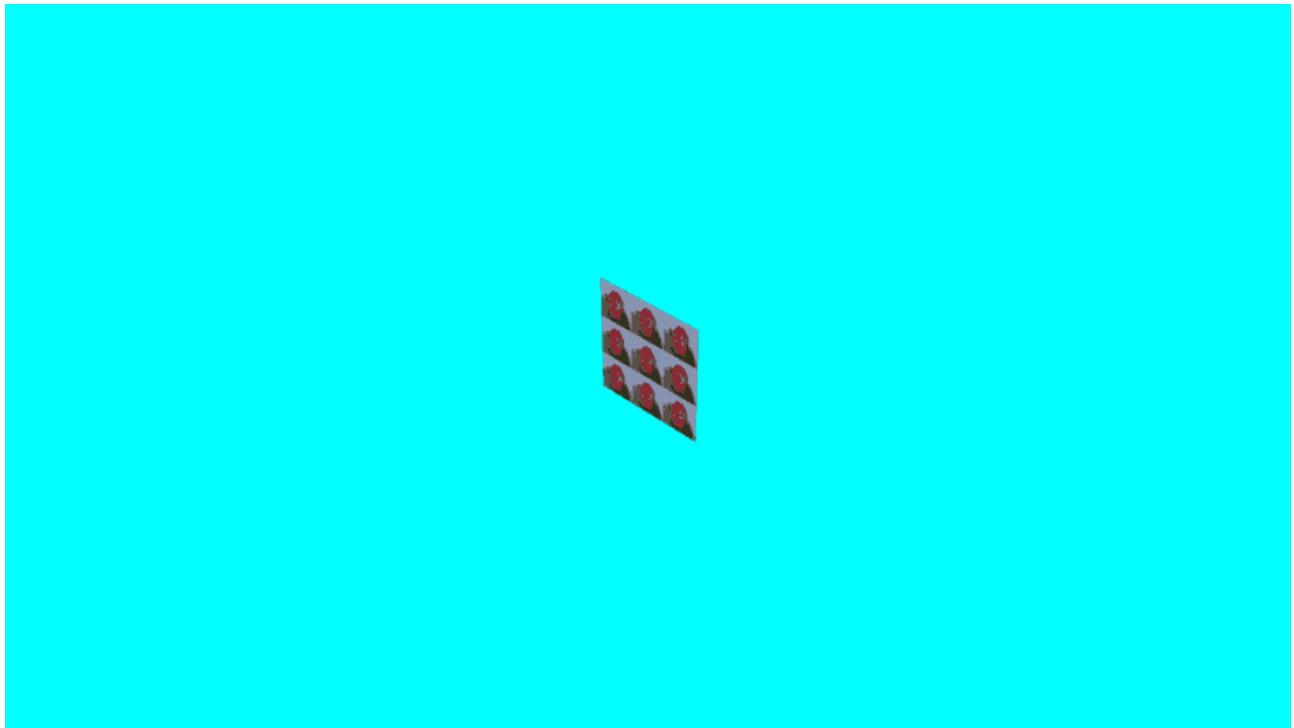


Figure 37.3: Output from Texture Repeat Lesson

Lesson 38

Working with CodeXL

CodeXL is a tool to analyse OpenGL applications (and OpenCL ones as well). If you do a Google search you will be able to find the download. The tool is provided by AMD. If you have Visual Studio 2013 then CodeXL will also act as a plugin. This is how we will work with CodeXL although it is fairly easy to work out how we are using CodeXL if you are using the stand alone application.

38.1 Starting CodeXL

When you start Visual Studio with CodeXL installed, you will see that there is a menu item for CodeXL at the top. This will allow you to run an application via CodeXL debugging. Figure 38.1 illustrates.

Not much to see here. The interesting buttons at the moment are the various run, step and pause buttons (the green arrows, etc.). If you click the start button, you will see your application run. Pause the application, and click the **draw step** button until the final image is displayed on the screen.

38.2 Analysing Buffers

On the left of the Visual Studio window, alongside the **Solution Explorer** you will find the CodeXL explorer. You may have to enable this via the **View** entry of the CodeXL menu. Figure 38.2 illustrates.

This shows you the current textures and buffers loaded into the application. You can also explore some of the other buffers. This includes all the buffers we have created so far in the application (just geometry at the moment). You should try and determine what all the buffers are, and explore the information given. This can be an invaluable tool to determine if your geometry is correct. You should be able to tell which buffer is the position data for the quad quite easily.

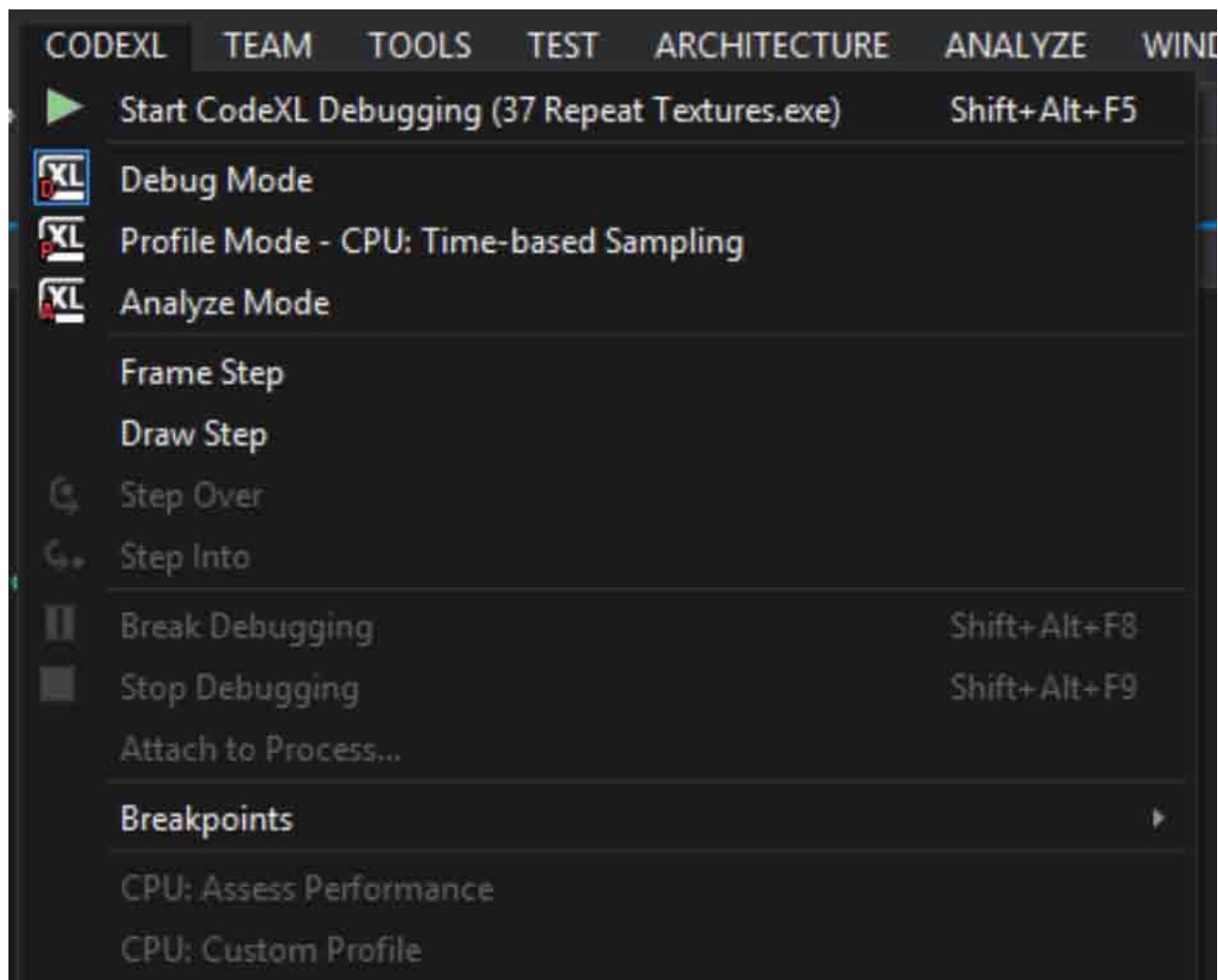


Figure 38.1: CodeXL Menu in Visual Studio

38.3 Shaders Viewer

The final viewer we will look at is the **Shaders and Kernels Source Code Editor**. This view allows us to see the loaded in shaders, and the current bound uniforms. Take your time to explore this view as well. Checking that your uniforms are correct is an important part of the debugging process for your applications.

38.4 Stepping through a Program

One of the most powerful features of CodeXL is the ability to step through programs and evaluate the current state of OpenGL, including loaded textures, shader values, etc. By now you should be familiar with stepping through programs in Visual Studio. The limitation there is that you can only look at program level variables. CodeXL gives a much broader view of what is happening within OpenGL than simple debugging alone will do. When your program is running but the render isn't correct, CodeXL is a much better tool to work out what the problems are than Visual Studio.

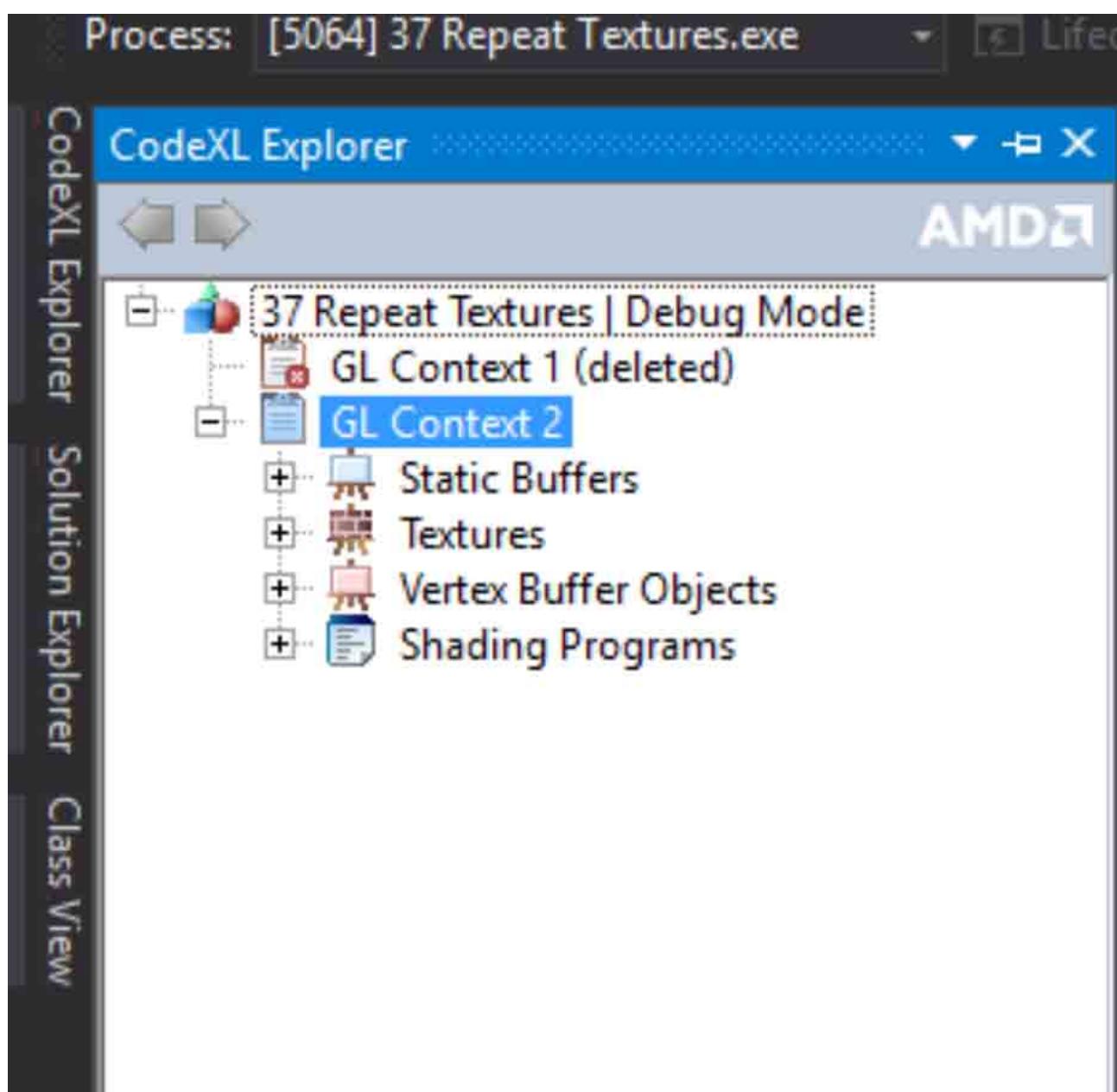


Figure 38.2: **CodeXL Explorer**

38.5 Excercises

1. Run a few other programs through CodeXL to get a feel for how the application works. Look at views and examine data. Try and work out where you can find information.
2. Examine the documentation for CodeXL and try and see where you can find out information about features and run through any small tutorials. CodeXL will become an invaluable tool for the rest of the module so get to know it now.

Part IV

Models Meshes and Transforms

Lesson 39

Geometry Builder

Up until now you have been working on defining your own geometric shapes by filling in a geometry object. Although this is useful knowledge to have (this is why we have focused on geometry generation for so long), it isn't really realistic to have you developing geometry like this for every application.

The graphics framework therefore provides a handy utility object called `geometry_builder` that will allow you to generate primitive geometric objects trivially. It has methods to generate the following basic geometry objects:

- Box
- Tetrahedron
- Pyramid
- Disk
- Cylinder
- Sphere
- Torus
- Plane

Each of these geometry types come with built in position data, texture coordinates, and surface normals (more on these later in the module). From now on we will predominately use the `geometry_builder` to generate geometry for us. Let us look at how we build each of these objects in turn.

39.1 Box

A box is a fairly simple piece of geometry - it is just the cube we have used when working in 3D. The reason we call it a box and not a cube is that the sides of a box are not necessarily equal. Some graphics libraries use the term cube when creating a box, but we are using the correct definition in our framework.

As you have already defined a cube you know how the shape works. The only difference is that we can also define the size of the box. The method call to create a box is shown below.

```
1 geometry geom = geometry_builder::create_box(vec3 dimensions);
```

The `dimensions` parameter defaults to $(1, 1, 1)$ so you do not have to define it if you don't need to. Any dimensions not set as $(1, 1, 1)$ will provide tiling of textures on the box. This is true for all the geometry types that you can define dimensions for via the `geometry_builder`.

39.2 Tetrahedron

A tetrahedron is similar to a box, except that ever side is a triangle not a quad. As such it has only four sides, not six. Creating a tetrahedron is similar to box creation - all we need to provide extra is a dimension value (defaulted to $(1, 1, 1)$).

```
1 geometry geom = geometry_builder::create_tetrahedron(vec3 dimensions ←
    );
```

39.3 Pyramid

Hopefully you know what a pyramid looks like by now. A pyramid is similar to a tetrahedron except that it has a square base rather than a triangle. Therefore it has five sides instead of four. Creating a pyramid with the `geometry_builder` is similar to the box and tetrahedron - we only require a dimension value (defaulted to $(1, 1, 1)$).

```
1 geometry geom = mesh(geometry_builder::create_pyramid(vec3 ←
    dimensions);
```

39.4 Disk

So far our basic shapes have been very primitive, made up of only a few triangles. This is because the shapes we have created have no curves. To create curved surfaces (such as our sphere using sub-division) we require more triangles.

The first shape we are going to create is a disk. A disk is essentially a 2D object - a circle. The technique we are going to use to create a disk is using the `GL_TRIANGLE_FAN` technique we looked at near the start of the workbook. We determine how "round" the disk looks by stating the number of triangles (or `slices`) we want the disk to have. The more `slices` the rounder the object will look, but it will require more triangles (and more triangles takes longer to render).

As before the disk also takes a dimensions value to enable the size of the disk to be determined. Therefore the call to create a piece of disk `geometry` is as follows:

```
1 geometry geom = geometry_builder::create_disk(unsigned int slices, ←
    vec3 dimensions);
```

39.5 Cylinder

A cylinder is the natural progression from a disk. A cylinder is just two disks (a top and a bottom) connected together. The disk creation method we have already explained. For the cylinder, slices are also important (obviously). Another property we can set is the number of stacks. A stack is just a row of triangles linked together to make a loop. The more stacks we have, the more triangles.

As before, we can also set the dimensions of our cylinder. Our creation call then looks as follows:

```
1 geometry geom = geometry_builder::create_cylinder(unsigned int ←
    stacks, unsigned int slices, vec3 dimensions);
```

39.6 Sphere

We have already seen a technique for creating a sphere using recursion and subdivision. Although a neat computational trick, this approach takes time and resources to undertake (and doesn't produce uniform results). It is also difficult to calculate texture coordinates trivially using this approach.

Another technique is to follow a similar approach to cylinder creation. We have a top and base (which are no longer flat disks but slightly curved) and a number of stacks or loops that increase then decrease in width at their edges. Enough stacks and enough slices and we have a convincing sphere.

Again, dimensions can be set for the cylinder. This provides a sphere creation method similar to the cylinder.

```
1 geometry geom = geometry_builder::create_sphere(unsigned int stacks, ←
    unsigned int slices, vec3 dimensions);
```

39.7 Torus

A torus is an interesting shape to create in 3D graphics as it allows you to test texturing, lighting and shadowing very convincingly. A torus is essentially a donut shape or a ring. You can also think of it as a cylinder which is curved in such a way that the top and bottom connect. As such, we can use the stacks and slices technique we have used previously.

Another factor we must consider this time is the radius of the ring (i.e. the cylinder part) and the total radius of the torus. These two values replace the dimensions value from before. The call to create a torus is therefore as follows:

```
1 geometry geom = geometry_builder::create_torus(unsigned int stacks, ←
    unsigned int slices, float ring_radius, float outer_radius);
```

39.8 Plane

Our final piece of geometry provided by the `geometry_builder` is a plane (although strictly speaking not really - a plane is infinite). A plane looks like a quad, but is actually made up of a number of triangles. We can define the width and depth of the plane so that we can determine the number of triangles. A single unit of the plane (1 width and 1 depth) is made up of two triangles (similar to a quad).

The call to create a piece of plane geometry is below:

```
1 geometry geom = geometry_builder::create_plane(unsigned int width, ←
     unsigned int depth);
```

39.9 Exercise

Now that you have seen how to create geometry with the `geometry_builder` it is time for you to use these techniques to create a scene. Below is a description of the objects you have to create. You just need to create these objects and allocate them to meshes so that you can set their relative transforms.

A `map` object has been created to store the `mesh` objects mapped to a name (represented by a `string`). A `map` is sometimes called a dictionary in other languages. A plane has already been added to the `map` to illustrate how to use it, although the box below will help you further.

Working with std::map

A `std::map` (or just `map`) is an associative date structure used to store an object value against a key that can be looked up. You can think of it as a simple database of key-value pairs. The keys and values can be almost anything. In our case we are creating a `map` that associates names (`string`) to meshes. The type of the `map` is therefore:

```
1 map<string, mesh> meshes;
```

There are different methods to add items to a `map`, but the simplest is to just allocate to the given key value (this will add to the `map` or overwrite the existing value):

```
1 meshes["name"] = mesh(...);
```

To retrieve an item from a `map` we use the same technique in reverse:

```
1 auto m = meshes["name"];
```

This will return a reference to the value store in the `map` (*note - a reference, not a copy*). This allows us to manipulate the values stored in the `map`.

In the application it is worth looking at the `render` function. It uses a `for` each loop to iterate through the `meshes` `map`.

```
1 for (auto &m : meshes)
2     m....
```

In this case, `m` is a reference to the `map` entry, which is represented as a `pair` (`std::pair<string, mesh>` in our case). This means that we have to get the individual values from the entry ourselves (`first` being the key and `second` being the value).

If you are unsure about maps then ask.

The scene you are to create is as follows:

box default dimensions

scale (5, 5, 5)
position (-10, 2.5, -30)

tetra default dimensions

scale (4, 4, 4)
position (-30, 10, -10)

pyramid default dimensions

scale (5, 5, 5)
position (-10, 7.5, -30)

disk 20 slices, default dimensions

scale (3, 1, 3)
position (-10, 11.5, -30)
rotation_x $\frac{\pi}{2}$

cylinder 20 slices, 20 stacks, default dimensions

scale (5, 5, 5)
position (-25, 2.5, -25)

sphere 20 slices, 20 stacks, default dimensions

scale (2.5, 2.5, 2.5)
position (-25, 10, -25)

torus 20 slices, 20 stacks, 1 ring radius, 5 outer radius

position (-25, 10, -25)
rotation_x $\frac{\pi}{2}$

Figure 39.1 illustrates the expected output from this exercise.

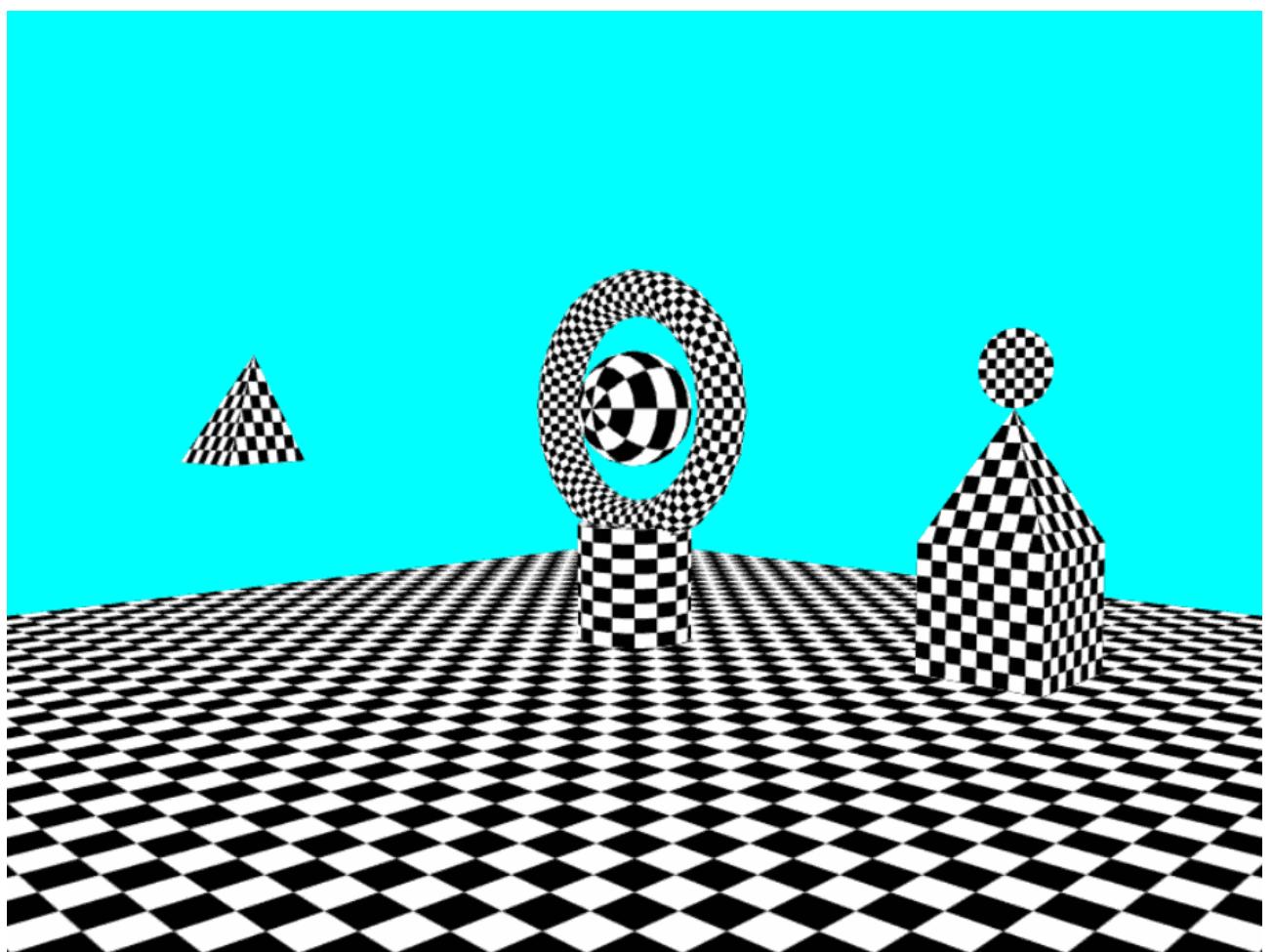


Figure 39.1: Output from `geometry_builder` Application

Lesson 40

Transform Hierarchy

40.1 Transform refresher

As we have learned in Section II, to render meshes we need a Model matrix which defines the location, scale and rotation of an object.

$$[ModelMatrix] = [Translation] \times [Rotation] \times [Scale]$$

This is used in conjunction with the Projection Matrix (calculated from the Field of View and window dimensions) and View Matrix (camera position and rotation)

$$[MVP] = [ProjectionMatrix] \times [ViewMatrix] \times [ModelMatrix]$$

You should understand by now that as the rules of matrix multiplication dictate, this process is like a chain of operations, each matrix is applied to the next, transforming it in some way. We could skip the two line approach and do it all in one line like so:

$$[MVP] = [V] \times [P] \times [T] \times [R] \times [S]$$

This would be wasteful, in fact it's unlikely that [P] ever change, and [V] will only ever change if the camera moves, so it would be good optimisation to calculate $[P^*V]$ only once per frame, or even better: in the update function whenever the camera moves. Then we can just do this in the render loop, and skip one Mat4 multiplication for every mesh:

$$[MVP] = [PV] \times [ModelMatrix]$$

Could we go further? You Bet! The Model matrix is only ever going to change when a model is moved, rotated, or transformed. Why not save the computed transform matrix within the mesh, and only ever re-calculate when necessary? The framework doesn't do this (for code clarity reasons), but if you are ever building your own render codebase, you should implement optimisations like this.

40.2 Chaining transforms

Now we have refreshed ourselves on the magic of Matrices, let's cover some new ground: If we consider the model matrix, which is a chain of three transformations, there is nothing stopping

us adding more. For example we could have global translate Matrix, which will move every object in the scene in some direction.

$$[ModelMatrix] = [GlobalTranslate] \times [Translation] \times [Rotation] \times [Scale]$$

This is perfectly valid, but a little contrived. How about something incredibly useful, like having one object move along with another. Say we have Two cubes, We want Cube 2 to sit on top of Cube 1 and move where it moves. You may think to do it like this:

-
- 1: $\text{vec3 } C1_Trans = \text{SomeMovingPosition}();$
 - 2: $[C1ModelMatrix] = [\text{translate}(C1_Trans)] \times [C1_Rot] \times [C1_Scale]$
 - 3: $[C2ModelMatrix] = [\text{translate}(C1_Trans + \text{vec3}(0, 1, 0))] \times [C2_Rot] \times [C2_Scale]$
-

This has many problems: We have to keep another Vec3 around, and what if Cube1 were to rotate? or scale?

Why not do it like this:

-
- 1: $[C1ModelMatrix] = [\text{translate}(\text{SomeMovingPosition}())] \times [C1_Rot] \times [C1_Scale]$
 - 2: $[C2ModelMatrix] = [C1ModelMatrix] \times [\text{translate}(\text{vec3}(0, 1, 0))] \times [C2_Rot] \times [C2_Scale]$
-

If you think about it and expand it out, what we are doing is copying all the C1 transforms to C2, and then applying C2 transforms (i.e , moving it up 1 unit to sit on top of the cube). Now we inherit all the scale, rotation and translation of C1, all with just one multiply!

Using this technique we can set-up chains of objects that are connected to each other, in a parent-child hierarchy.

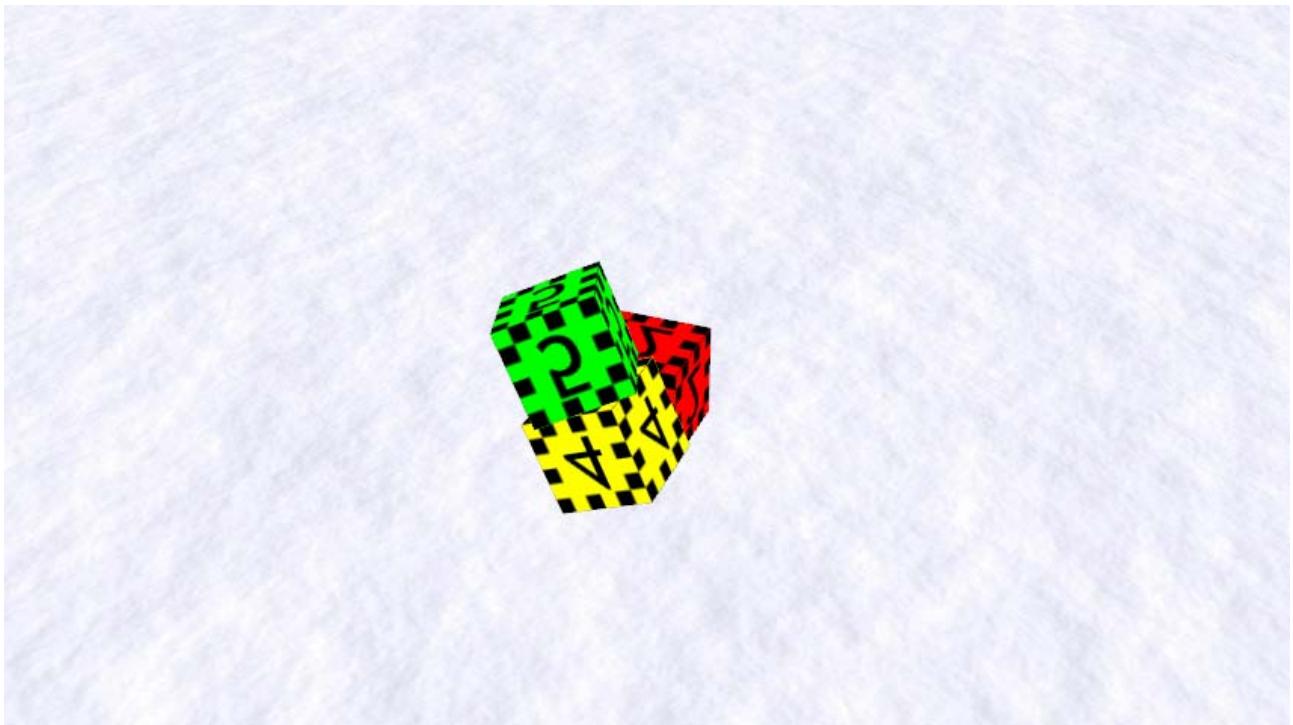


Figure 40.1: **Output of Practical**

Lesson 41

Loading Models for Geometry

Of course, we cannot just work with graphics rendering with simple models. The rendering framework also supports the ability to load in model data produced by artists. This is handled by the AssImp (Asset Importer) library. Let us look at how we load in a model as geometry.

41.1 Loading a Model

The `geometry` object comes with a constructor that accepts a `string` as input. This `string` is a filename for a model to load. For example, we can load a model as follows:

```
1 geometry geom("filename");
```

Normally we would want to do this as part of `mesh` creation:

```
1 mesh m(geometry("filename"));
```

The render framework will take care of creating the relevant buffers of data for you to use. Working with models like this is useful for our work in graphics rendering, however you should try and look into loading models yourself without AssImp. This can be an important skill to pick up.

41.2 Updating Lesson

Your task in this lesson is to load a model and texture to render. A good initial model to use is a teapot, and you will find one in `../resources/models` folder. You can chose which texture to load. Running the application should give you an output similar that shown in Figure 41.1.

41.3 Exercise

Try downloading and using other models in your application. You will very likely have to play around with the scaling of certain models you download as they are not uniform. You may also



Figure 41.1: Output from Model Loading Lesson

find models that come with textures. You should try and use these textures as well.

Part V

Cameras

Lesson 42

Cameras

We are going to digress from our work on shader development for a while, and do some work with cameras.

42.1 What do we mean by Cameras?

A camera is just a convenient method for us to discuss the concept of the viewer for a scene. We have been working with these concepts for a long time - we set up a view matrix using eye position, view angle, etc. What we want to do is have an approach that lets us have some standard cameras that we can reuse to explore our scenes. Over the next few lessons, we will build a set of standard cameras. We will use methods to move the eye position, rotate our angle of view, and other more specific methods.

Figure 42.1, taken from Real Time Rendering [1], should help you see what we are essentially doing.

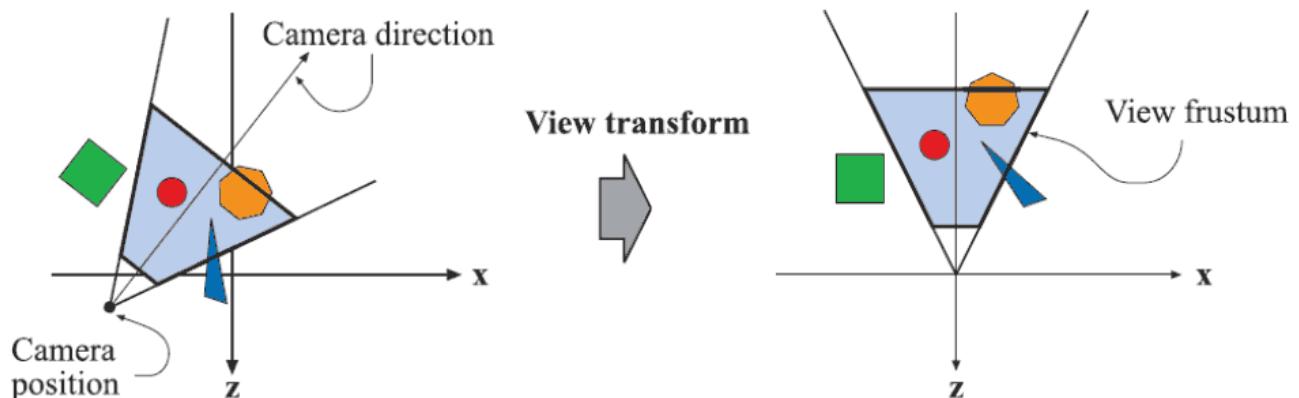


Figure 42.1: View Transformation

Figure 42.1 shows how we can view a camera. We have the position and the direction (or camera target). This helps define what is known as a view frustum, which can be considered the camera space or view point.

Figure 42.2 shows how the view from the camera is transformed into a 2D image. There are items that can be seen and not seen. The final render is based on what is in the viewpoint of the camera.

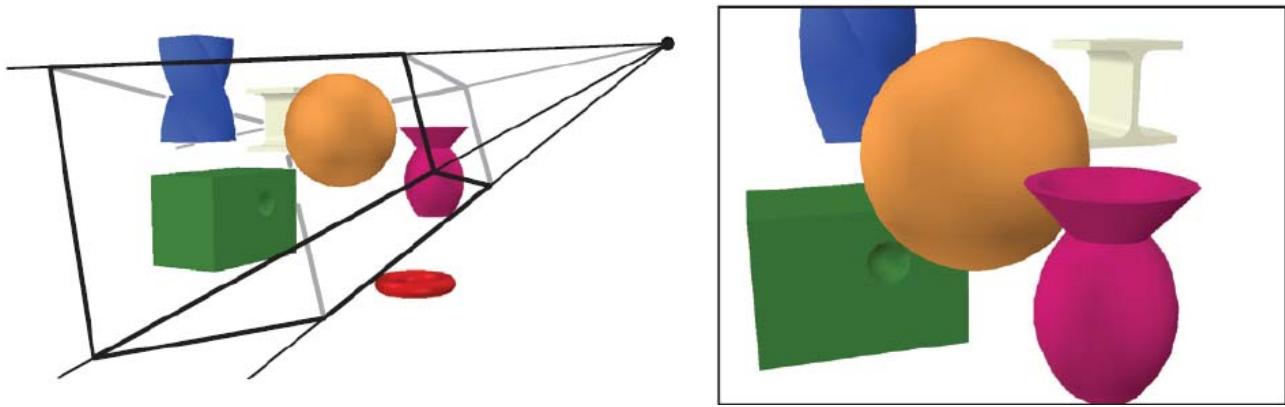


Figure 42.2: Screen Transformation

42.1.1 Why do we need Cameras?

The big reason we are building cameras is to allow us to generate our view and projection matrices easily, and use them in our shaders. This will make our life much easier.

Our work with cameras will depend on the type of camera we implement. However, there is a base class for the camera which we will look at shortly. First, let us build up our understanding of what a camera is by examining how the eye works.

42.2 From the Eye to a Camera

Let us now build up our understanding of our camera by exploring some basic aspects of vision and how we can build these ideas up to a synthetic (or virtual) camera model. Let us start our examination by first looking at the eye.

42.2.1 The Eye

The eye takes in light through the pupil. The light travels through the eye before hitting the retina (the back of the eye). The retina is covered with rods and cones, which are activated by the light providing our brain with the information to create an image.

There are two main components of an eye that allow us to perceive an image:

Pupil captures light through a lens

Retina receives the light to create an image

We can simplify these to having some form of lens to capture light and some form of sensor to receive the light. Figure 42.3 provides an illustration of how the eye works on an abstract level. Notice as well that a sensed image is upside down. The brain corrects for this.

42.2.2 Pinhole Camera

Now let us look at cameras and how they relate to the eye. We have already defined the requirements for a lens and a sensor. For a simple pinhole camera, we have a pinhole in the front to receive light. This light is then projected onto the back of the camera, creating an

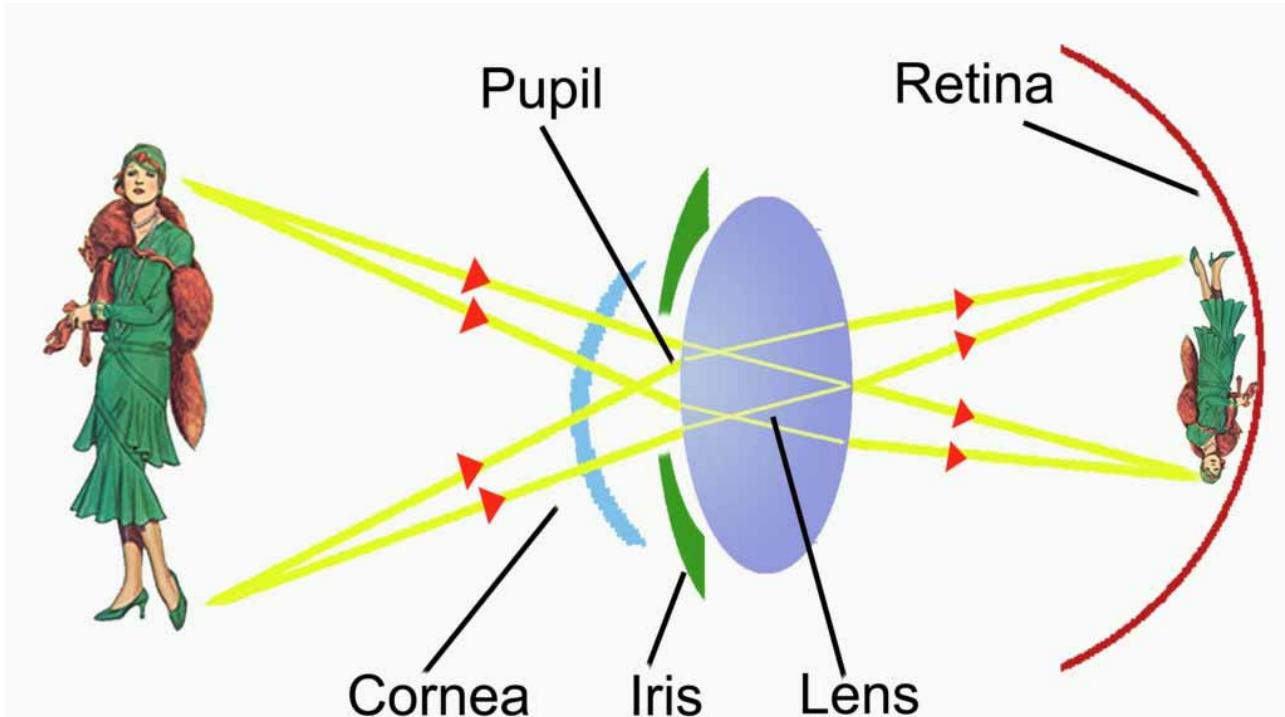


Figure 42.3: The Eye

image. This is just a lens (the pinhole) and a sensor (the back of the camera onto which the image is projected).

The idea of a lens and a projection are fundamental to our understanding of how 3D graphics are transformed onto to allow displaying on the screen. We will now use the terms for the lens and sensor which we will use in the rest of the module - ***view*** and ***projection***. Section 1.4 of *Interactive Computer Graphics* [3] provides more details on pinhole cameras. Figure 42.4 illustrates a basic pinhole camera model.

42.2.3 Synthetic Camera Model

One thing you might have noticed when we have described the eye and a camera is that the projected image always forms upside down. This is due to the way light travels from the eye to the projector. For the eye, the brain inverts the image for us. For a camera, the film can be flipped or the image inverted on the device.

For 3D rendering, we take a different approach - the image isn't projected past the lens but in front of it. Figure 42.5 illustrates the general idea.

Knowing now what you do about the eye and cameras, we can define the basic properties of a camera. You have to think about the camera existing in our 3D world and what properties it would require as such. We need to define the following properties for the lens (***view***):

World position the location in the world the camera is

Direction where the camera is facing

Orientation which way the camera is rotated

For the sensor (***projection***) we require the following:

Field of view how wide an angle can the sensor pick up from the lens

Pinhole Camera : Principle

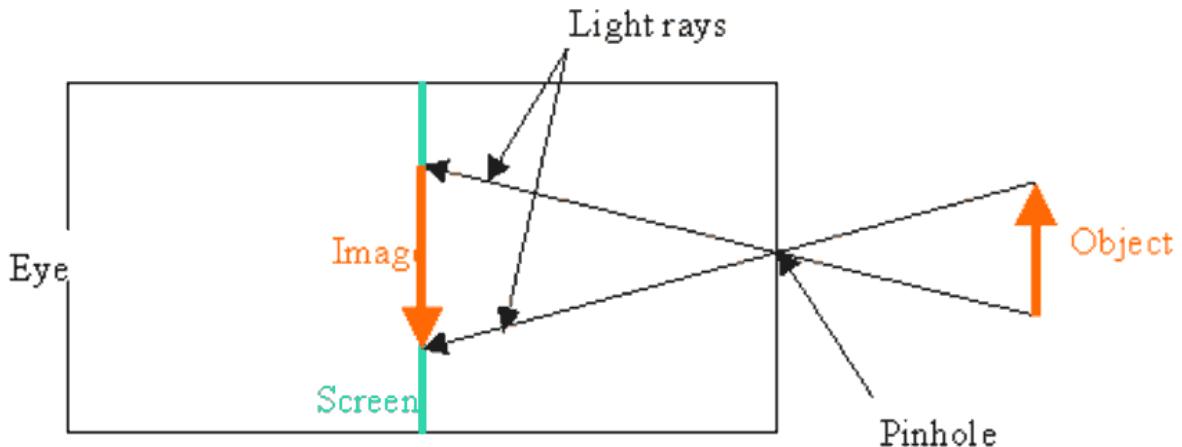


Figure 42.4: Pinhole Camera

Aspect ratio the ratio of the width to height of the sensor

Near plane position where we are projecting to

Far plane position how far the camera can see.

We will look at view and projection individually soon, but we have the basic properties that we can use to define a camera in our render framework.

42.3 Base Camera Class

The base class for a camera is given below:

```

1 class camera
2 {
3 protected:
4     // The current position of the camera
5     glm::vec3 _position;
6     // The current target the camera is looking at
7     glm::vec3 _target;
8     // The current orientation of the camera
9     glm::vec3 _up;
10    // The currently built view matrix since the last frame update
11    glm::mat4 _view;
12    // The currently built projection matrix since the last call to ←
13    //      set_projection
14    glm::mat4 _projection;
15    // Creates a new camera. Called by the sub-classes
16    camera() = default;

```

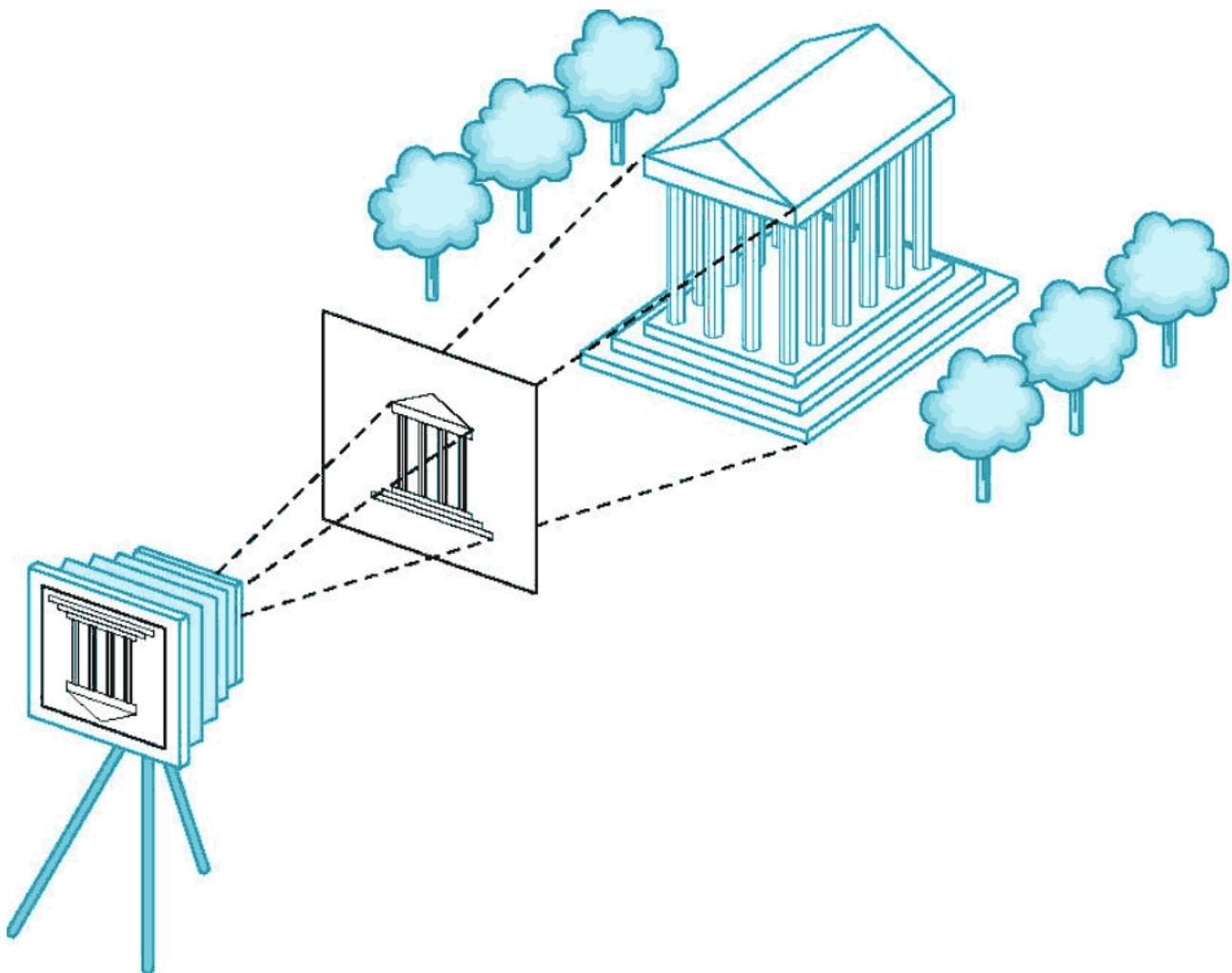


Figure 42.5: ynthetic Camera Model taken from *Real-Time Rendering* [1]

```

17 // Destroys the camera. Virtual destructor as this is an abstract ←
18     class
19     virtual ~camera() { }
20     // Gets the current position of the camera
21     const glm::vec3& get_position() const { return _position; }
22     // Sets the position of the camera
23     void set_position(const glm::vec3 &value) { _position = value; }
24     // Gets the current target of the camera
25     const glm::vec3& get_target() const { return _target; }
26     // Sets the target of the camera
27     void set_target(const glm::vec3 &value) { _target = value; }
28     // Gets the current up direction of the camera
29     const glm::vec3& get_up() const { return _up; }
30     // Sets the up direction of the camera
31     void set_up(const glm::vec3 &value) { _up = value; }
32     // Gets the current view matrix for the camera
33     const glm::mat4& get_view() const { return _view; }
34     // Gets the current projection matrix for the camera
35     glm::mat4 get_projection() const { return _projection; }
36     // Builds projection matrix
37     void set_projection(float fov, float aspect, float near, float far ←
38 );

```

```
37 // Updates the camera. This is a pure virtual function.  
38 virtual void update(float delta_time) = 0;  
39 };
```

A basic camera has the following attributes:

position the position of the camera in world space.

target the position in the world space that the camera is looking at.

up the up direction of the camera in world space. This defines the orientation of the camera.

view the view transformation matrix. Used to convert geometry from world space to camera space.

projection the projection transformation matrix. Use to convert geometry from camera space to screen space.

We will look at space transformations in the next three lessons. The point here is that the camera itself requires the generation and management of the **view** and **projection** matrices. This is done by updating and managing the three other attributes - **position**, **target**, and **up**.

The majority of the work undertaken in our cameras will be done in a method called **update**. This will define the current view matrix (the lens representation). The projection matrix is set at the start of the application and has no need to change for our work.

Lesson 43

Coordinate Spaces

Before we look at individual camera implementations let us examine some of the underlying theory in transforming our objects so that they are represented on the screen. This requires us to understand a little bit about coordinate spaces. A coordinate space is just the definition of a 3D space based on where the origin is, and which way the three axes point.

Consider that we have a 3D coordinate space C that has an origin and three coordinate axes. Any point in this coordinate space can be defined using $\langle x, y, z \rangle$, with these values indicating the distance travelled along each of the three relevant axes to reach the point.

Suppose now we want to exist in a different 3D coordinate space, C' . The original point, P can be expressed as having coordinates $\langle x', y', z' \rangle$ in this coordinate space. These new coordinates can be expressed as linear functions of the original coordinates:

$$\begin{aligned}x'(x, y, z) &= U_1x + V_1y + W_1z + T_1 \\y'(x, y, z) &= U_2x + V_2y + W_2z + T_2 \\z'(x, y, z) &= U_3x + V_3y + W_3z + T_3\end{aligned}$$

What we will look at over the next few lessons is how we generate these different transformations.

43.1 Linear Transformations

Our transformation from C to C' is what we call a *linear transformation*. We can define this as a matrix calculation:

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} U_1 & V_1 & W_1 \\ U_2 & V_2 & W_2 \\ U_3 & V_3 & W_3 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} T_1 \\ T_2 \\ T_3 \end{bmatrix}$$

There are a number of values to consider here:

- The vector $\langle x', y', z' \rangle$ represents the distance travelled along the axes of C' to reach P
- The vector \mathbf{T} represents the translation of the origin of C to C'
- The vectors \mathbf{U} , \mathbf{V} and \mathbf{W} represent the change in orientation between the coordinate spaces.

If the transformation matrix is invertible, then the following equation holds:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} U_1 & V_1 & W_1 \\ U_2 & V_2 & W_2 \\ U_3 & V_3 & W_3 \end{bmatrix}^{-1} \left(\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} - \begin{bmatrix} T_1 \\ T_2 \\ T_3 \end{bmatrix} \right)$$

43.2 Transformation Matrix

We have already looked at our model transformation matrices in Lesson 10, and noted that we use 4×4 matrices rather than a 3×3 matrix. This is just to allow us to perform a transformation in a single matrix multiplication. Therefore, we typically extend our transformation matrix to the following:

$$\begin{bmatrix} U_1 & V_1 & W_1 & T_1 \\ U_2 & V_2 & W_2 & T_2 \\ U_3 & V_3 & W_3 & T_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

In Lesson 10 we defined the **T** vector as our translation, with the other three vectors (**U**, **V**, **W**) representing our combined rotation and scale transformations. This is similar to the idea of moving the origin (we translate the object into the world) and changing the axes (through rotation and scaling).

We can concatenate multiple transformations by multiplying transformation matrices together. This is the fundamental part of what we are doing. When we work with vectors, we move from object space to world space (via the model matrix) to camera space (via the view matrix) to screen space (via the projection matrix). Concatenating matrices allows us to do this in one operation.

43.3 Coordinate Spaces of Interest

When working with computer graphics we have four coordinate spaces of interest that we move between. The transformation of these spaces is shown below:

$$\textit{Model} \Rightarrow \textit{World} \Rightarrow \textit{Camera} \Rightarrow \textit{Screen}$$

These spaces are defined as follows:

Model the space where the origin is the centre of the model. This is how we define our geometry within the buffers. Each coordinate is related to the origin being $(0, 0, 0)$.

World the space where we move our object into. The translation matrix determines where we move the origin of the object to, and the rotation and scale matrices orientate and scale the object accordingly. The world coordinate space is how you should be thinking about the scene you create. It positions the objects accordingly.

Camera the space that the camera works in. The camera space has its own position as the origin point, so the entire world is essentially translated to suit this. The world is also rotated based on the camera's orientation.

Screen the space that the screen works in. The screen space is essentially taking everything in the world transforming their coordinates for how the projection is defined (view angle, aspect ratio, near and far projections). The screen itself is defined as a space with $(0, 0, 0)$ at the centre, and only has an extent of 1 unit in each direction. This means that any vertex which has a final screen coordinate not in the range $(-1, -1, -1)$ to $(1, 1, 1)$ is not visible on the screen.

43.4 Transforming Between Spaces

We have already covered the transformation from model to world space, and the next two lessons look at the camera transformation (the *view* matrix) and the screen transformation (the *projection* matrix). The one thing to consider at the moment is the order of multiplication. Consider a vertex \mathbf{A} that we want to transform from one space to another. If it starts in the model space, then we can transform it into the world space using the model transform:

$$\mathbf{A}_w = \mathbf{M} \times \mathbf{A}$$

If we want to transform this vertex into the camera space, we do the following:

$$\mathbf{A}_c = \mathbf{V} \times \mathbf{A}_w = \mathbf{V} \times (\mathbf{M} \times \mathbf{A})$$

And finally if we want to transform the vertex into camera space, we do the following:

$$\mathbf{A}_s = \mathbf{P} \times \mathbf{A}_c = \mathbf{P} \times (\mathbf{V} \times (\mathbf{M} \times \mathbf{A}))$$

Reading from left to right, our order of multiplication is *PVM*. Multiplying our matrices in this order produces a single *model-view-projection* transformation matrix (notice the name is the reverse of the multiplication order).

Now let us consider how we go from a vertex in screen space to a vertex in model space (we do sometimes have to go back through our spaces). To go back through we need to use the inverse of the transformation matrix (e.g. \mathbf{P}^{-1}). So to transform a vertex \mathbf{A} from screen space to camera space we do the following:

$$\mathbf{A}_c = \mathbf{P}^{-1} \times \mathbf{A}$$

Likewise to transform this vertex to world space we perform the following calculation:

$$\mathbf{A}_w = \mathbf{V}^{-1} \times \mathbf{A}_c = \mathbf{V}^{-1} \times (\mathbf{P}^{-1} \times \mathbf{A})$$

And finally to transform back to model space we have the following:

$$\mathbf{A}_m = \mathbf{M}^{-1} \times \mathbf{A}_w = \mathbf{M}^{-1} \times (\mathbf{V}^{-1} \times (\mathbf{P}^{-1} \times \mathbf{A}))$$

This time reading from left to right we have the following transformation matrix, which can be simplified into 1 inversion operation:

$$\mathbf{M}^{-1} \mathbf{V}^{-1} \mathbf{P}^{-1} = (\mathbf{PVM})^{-1}$$

We will be travelling through these coordinate spaces in different manners depending on context. Sometimes we need to work in world space or camera space for certain effects. Also, we will be sometimes going from screen space back to world space to work with object interaction.

43.5 Recommended Reading

Chapter 4 of *Mathematics for 3D Game Programming and Computer Graphics* [2] goes into far more detail of these concepts than we do here. Review the chapter, and attempt some of the questions within it. Some important parts we haven't covered are arbitrary axis rotation and spherical linear interpolation using quaternions.

Lesson 44

View Transformation Matrix

We have now defined what a camera is, and we have also looked at the underlying theory of coordinate spaces. Now let us look at the camera transformation matrix, or the view transformation matrix. As with our model transformation matrix, we are going to define a 4×4 matrix so we can combine it with our model transform.

44.1 Defining a View

We already defined the values required for a camera, but just to recap we have the following values of interest:

view position the position in the world where the camera is placed

view direction the direction the camera is facing

view orientation the rotation of the camera (which way is up)

What we need to do is understand how we transform these values into our 4×4 transformation matrix.

44.2 Constructing a View Matrix

The job of the view transformation matrix is to change the geometry of the world so that the camera is located at the origin $(0, 0, 0)$. Therefore we need to reorientate the world so that it is rotated to suit the camera, and then translate the world so that the camera is in the centre.

Remember that our transformation takes the form:

$$\begin{bmatrix} U_1 & U_2 & U_3 & T_1 \\ V_1 & V_2 & V_3 & T_2 \\ W_1 & W_2 & W_3 & T_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Where the vectors **U**, **V** and **W** represent the new axes, and the vector **T** represents the translation across these axes. These three vectors actually represent the *side*, *up*, and *forward*

vectors that define a camera. Therefore, we can actually redefine our transformation matrix for the view as follows:

$$\begin{bmatrix} side_x & side_y & side_z & T_1 \\ up_x & up_y & up_z & T_2 \\ -forward_x & -forward_y & -forward_z & T_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Why is the Forward Vector Negated?

Remember that in OpenGL the forward direction is negative down the Z-axis (that is forward is $(0, 0, -1)$). Because of this, we have to negate the forward vector when generating our view matrix.

So given our three camera values (position, target and orientation) how do we generate our axes? This is actually just a few calculations, remembering that our axes need to be unit length (therefore normalised). We can calculate our individual values as follows:

$$\begin{aligned} forward &= (\widehat{\text{target} - \text{position}}) \\ side &= (forward \times \widehat{\text{orientation}}) \\ up &= side \times forward \end{aligned}$$

With our axis definitions in place, all we need to do is calculate our translation. We can do this using the dot product. The full calculation of our view matrix is given in Algorithm 9.

Algorithm 9 View Matrix Generation

Require: $\text{position} \neq \text{target}$

```

1: function LOOK_AT( $\widehat{\text{position}}, \widehat{\text{target}}, \widehat{\text{orientation}}$ )
2:    $forward \leftarrow (\widehat{\text{target} - \text{position}})$ 
3:    $side \leftarrow (forward \times \widehat{\text{orientation}})$ 
4:    $up \leftarrow side \times forward$ 
5:    $view \leftarrow \begin{bmatrix} side_x & side_y & side_z & T_1 \\ up_x & up_y & up_z & T_2 \\ -forward_x & -forward_y & -forward_z & T_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$ 
6:   return  $view$ 

```

44.3 Transforming Objects - World Space to Camera Space

Let us now test this view matrix definition. We will start with a basic camera with the following values:

$$\begin{aligned} \textit{position} &= (0, 0, 0) \\ \textit{target} &= (0, 0, -1) \\ \textit{orientation} &= (0, 1, 0) \end{aligned}$$

If we now calculate our axes values we get the following:

$$\begin{aligned} \textit{forward} &= (\widehat{\textit{target} - \textit{position}}) = (0, \widehat{0}, -1) = (0, 0, -1) \\ \textit{side} &= (\widehat{\textit{forward} \times \textit{orientation}}) = (1, \widehat{0}, 0) = (1, 0, 0) \\ \textit{up} &= \textit{side} \times \textit{forward} = (0, 1, 0) \end{aligned}$$

Our three translation values are:

$$\begin{aligned} T_x &= -(\textit{side} \cdot \textit{position}) = 0 \\ T_y &= -(\textit{up} \cdot \textit{position}) = 0 \\ T_z &= \textit{forward} \cdot \textit{position} = 0 \end{aligned}$$

Our view transformation matrix is therefore:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Which is the identity matrix. Does this make sense? Well think about it. Our camera is in position 0, so we wouldn't expect any translation. Also, the forward and up directions of the camera are the same as the forward and up directions of the world ((0, 0, -1) and (0, 1, 0) respectively). Therefore we would expect the identity matrix - no transformation has to happen.

44.3.1 Performing a Camera Transformation

OK, let us now define another camera with values that don't give us the identity matrix. Let our camera values be:

$$\begin{aligned} \textit{position} &= (20, 20, 20) \\ \textit{target} &= (0, 0, 0) \\ \textit{orientation} &= \left(-\frac{1}{\sqrt{6}}, \frac{2}{\sqrt{6}}, -\frac{1}{\sqrt{6}}\right) \end{aligned}$$

Calculate our transformation axes values we get:

$$\begin{aligned} \text{forward} &= (\text{target} - \widehat{\text{position}}) = (-20, \widehat{-20}, -20) = \left(-\frac{1}{\sqrt{3}}, -\frac{1}{\sqrt{3}}, -\frac{1}{\sqrt{3}}\right) \\ \text{side} &= (\text{forward} \times \widehat{\text{orientation}}) = \left(\frac{1}{\sqrt{2}}, 0, -\frac{1}{\sqrt{2}}\right) = \left(\frac{1}{\sqrt{2}}, 0, -\frac{1}{\sqrt{2}}\right) \\ \text{up} &= \text{side} \times \text{forward} = \left(-\frac{1}{\sqrt{6}}, \frac{2}{\sqrt{6}}, -\frac{1}{\sqrt{6}}\right) \end{aligned}$$

Calculating our translation values we get:

$$\begin{aligned} T_x &= -(\text{side} \cdot \text{position}) = -\left(\frac{20}{\sqrt{2}} - \frac{20}{\sqrt{2}}\right) = 0 \\ T_y &= -(\text{up} \cdot \text{position}) = -\left(\frac{-20}{\sqrt{6}} + \frac{40}{\sqrt{6}} + \frac{-20}{\sqrt{6}}\right) = 0 \\ T_z &= \text{forward} \cdot \text{position} = \frac{-20}{\sqrt{3}} + \frac{-20}{\sqrt{3}} + \frac{-20}{\sqrt{3}} = -\frac{60}{\sqrt{3}} \end{aligned}$$

Therefore, our final transformation matrix is as follows:

$$\begin{bmatrix} \frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}} & 0 \\ -\frac{1}{\sqrt{6}} & \frac{2}{\sqrt{6}} & -\frac{1}{\sqrt{6}} & 0 \\ \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{3}} & -\frac{60}{\sqrt{3}} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Let us now use this transformation matrix on the point $(5, 5, 5)$. Let us just consider that this point has already been transformed from model space into world space. Our calculation is:

$$\begin{bmatrix} \frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}} & 0 \\ -\frac{1}{\sqrt{6}} & \frac{2}{\sqrt{6}} & -\frac{1}{\sqrt{6}} & 0 \\ \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{3}} & -\frac{60}{\sqrt{3}} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 5 \\ 5 \\ 5 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -\frac{45}{\sqrt{6}} \\ 1 \end{bmatrix}$$

Again does this make sense? Well think about the point in relation to the camera. It will be in the centre of the camera view (point $(5, 5, 5)$ is on the line between $(0, 0, 0)$ and $(20, 20, 20)$). So we expect the x and y components to be 0.

Thankfully GLM takes care of generating and managing view transformation matrices for us, but you should still be aware of the underlying theory and be able to calculate a view matrix from camera values if needs be.

44.4 Exercises

1. Given the following camera properties generate the appropriate view matrices using the technique described.
 - (a) **position** $(0, 0, 0)$

- target** $(0, 0, 1)$ *looking backward*
orientation $(0, 1, 0)$
- (b) **position** $(0, 0, 10)$ *translated back from the target*
target $(0, 0, 0)$
orientation $(0, 1, 0)$
- (c) **position** $(10, 0, 0)$ *looking to the left*
target $(0, 0, 0)$
orientation $(0, 1, 0)$
- (d) **position** $(0, 10, 0)$ *looking down*
target $(0, 0, 0)$
orientation $(0, 0, -1)$
- (e) **position** $(0, 0, 0)$
target $(0, 0, -1)$
orientation $(0, -1, 0)$ *camera rotated 2π rad*

2. Using the matrices you defined in question 1, transform the following world coordinates.

- (a) $(0, 0, 0)$
(b) $(0, 100, 0)$
(c) $(100, 0, 0)$
(d) $(0, 0, 100)$
(e) $(0, 0, -100)$

44.5 Answers

1. (a)

$$\begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

(b)

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -10 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

(c)

$$\begin{bmatrix} 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & -10 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

(d)

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & -10 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

(e)

$$\begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2. (a) i. $(0, 0, 0)$
ii. $(0, 0, -10)$
iii. $(0, 0, -10)$
iv. $(0, 0, -10)$
v. $(0, 0, 0)$
- (b) i. $(0, 100, 0)$
ii. $(0, 100, -10)$
iii. $(0, 100, -10)$
iv. $(0, 0, 90)$
v. $(0, -100, 0)$
- (c) i. $(-100, 0, 0)$
ii. $(100, 0, -10)$
iii. $(0, 0, 90)$
iv. $(100, 0, -10)$
v. $(-100, 0, 0)$
- (d) i. $(0, 0, -100)$
ii. $(0, 0, 90)$
iii. $(-100, 0, -10)$
iv. $(0, -100, -10)$
v. $(0, 0, 100)$
- (e) i. $(0, 0, 100)$
ii. $(0, 0, -110)$
iii. $(100, 0, -10)$
iv. $(0, 100, -10)$
v. $(0, 0, -100)$

Lesson 45

Projection Transformation Matrix

Now that we have seen how to build a view transformation matrix, let us move onto defining our final transformation matrix. This is the transformation that converts our camera space coordinates into final screen space coordinates.

When it comes to screen coordinates, we are only interested in those coordinates that are within the range $(-1, -1, -1)$ and $(1, 1, 1)$. These are the coordinates that are visible to the camera once they have been transformed into screen space. Any other coordinates are not visible, and therefore are not rendered.

This is another lesson looking at theory and how to generate transformation matrices. After this lesson we will get back to coding, looking at different camera types that we can use.

45.1 Defining a Projection

When we looked at cameras in Lesson 42 we looked at the values that make up the lens (or the view) and the sensor (the projection). For the projection, we discussed ideas such as the *field-of-view*, how close the camera could see, and how far. We also discussed the aspect ratio of the sensor (its width divided by its height). These are the four values we are interested in for our camera. To summarise:

fov the field of view (angle) that the camera can see at. We typically use $\frac{\pi}{4}$ here.

aspect the aspect ratio of the projection (window or screen size). We calculate the aspect ratio by dividing the window width by the window height. Typical values are $\frac{16}{9}$ or $\frac{4}{3}$.

near the closest an object can be and still be projected onto the screen.

far the farthest an object can be and still be projected onto the screen.

The question is how do we go from these values to a projection matrix. Let us look at this in the next section.

45.2 Constructing a Projection Matrix

The task of the projection matrix is to convert the camera space coordinates to screen coordinates. To do this, we need to think about how we want to convert the x , y , and z components

in camera space to those in screen space. Let us think about which values affect which components:

fov affects the x and y components

aspect affects the x and y components

near affects all components

far affects all components

For our x component, we need to determine whether the value is visible from the field of view. Considering that the field of view represents the full field of view, we only need to consider half of it (think about it - viewer in the centre and field of view in front. One side only has half the field of view). As the field of view is also essentially a ratio for us to work with, we use the tan function to get this value. We will define this value as a :

$$a = \tan(fov/2)$$

Also, our screen is typically wider than it is tall, so we use the aspect ratio and a together to calculate our x component of the screen coordinate. Finally, this value is used to convert the x component into our range. Thus, for our x component, we need to use the following:

$$x' = x \cdot \frac{1}{\text{aspect} \cdot a}$$

Our y component is similar, but doesn't rely on the aspect ratio (as the screen is wider than tall). To calculate the y component we use the following:

$$y' = y \cdot \frac{1}{a}$$

Our z component is concerned with where it fits between the near and far plane as a ratio. This is actually a simple equation:

$$z' = z \cdot \frac{\text{far} + \text{near}}{\text{far} - \text{near}}$$

We now have enough information to start constructing our matrix. So far we have the following:

$$\begin{bmatrix} \frac{1}{\text{aspect} \cdot a} & 0 & 0 & ? \\ 0 & \frac{1}{a} & 0 & ? \\ 0 & 0 & -\frac{\text{far} + \text{near}}{\text{far} - \text{near}} & ? \\ ? & ? & ? & ? \end{bmatrix}$$

So what goes in the ? values. Most of these come out as 0, but two we need to set the value. The first is value m_{32} which we set to -1. This is to take into account that we have a perspective. A point further down the z-axis will appear closer to the centre than if it was nearer the viewer (assuming we the x and y values do not change).

The other value we need is used to manipulate the z-value to take account of the squashing performed by the perspective. Let us look at our final algorithm to generate our perspective projection, which is shown in Algorithm 10.

Algorithm 10 Perspective Projection Matrix Generation

Require: $fov \neq 0, far \neq near$

```

1: function PERSPECTIVE( $fov, aspect, near, far$ )
2:    $a \leftarrow \tan(fov/2)$ 
3:    $projection \leftarrow \begin{bmatrix} \frac{1}{aspect \cdot a} & 0 & 0 & 0 \\ 0 & \frac{1}{a} & 0 & 0 \\ 0 & 0 & -\frac{far+near}{far-near} & -\frac{2 \cdot far \cdot near}{far-near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$ 
4:   return projection

```

45.3 Transforming Objects - Camera Space to Screen Space

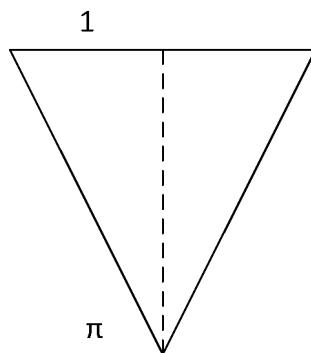
Let us now experiment with our algorithm. Consider the following properties that define our projection:

$$\begin{aligned}fov &= \frac{\pi}{4} \text{ rad} \\ aspect &= \frac{16}{9} = 1.778 \\ far &= 10000 \\ near &= 2.414\end{aligned}$$

Why do we use 2.414 for the Near Plane?

A simple bit of mathematics will answer this question!

Consider our view to the near plane as a triangle, with the viewer at the apex. We know that the near plane goes from -1 to 1 , so is two units across. We also know that the angle at the apex is $\frac{\pi}{4}$ as defined above. Let us split the triangle in half to give us the following;



Simple trigonometry can give us the distance from the viewer position (apex of the triangle) to the near plane. We have the following:

$$\tan \frac{\pi}{8} = \frac{1}{n} \Rightarrow 1 = n \tan \frac{\pi}{8} \Rightarrow n = \frac{1}{\tan \frac{\pi}{8}} \approx 2.414$$

Obviously if different field of view values are used, the near distance should be adjusted accordingly. However, you can change the near distance value to provide a different projection effect (for example a higher value will give a fish-eye lens effect).

We can calculate a as follows:

$$a = \tan(fov/2) = \tan \frac{\pi}{8} = 0.414$$

Our matrix definition then becomes:

$$\begin{bmatrix} \frac{1}{\text{aspect}\cdot a} & 0 & 0 & 0 \\ 0 & \frac{1}{a} & 0 & 0 \\ 0 & 0 & -\frac{\text{far}+\text{near}}{\text{far}-\text{near}} & -\frac{2\cdot \text{far}\cdot \text{near}}{\text{far}-\text{near}} \\ 0 & 0 & -1 & 0 \end{bmatrix} = \begin{bmatrix} \frac{1}{1.778\cdot 0.414} & 0 & 0 & 0 \\ 0 & \frac{1}{0.414} & 0 & 0 \\ 0 & 0 & -\frac{10002.414}{9997.586} & -\frac{2\cdot 10000\cdot 2.414}{9997.586} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} 1.359 & 0 & 0 & 0 \\ 0 & 2.414 & 0 & 0 \\ 0 & 0 & -1.000 & -4.829 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Let us now consider a coordinate that has been translated into camera space, giving the result $(20, 10, -100)$ (remember that our camera looks towards negative z). Transforming this by the projection transformation matrix gives us the following:

$$\begin{bmatrix} 1.359 & 0 & 0 & 0 \\ 0 & 2.414 & 0 & 0 \\ 0 & 0 & -1.000 & -4.829 \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} 20 \\ 10 \\ -100 \\ 1 \end{bmatrix} = \begin{bmatrix} 27.18 \\ 24.14 \\ -95.171 \\ -100 \end{bmatrix}$$

This is a strange looking coordinate, and doesn't really relate to our understanding of the coordinate being in the range $(-1, -1, 1)$ to $(1, 1, 1)$. The result is due to us working in homogeneous coordinates, and we will need to transform the coordinate back to a regular coordinate to determine the final screen position.

45.3.1 Homogeneous Coordinates

To apply our “perspective” value (the z -distance) we need to transform our 4-dimensional homogeneous coordinate to a standard 3-dimensional coordinate. This is done by applying the w component. We do this by dividing the x, y, and z components by the w component. In our example, we have to do the following:

$$\begin{bmatrix} 27.18 \\ 24.14 \\ -95.171 \\ -100 \end{bmatrix} = \begin{bmatrix} \frac{27.18}{-100} & \frac{24.14}{-100} & \frac{-95.171}{-100} \end{bmatrix} = \begin{bmatrix} -0.271 & -0.241 & 0.952 \end{bmatrix}$$

Which is within the screen boundaries (remember the range) and thus visible to the camera. It is in fact very close to the centre of the screen (x and y coordinates close to 0).

45.4 Exercises

1. Given the following camera properties generate the relevant projection transformation matrix.
 - (a) **fov** $\frac{\pi}{2}$
aspect $\frac{16}{9}$
near 1
far 100000
 - (b) **fov** $\frac{2\pi}{3}$
aspect $\frac{16}{8}$
near 0.577
far 100
 - (c) **fov** $\frac{\pi}{4}$
aspect $\frac{4}{3}$
near 2.414
far 1000
 - (d) **fov** $\frac{\pi}{4}$
aspect $\frac{16}{9}$
near 0
far 10000
 - (e) **fov** $\frac{\pi}{2}$
aspect $\frac{16}{9}$
near 20
far 50
2. Using your generated projection matrices, transform the following camera coordinates into screen space.
 - (a) $(-20, -20, -20)$
 - (b) $(-100, 200, -120)$
 - (c) $(0, 400, -500)$
 - (d) $(250, 100, -300)$
 - (e) $(80, 80, -5)$

45.5 Answers

1. (a)

$$\begin{bmatrix} 0.562 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1.000 & -2.000 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

(b)

$$\begin{bmatrix} 0.289 & 0 & 0 & 0 \\ 0 & 0.577 & 0 & 0 \\ 0 & 0 & -1.012 & -1.161 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

(c)

$$\begin{bmatrix} 1.811 & 0 & 0 & 0 \\ 0 & 2.414 & 0 & 0 \\ 0 & 0 & -1.005 & -4.840 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

(d)

$$\begin{bmatrix} 1.359 & 0 & 0 & 0 \\ 0 & 2.414 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

(e)

$$\begin{bmatrix} 0.563 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -2.333 & -66.667 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

2. (a) i. $(-0.562, -1, 0.9)$ - visibleii. $(-0.289, -0.577, 0.954)$ - visibleiii. $(-1.811, -2.414, 0.763)$ - not visibleiv. $(-1.359, -2.414, 1)$ - not visiblev. $(-0.563, -1, -1)$ - visible (just)(b) i. $(-0.468, 1.667, 0.983)$ - not visibleii. $(-0.241, 0.962, 1.007)$ - not visibleiii. $(-1.509, 2.017, 0.965)$ - not visibleiv. $(-1.133, 2.017, 1)$ - not visiblev. $(-0.469, 1.667, 1.777)$ - not visible(c) i. $(0, 0.8, 0.996)$ - visibleii. $(0, 0.462, 1.010)$ - not visible

- iii. $(0, 1.931, 0.995)$ - not visible
 - iv. $(0, 1.931, 1)$ - not visible
 - v. $(0, 0.8, 2.200)$ - not visible
- (d) i. $(0.468, 0.333, 0.993)$ - visible
- ii. $(0.241, 0.192, 1.008)$ - not visible
 - iii. $(1.509, 0.805, 0.989)$ - not visible
 - iv. $(1.133, 0.805, 1)$ - not visible
 - v. $(1.480, 0.333, 2.112)$ - not visible
- (e) i. $(8.992, 16, 0.6)$ - not visible
- ii. $(4.624, 9.232, 0.780)$ - not visible
 - iii. $(28.976, 38.624, 0.037)$ - not visible
 - iv. $(21.744, 38.624, 1)$ - not visible
 - v. $(9.008, 16, -11.000)$ - not visible

Note that the answers from using the matrix generated in 1(d) all have a z-coordinate of 1. Do you know why? Can you also determine why some of the matrices produce visible coordinates and many do not? What is the value that could be changed in most cases to improve visibility?

Lesson 46

Target Camera

The first camera we will look at is the simplest. A target camera doesn't add any real functionality to our base camera object. It simply has a position, something it looks at, at the orientation.

46.1 What is a Target Camera?

As mentioned, a target camera is our simplest camera. You can think of it as a fixed position camera. It doesn't move around the world, nor does it rotate. As such, we don't have to add much to our understanding. Our other cameras provide different interaction mechanisms to allow us to manipulate the camera accordingly.

46.2 Target Camera Properties

Our target camera does not add any new properties. It simply has the three standard values we defined previously:

position the position of the camera in the world

target the position that the camera is looking at

up the orientation of the camera

46.3 Updating the Target Camera

Although our target camera does have everything we need to actually calculate our view matrix, we do perform some calculation prior to doing this. This calculation is to generate the actual up direction of the camera. Consider Figure 46.1. On the left, the up direction is not at $\frac{\pi}{2}$ radians to the forward direction. We need to change this, as shown to the right of Figure 46.1. We do this using some simple vector manipulation. Algorithm 11 provides the pseudocode.

Below is the actual code for updating the target camera.

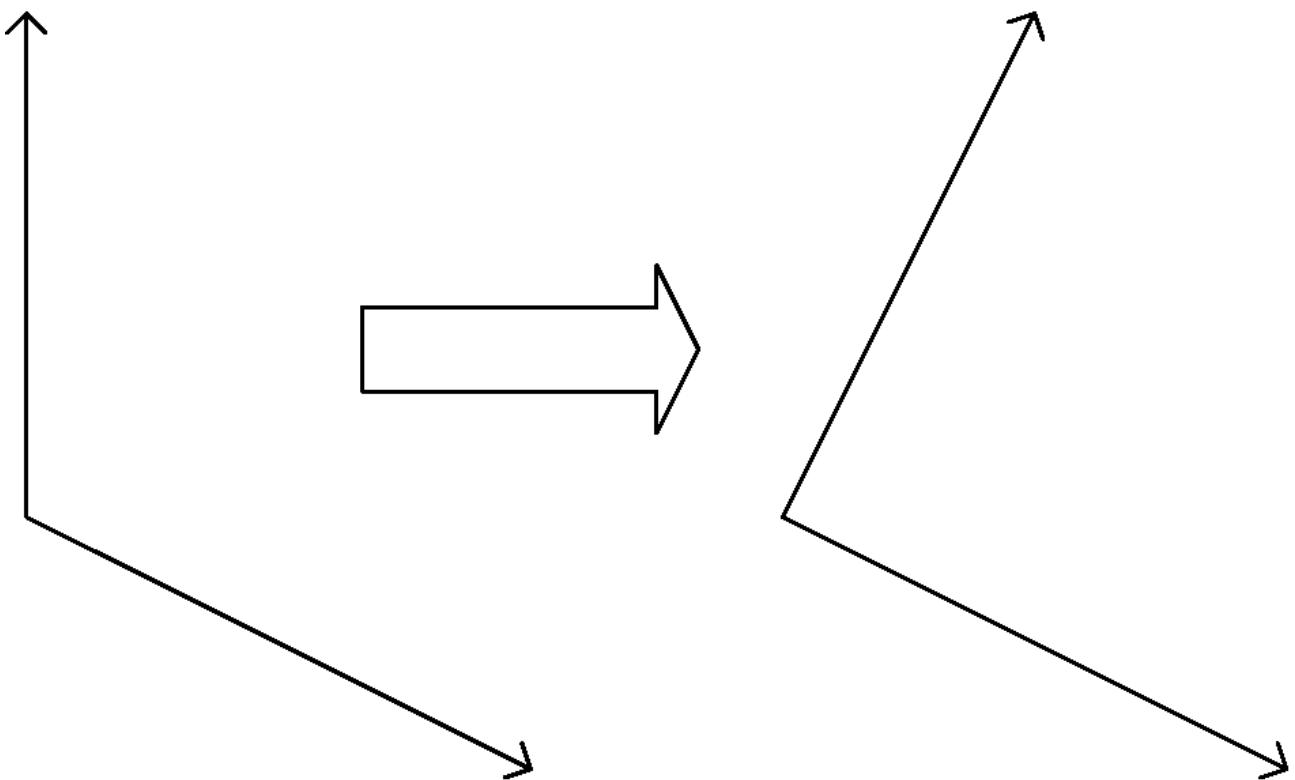


Figure 46.1: Converting to Actual Up

Algorithm 11 Target Camera Update

```

1: procedure TARGET_CAMERA(position, target, up)
2:   forward  $\leftarrow$  target – position
3:   side  $\leftarrow$  up  $\times$  forward
4:   frame_up  $\leftarrow$  (forward  $\times$  side)
5:   view  $\leftarrow$  LOOK_AT(position, target, frame_up)

```

▷ See Algorithm 9

```

1 void target_camera::update(float delta_time)
2 {
3   // Calculate forward and side vectors
4   glm::vec3 forward = _target - _position;
5   glm::vec3 side = glm::cross(_up, forward);
6   // Use forward and side to calculate up vector
7   glm::vec3 up = glm::cross(forward, side);
8   up = glm::normalize(up);
9   // Set view matrix accordingly
10  _view = glm::lookAt(_position, _target, up);
11 }

```

46.4 Exercise

In this lesson you need to add some keyboard controls to enable manipulation of the camera. All we are going to do is change the position of the camera based on whether one of the keys 1-4 have been pressed. This is a fairly simple task. The four camera positions we are going to

use are:

1. $(50, 10, 50)$
2. $(-50, 10, 50)$
3. $(-50, 10, -50)$
4. $(50, 10, -50)$

The output for this lesson (at camera position 3) is shown in Figure 46.2.

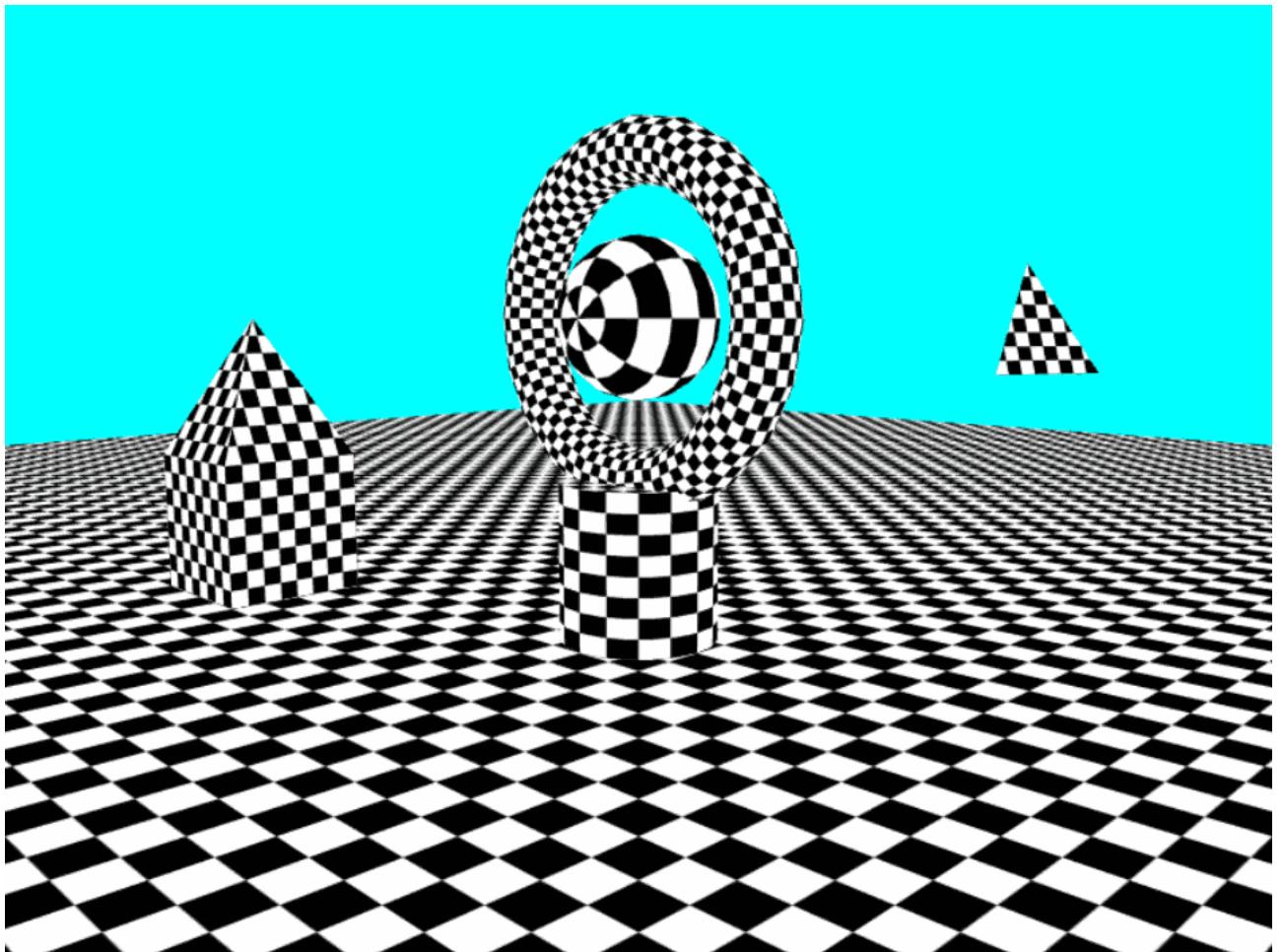


Figure 46.2: Output from Target Camera Lesson

Lesson 47

Free Camera

Our next camera is a bit more interesting - a first person like camera. This camera type allows free movement by allowing us to rotate around and move forward in the direction we are facing. This is the standard camera used in first-person shooter type games - except we will be able to 'fly' more (move in any direction).

47.1 What is a Free Camera?

A true free camera allows movement in any direction, with rotation upon any axis. A good way to think of a free camera is like being in control of a plane (or even better a helicopter). You generally move forward in the direction you are facing, with the ability to change that direction by adjusting your yaw, pitch or roll. Thinking more like a helicopter, you can also move backwards, up and down, and side to side.

47.1.1 Restricted Free Camera

It is a very rare occurrence that we would like our camera to rotate around the z-axis. As such, the free camera in the graphics framework does not provide mechanisms to rotate on the z-axis. This is just to make the camera a little simpler, and therefore easier to explain. You will find that few games (outside flight simulators or games with flight like vehicles) have z-axis rotation in a first person view.

47.2 Free Camera Properties

Our free camera has the same job as a standard camera - the creation of a view matrix for the given frame by use of a camera position, target and up direction. However, the free camera does this using other properties. These are as follows:

pitch current rotation on the x-axis

yaw current rotation on the y-axis

translation direction and magnitude to move this frame. This is represented by a 3-dimensional vector

We use these three values to generate our position, target and up. The free camera also provides methods to enable the manipulation of these values.

47.3 Controlling the Free Camera

Besides standard getter and setter methods for the additional properties, the free camera also provides the following two methods:

rotate rotates the camera by the given yaw and pitch values

move moves (translates) the camera by the given 3-dimensional vector

These are the two methods you should use to control the camera. Although you can set the values directly, it is probably easier to use these two methods instead.

47.4 Updating the Free Camera

As before, the job of the **update** method is to generate a view transformation matrix. It does this using the pitch, yaw and translation values to calculate the position, target and up values. Let us think about the stages we will have to go through:

1. Calculate the forward vector. This can be calculated using standard trigonometry using the yaw and pitch
2. Calculate the side vector. We can do this by transforming the standard side vector by the rotation transformation matrix around the y-axis (yaw).
3. Calculate the up vector. This is simply the cross product of the side and forward vectors.

Once we have the forward, side and up vectors we can generate the other values accordingly. The pseudocode for our free camera is given in Algorithm 12.

Algorithm 12 Free Camera Update

```
1: procedure FREE_CAMERA(position, pitch, yaw, translation)
2:   forward  $\leftarrow (\cos \text{pitch} \times -\sin \text{yaw}, \sin \text{pitch}, -\cos \text{yaw} \times \cos \text{pitch})$ 
3:   right  $\leftarrow \begin{bmatrix} \cos \text{yaw} & 0 & \sin \text{yaw} \\ 0 & 1 & 0 \\ -\sin \text{yaw} & 0 & \cos \text{yaw} \end{bmatrix} \times \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$ 
4:   up  $\leftarrow \text{right} \times \text{forward}$ 
5:   trans  $\leftarrow \text{translation.x} \times \text{right}$ 
6:   trans  $\leftarrow \text{trans} + \text{translation.y} \times \text{up}$ 
7:   trans  $\leftarrow \text{trans} + \text{translation.z} \times \text{forward}$ 
8:   position  $\leftarrow \text{position} + \text{trans}$ 
9:   target  $\leftarrow \text{position} + \text{forward}$ 
10:  translation  $\leftarrow (0, 0, 0)$ 
11:  view  $\leftarrow \text{LOOK\_AT}(\text{position}, \text{target}, \text{up})$ 
```

Below is our code listing for the free camera update.

```
1 void free_camera::update(float delta_time)
2 {
```

```

3 // Calculate the forward direction - spherical coordinates to ←
4 // Cartesian
5 glm::vec3 forward(cosf(_pitch) * -sinf(_yaw), sinf(_pitch), -cosf(←
6 // Normalize forward
7 forward = glm::normalize(forward);
8
9 // Calculate standard right. Rotate right vector by yaw
10 glm::vec3 right = glm::vec3(glm::eulerAngleY(_yaw) * glm::vec4(1.0←
11 f, 0.0f, 0.0f, 1.0f));
12 // Normalize right
13 right = glm::normalize(right);
14
15 // Up vector is up rotated by pitch
16 _up = glm::cross(right, forward);
17 // Normalize up
18 _up = glm::normalize(_up);
19
20 // We can now update position based on forward, up and right
21 glm::vec3 trans = _translation.x * right;
22 trans += _translation.y * _up;
23 trans += _translation.z * forward;
24 _position += trans;
25
26 // Target vector is just our position vector plus forward vector
27 _target = _position + forward;
28
29 // Set the translation vector to zero for the next frame
30 _translation = glm::vec3(0.0f, 0.0f, 0.0f);
31
32 // We can now calculate the view matrix
33 _view = glm::lookAt(_position, _target, _up);
34 }

```

47.5 Exercise

In this exercise you need to add controls for the free camera, working with the mouse and keyboard. To do this, we will have to introduce a couple of new techniques within the render framework - mainly around working with the mouse.

47.5.1 Working with the Mouse

So far we have worked with the keyboard as a source of input. Now let us use the mouse to interact with our camera. To do this, we are going to introduce two ideas - disabling the cursor and capturing the cursor position.

Disabling the Mouse Cursor

GLFW is our primary method of interacting with our application. GLFW allows us to manipulate input modes, one of which is the turning on and off of the mouse pointer. To do this, we use the following command:

```
1 glfwSetInputMode(GLFWwindow*, GLFW_CURSOR, GLFW_CURSOR_DISABLED);
```

The first parameter is a pointer to the window created by the application. Remember that the render framework maintains this for us. The second and third parameters are provided constants from GLFW.

This will have to go in the `initialise` method declared in this application. The `initialise` method is called first by the render framework, and allows us to set some state and other conditions required in our render framework. It is called before `load_content`, which requires initialisation to occur before content can be created. The render framework always initialises - in particular initialising the various libraries we use in the framework.

Getting the Mouse Position

Even though we disable the mouse cursor (the visible representation of the mouse position), the position of the mouse is still updated when we move the mouse. This enables the mouse to change to values that normally wouldn't be possible (outside the window bounds). To capture the mouse, we use the following command:

```
1 glfwGetCursorPos(GLFWwindow*, double*, double*);
```

The first parameter is a pointer to our window again. The second and third parameters are filled by the call to get the mouse position. You will need to create variables for these and pass them into the call. You will need to do this in the `initialise` and `update` methods.

47.5.2 Update

Our `update` method requires some work to convert from a change in the mouse position (the amount of rotation on the x and y axes) to allow movement of the mouse. Algorithm 13 provides the pseudocode for rotating the camera with the mouse.

Algorithm 13 Using the Mouse to Control Camera Rotation

```

1: procedure MOUSE_CONTROLLED_CAM(cam, prev_x, prev_y, ratio_width, ratio_height)
2:   current_x  $\leftarrow$  0
3:   current_y  $\leftarrow$  0
4:   GLFWGETCURSORPOS(window, current_x, current_y)
5:   delta_x  $\leftarrow$  current_x  $-$  prev_x                                 $\triangleright$  Change in x for the frame
6:   delta_y  $\leftarrow$  current_y  $-$  prev_y                                 $\triangleright$  Change in y for the frame
7:   delta_x  $\leftarrow$  delta_x  $\times$  ratio_width           $\triangleright$  ratio_width is the pixels to radians conversion
       factor - provided
8:   delta_y  $\leftarrow$  delta_y  $\times$  ratio_height       $\triangleright$  ratio_height is the pixels to radians conversion
       factor - provided
9:   CAM.ROTATE(delta_x, delta_y)                   $\triangleright$  rotates the camera by the necessary amount

```

You should be able to do the movement part of the code yourself by now. Running this application should allow you to move the camera around with the keyboard and mouse. An example output is shown in Figure 47.1.

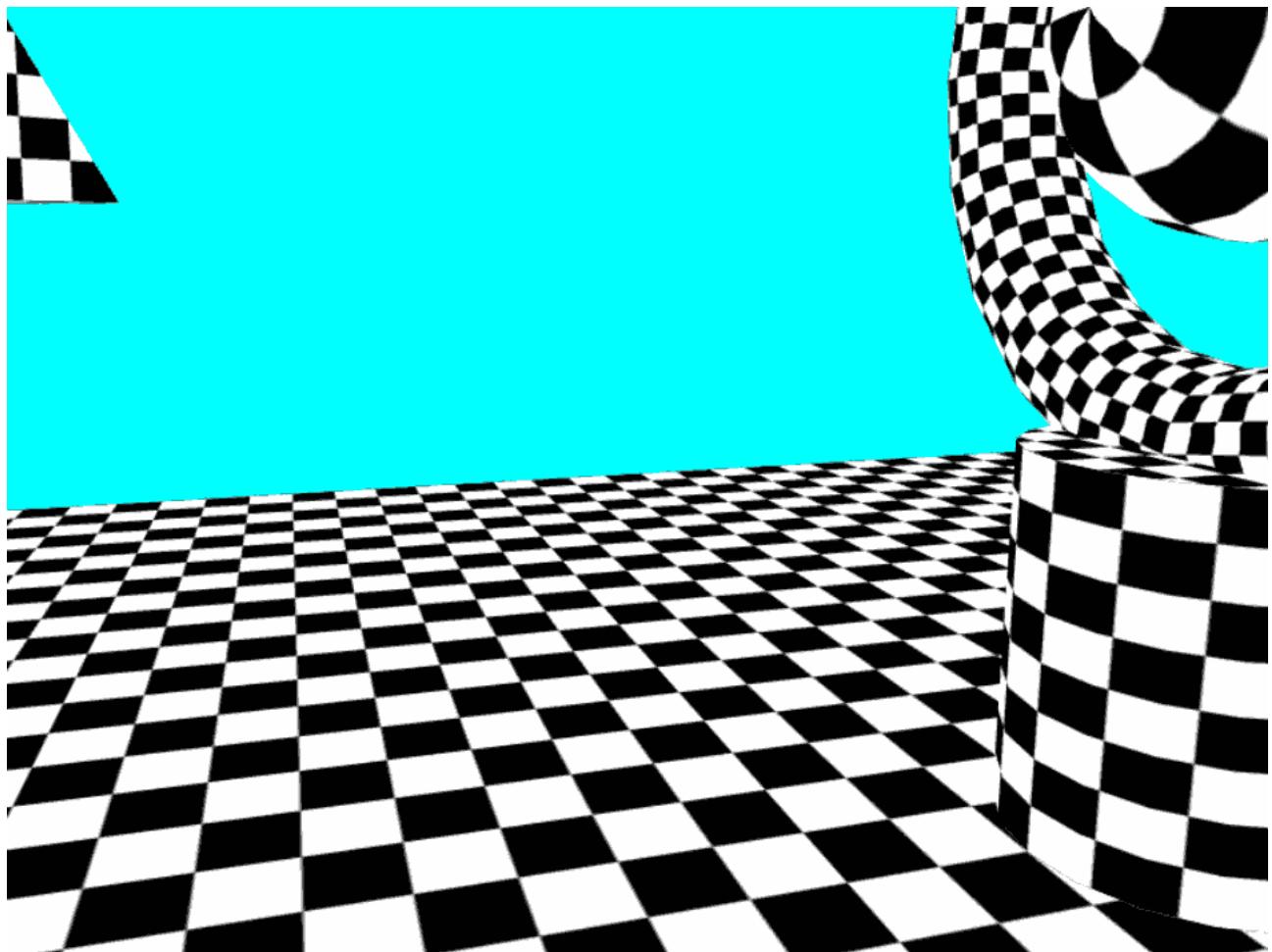


Figure 47.1: Output from Free Camera Lesson

Lesson 48

Chase Camera

Our third camera type is what we call a chase camera. Like the free camera, the chase camera is a very common camera type in computer games, allowing us to look at an object as it moves around the world.

48.1 What is a Chase Camera?

A chase camera is often called a 3rd person camera, and is seen in numerous games. The general idea of a chase camera is that there is an object in the world that the camera is attached to. As the object moves around the world, the camera follows it. The camera can also rotate around the object as well. Figure 48.1 illustrates the general idea of a chase camera.

48.2 Chase Camera Properties

The chase camera needs a few other properties to enable the chase like behaviour. These values are to do with how the camera is offset from the target object. The chase camera requires the following additional properties:

target position the position of the object being “chased”

position offset the offset of the camera from its desired position

target rotation the rotation of the object being “chased”

target offset the offset of the camera relative to the object being “chased”

relative rotation the rotation of the camera relative to the object being “chased”

springiness springiness factor of the camera - determines how quickly the camera snaps to its desired position

48.3 Controlling the Chase Camera

The chase camera is similar to the free camera in its general controls - it requires the ability to move (with the target object) and the ability to rotate around the object. These operations

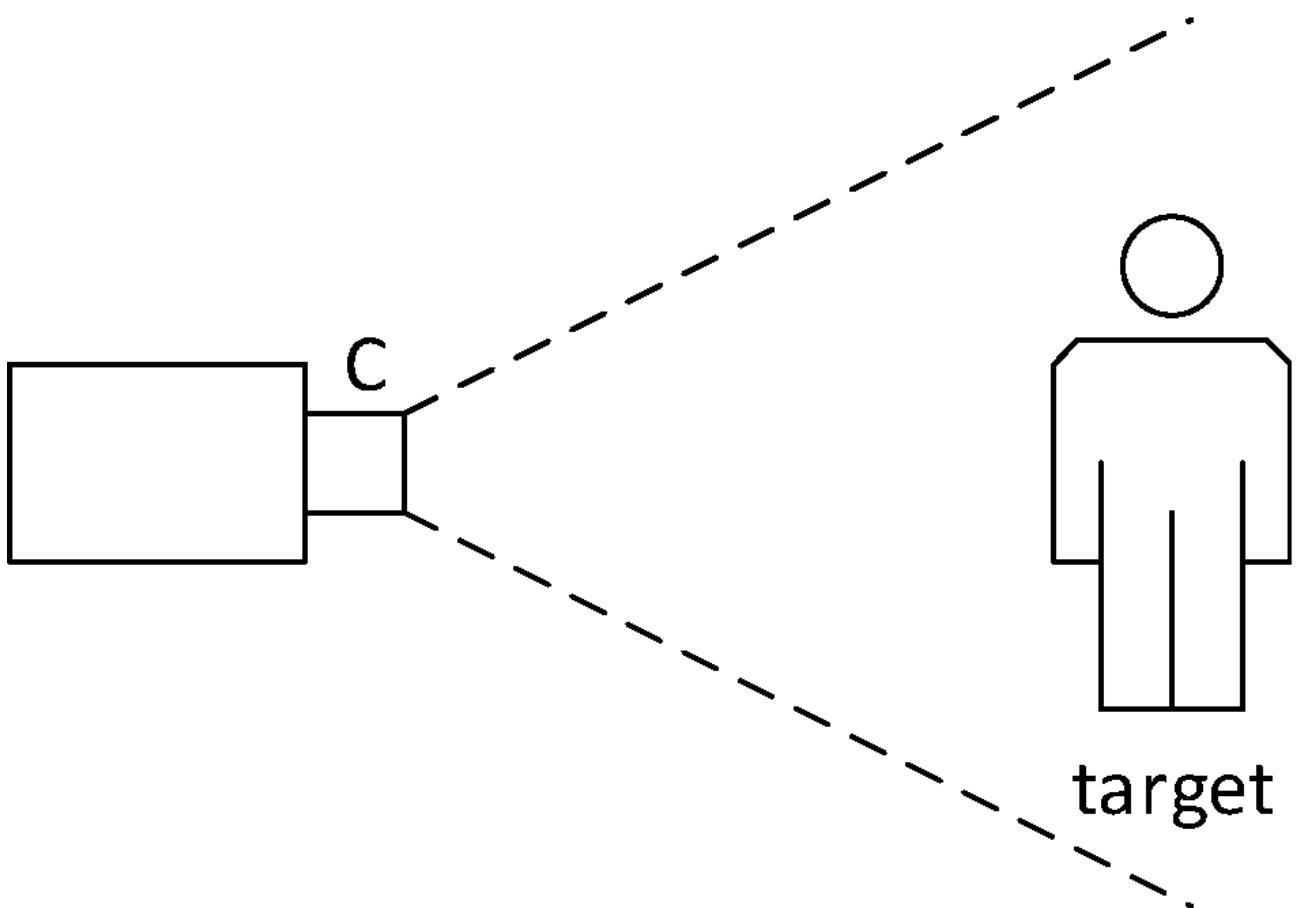


Figure 48.1: Chase Camera

are defined as:

move moves the camera by updating the position and rotation of the “chased” object

rotate rotates the camera around the target

Note that the **move** operation requires both the position of the target and the current rotation of the target.

48.4 Updating the Chase Camera

As with our other camera types, the goal of the **update** method is to generate the view matrix. To do this, we need to generate the **position**, **target** and **up** values. Let us consider how we generate these values:

positon requires the target position and the position offset (transformed by the rotation) from the target, manipulated by the springiness

target requires the target position and target offset (transformed by the rotation)

up standard up transformed by the rotation

Algorithm 14 represents the pseudocode for the chase camera.

The **update** method implemented in the render framework is below:

Algorithm 14 Chase Camera Update

```

1: procedure CHASE_CAMERA(position, position_offset, target_position, target_offset,
   target_rotation, relative_rotation, springiness)
2:   rotation  $\leftarrow$  QUATERNION(target_rotation + relative_rotation)
3:   desired_position  $\leftarrow$  target_position + (rotation  $\times$  position_offset)
4:   position  $\leftarrow$  LERP(position, desired_position, springiness) ▷ lerp is linear
   interpolation. GLM provides mix
5:   target_offset  $\leftarrow$  rotation  $\times$  target_offset
6:   target  $\leftarrow$  target_position + target_offset
7:   up  $\leftarrow$  rotation  $\times$  (0.0, 1.0, 0.0)
8:   view  $\leftarrow$  LOOK_AT(position, target, up)

```

```

1 // Updates the chase camera
2 void chase_camera::update(float delta_time)
3 {
4     // Calculate the combined rotation as a quaternion
5     glm::quat rotation(_target_rotation + _relative_rotation);
6
7     // Now calculate the desired position
8     glm::vec3 desired_position = _target_pos + (rotation * _pos_offset);
9     // Our actual position lies somewhere between our current position
10    // and the
11    // desired position
12    _position = glm::mix(_position, desired_position, _springiness);
13
14    // Calculate new target offset based on rotation
15    _target_offset = rotation * _target_offset;
16    // Target is then the target position plus this offset
17    _target = _target_pos + _target_offset;
18
19    // Calculate up vector based on rotation
20    _up = rotation * glm::vec3(0.0f, 1.0f, 0.0f);
21
22    // Calculate view matrix
23    _view = glm::lookAt(_position, _target, _up);
}

```

48.5 Exercise

In this lesson your job is to implement the controls for the chase camera. The idea is similar to that of the free camera, but now you have to “chase” an object (defined as `target_mesh`). You will need to convert between the quaternion representing the rotation of the object and a 3D vector representing the Euler angles to pass into the camera `move` method. This call is below:

```
1 eulerAngles(quat);
```

A sample output from this lesson is shown in Figure 48.2.

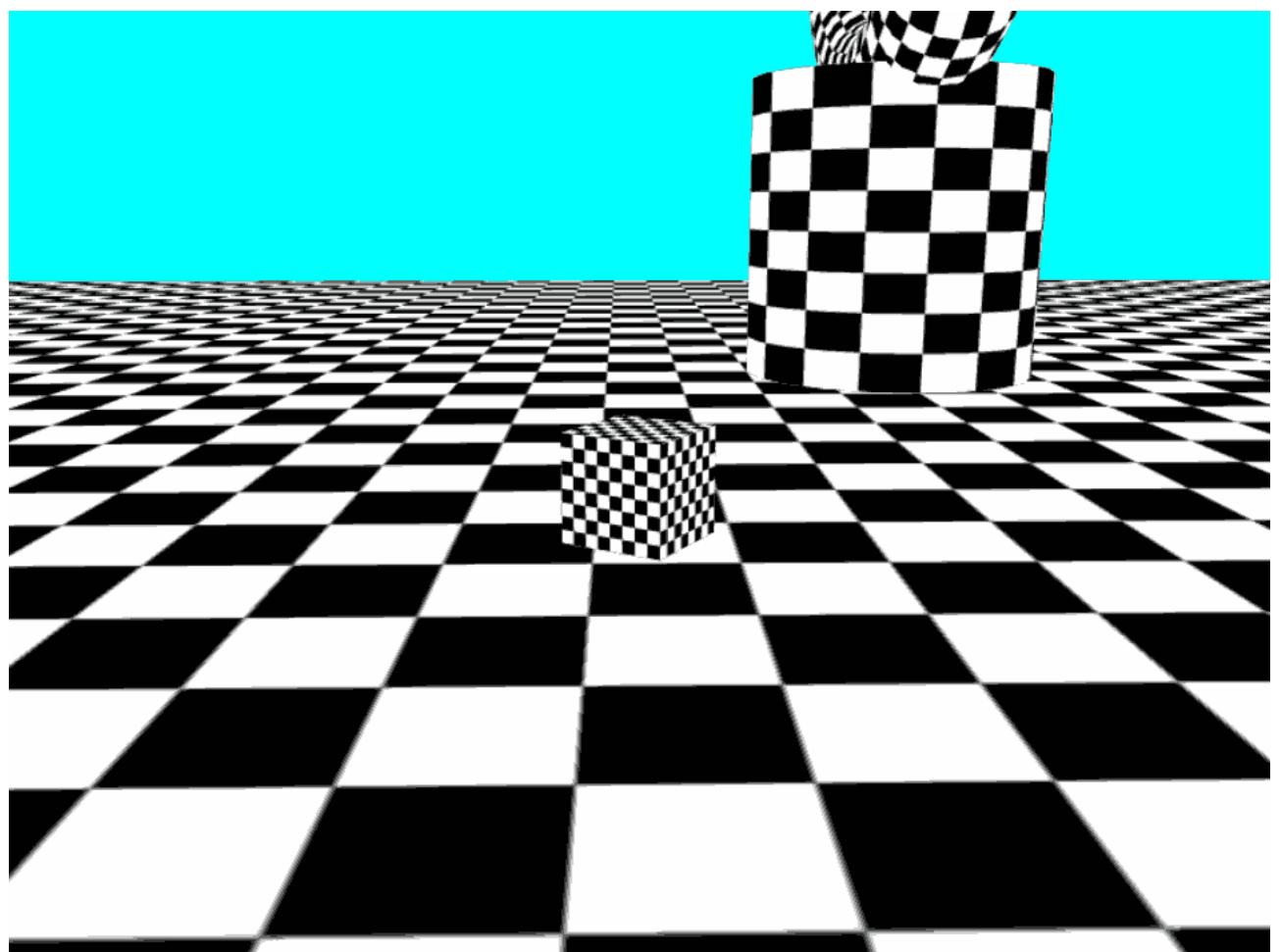


Figure 48.2: Output from Chase Camera Lesson

Lesson 49

Arc-ball Camera

Our final camera type is an arc-ball camera. A arc-ball camera operates in a similar way to our chase camera, but we aren't chasing an object now. We allow rotation around a specific target at a given distance.

49.1 What is an Arc-ball Camera?

An arc-ball camera is rarely used in games and therefore you might be unfamiliar with the concept. However, it is very useful in graphics rendering as it allows us to examine a particular object (moving back and forward from the object and rotating around it). It is the sort of camera a 3D graphics designer would be more familiar with. Figure 49.1 illustrates the basic idea behind an arc-ball camera.

49.2 Arc-ball Camera Properties

An arc-ball camera is a simpler model than the chase camera, and only requires a few new properties. These are:

rot_x rotation around the x-axis of the target

rot_y rotation around the y-axis of the target

distance the distance of the camera from the target

49.3 Controlling the Arc-ball Camera

The arc-ball camera provides methods to move and rotate much like our other camera types. However, the **move** method only takes a scalar value as we are only concerned with the distance from the target. The camera also provides a **translate** method to move the target looked at (although you would normally use a chase camera for a moving target). The operations are:

move moves the camera towards or away from the camera

rotate modifies the rotation values around the target

translate translates the target looked at by the camera

49.4 Updating the Arc-ball Camera

Updating the arc-ball camera is simple in comparison to the chase camera. As always, we must generate our view matrix by calculating the relevant *position*, *target* and *up* values. These are generated as follows:

position is the target plus the distance down the z-axis transformed by the rotation around the object

target is set directly

up is the standard up transformed by the rotation around the object.

The algorithm for updating the arc-ball camera is given in Algorithm 15.

Algorithm 15 Arc-ball Camera Update

```

1: procedure ARC-BALL-CAMERA(target, rot_x, rot_y, distance)
2:   rotation  $\leftarrow$  QUATERNION((rot_x, rot_y, 0))
3:   position  $\leftarrow$  target + (rotation  $\times$  (0, 0, distance))  $\triangleright$  rotate forward vector of magnitude distance
4:   up  $\leftarrow$  rotation  $\times$  (0, 1, 0)
5:   view  $\leftarrow$  LOOK-UP(position, target, up)

```

The actual implementation of the arc-ball camera is given below:

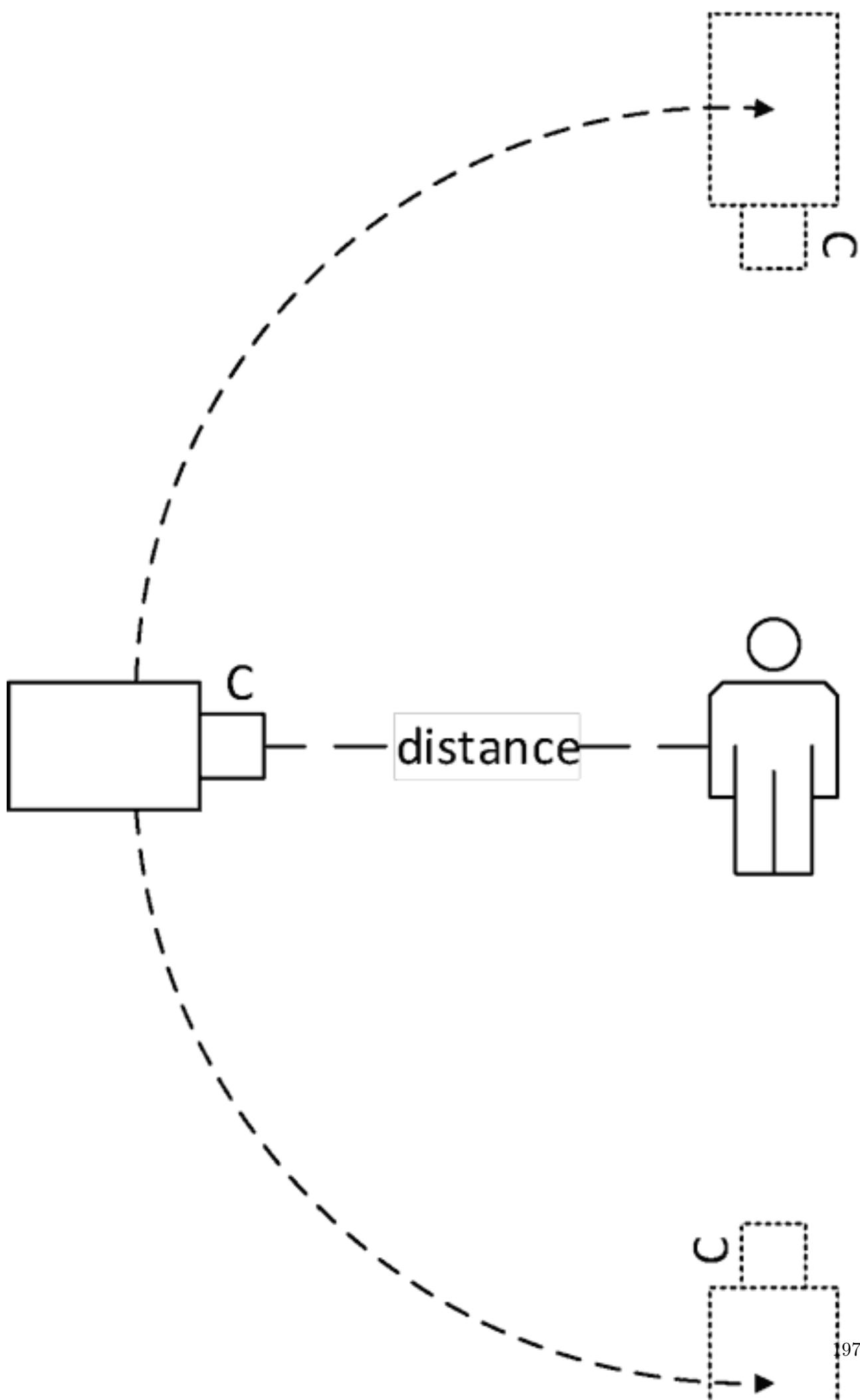
```

1 // Updates the arc_ball_camera
2 void arc_ball_camera::update(float delta_time)
3 {
4     // Generate quaternion from the rotation
5     glm::quat rotation(glm::vec3(_rot_X, _rot_Y, 0.0f));
6     // Multiply the rotation by translation vector to generate ←
7     // position
8     _position = _target + (rotation * glm::vec3(0.0f, 0.0f, _distance)←
9     );
10    // Up is standard up multiplied by rotation
11    _up = rotation * glm::vec3(0.0f, 1.0f, 0.0f);
12    // Calculate the view matrix
13    _view = glm::lookAt(_position, _target, _up);
14 }

```

49.5 Exercise

Again, the aim of this lesson is to add the camera controls. You should know everything you need by now to achieve this. An example output from this lesson is given in Figure 49.2.



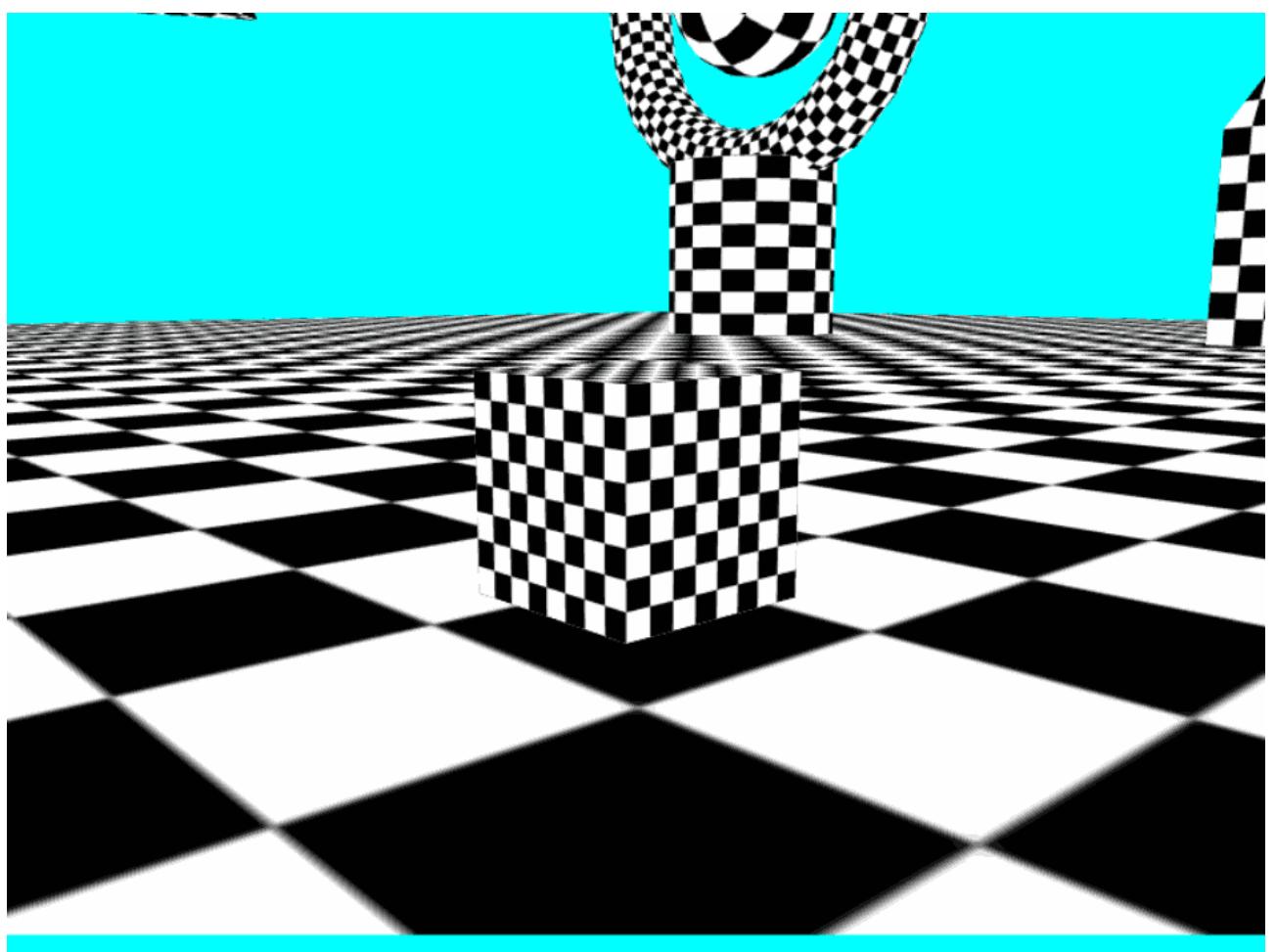


Figure 49.2: Output from Arc-ball Camera Lesson

Lesson 50

Picking

We end our work looking at cameras by implementing a system where we can select objects on the screen. This is a process called picking. We will look at the underlying theory and mathematics allowing us to understand how we select an object, and then implement these techniques to allow us to implement picking.

50.1 What is Picking?

Picking is the term used to describe the technique of selecting an object in our 3D world by selecting it with the mouse. As you can imagine, this does require us to somehow convert the mouse position (a 2D coordinate related to the window) to some form of representation in the 3D world. There are a few different ways we could achieve this selection, but we are going to use the most common - ray casting.

50.2 Picking with Ray-casting

The idea of raycasting is simple. We need a starting point for the ray (the point clicked on the screen) and the direction that the ray is heading. Figure 50.1 illustrates the basic idea.

We then test the ray to see if it has intersected with an object. If it has, then we know that the object has been *picked*. We will look at how we determine if an object has been picked shortly. First of all, we have to generate our ray.

50.3 Converting a Screen Position to a World Ray

When we looked at coordinate spaces, we discussed the idea of transforming from one coordinate space to another - going from the model space, through world space, camera space and finally ending up in screen space. We also discussed how we can go backwards using the inverses of the relevant transformation matrices. This is what we have to do now - we have to go back from screen coordinates to world coordinates.

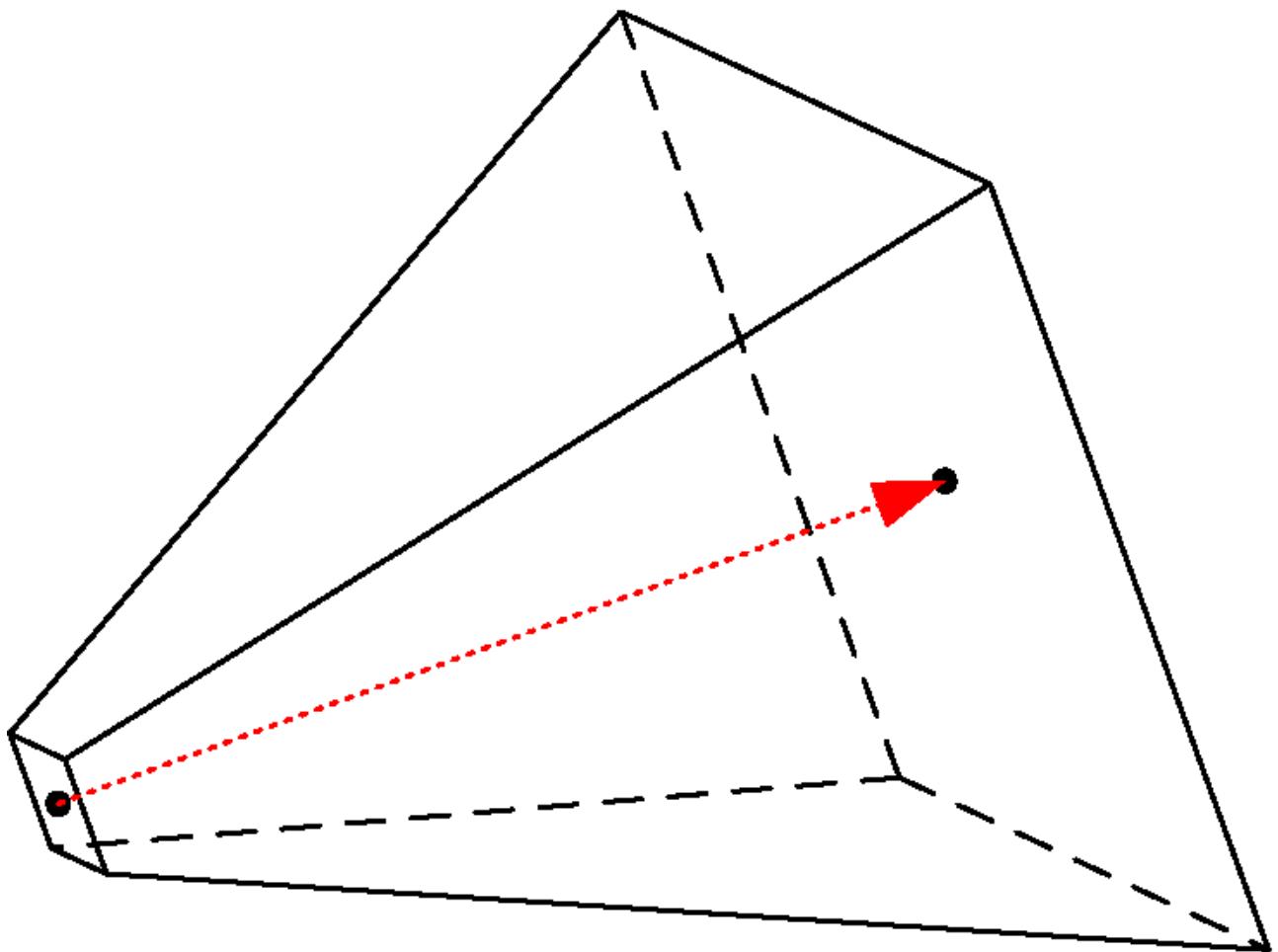


Figure 50.1: Raycasting in the View Frustum

50.3.1 Back to Coordinate Spaces

Remember that we defined a transformation matrix that can convert from one matrix to another as follows:

$$\begin{bmatrix} U_1 & V_1 & W_1 & T_1 \\ U_2 & V_2 & W_2 & T_2 \\ U_3 & V_3 & W_3 & T_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We also defined that we can combine multiple coordinate space transformations into a single transformation by multiplying the matrices together. For example we defined the transformation from the model space to screen space as follows:

$$MVP = P \times V \times M$$

We now need to consider the reverse of this process using inverse matrices.

50.3.2 Inverse Transforms

A linear transformation that is invertible enables us to transform back to our original coordinates:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} U_1 & V_1 & W_1 \\ U_2 & V_2 & W_2 \\ U_3 & V_3 & W_3 \end{bmatrix}^{-1} \left(\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} - \begin{bmatrix} T_1 \\ T_2 \\ T_3 \end{bmatrix} \right)$$

As such, we can transform back from our screen coordinates to our model coordinates if we invert the calculation:

$$(MVP)^{-1} = M^{-1} \times V^{-1} \times P^{-1} = (P \times V \times M)^{-1}$$

With our raycast however, we only need to convert from screen coordinates to world coordinates (the ray exists in world space). As such, the transformation matrix we need is as follows:

$$(VP)^{-1} = (P \times V)^{-1}$$

With this matrix in place, all we need to do now is work out our ray.

50.3.3 World Ray Calculation

A ray has two properties of interest:

origin where the ray originates

direction the direction of the ray as a unit vector

All we need to do is calculate these values. Remember that our screen coordinates operate in the range $(-1, -1, -1)$ to $(1, 1, 1)$. Therefore, we need to convert our mouse coordinates to this range. For the z values, we simply take the minimum (-1) for the origin, and use the centre (0) to calculate the direction. Therefore we are converting the following two vectors into world space:

start $(x', y', -1)$

end $(x', y', 0)$

We also make these homogeneous coordinates (1 for the w-component) to use them correctly. To calculate our direction, we will subtract the transformed start from the transformed end.

50.3.4 Screen Position to World Ray Algorithm

Algorithm 16 provides the pseudocode for converting a screen coordinate to a world ray. Take your time and study this algorithm as you will need to implement it.

Algorithm 16 Converting a Screen Position to a World Ray

```

1: procedure SCREEN_POSITION_TO_WORLD_RAY(mouse_x, mouse_y, screen_width,
   screen_height, view, proj, origin, direction)
2:   xx  $\leftarrow \frac{2 \cdot \text{mouse\_x}}{\text{screen\_width}} - 1$                                  $\triangleright \text{xx now in range } (-1, 1)$ 
3:   yy  $\leftarrow \frac{(2 \cdot (\text{screen\_height} - \text{mouse\_y}))}{\text{screen\_height}} - 1$            $\triangleright \text{yy now in range } (-1, 1)$ 
4:   ray_start_screen  $\leftarrow (xx, yy, -1, 1)$      $\triangleright \text{Homogeneous coordinate, starting at near plane}$ 
5:   ray_end_screen  $\leftarrow (xx, yy, 0, 1)$             $\triangleright \text{Homogeneous coordinate}$ 
6:   inverse_matrix  $\leftarrow (\text{proj} \times \text{view})^{-1}$        $\triangleright \text{Transform matrix from screen space to world}$ 
    $\text{space}$ 
7:   ray_start_world  $\leftarrow \text{inverse\_matrix} \times \text{ray\_start\_screen}$ 
8:   ray_start_world  $\leftarrow \frac{\text{ray\_start\_world}}{\text{ray\_start\_world.w}}$            $\triangleright \text{Convert from Homogeneous coordinates}$ 
9:   ray_end_world  $\leftarrow \text{inverse\_matrix} \times \text{ray\_end\_screen}$ 
10:  ray_end_world  $\leftarrow \frac{\text{ray\_end\_world}}{\text{ray\_end\_world.w}}$            $\triangleright \text{Convert from Homogeneous coordinates}$ 
11:  direction  $\leftarrow \text{ray\_end\_world} - \text{ray\_start\_world}$            $\triangleright \text{Returned to caller}$ 
12:  origin  $\leftarrow \text{ray\_start\_world}$            $\triangleright \text{Returned to caller}$ 

```

50.4 Defining an AABB

Now that we know how to generate our world ray, we need to focus on generating the correct bounding box to allow testing. We will work initially with the idea of an Axis-Aligned Bounding Box (or AABB). An AABB has the same directional axis as our world, and therefore allows simple calculation. However, as our objects may be transformed, we need to convert the AABB into an Object-Oriented Bounding Box (or OOBB).

The `mesh` object in the render framework comes with two methods that allow you to get what we can use to get the minimal and maximal coordinates of our bounding box:

`get_minimal` gets a 3D vector representing the minimal point of the bounding box

`get_maximal` gets a 3D vector representing the maximal point of the bounding box

These are the only two points we need to represent a bounding box. They represent the bottom left corner and top right corner respectively. With these values we just need to convert them into our OOBB based on the object transformation.

50.5 Converting an AABB to an OOBB

We already know how to transform our AABB to an OOBB - we use the model matrix. The model matrix provides us with our three new axes, as well as the position of the centre of the object in world space. As such, we now have everything we need to calculate an OOBB. We just need to perform the intersection test.

50.6 Determining Ray and OOBB Intersection

The general idea of ray and box intersection is simple. We examine each axis and determine where the plane intersects the planes defined by these axis. Figure 50.2.

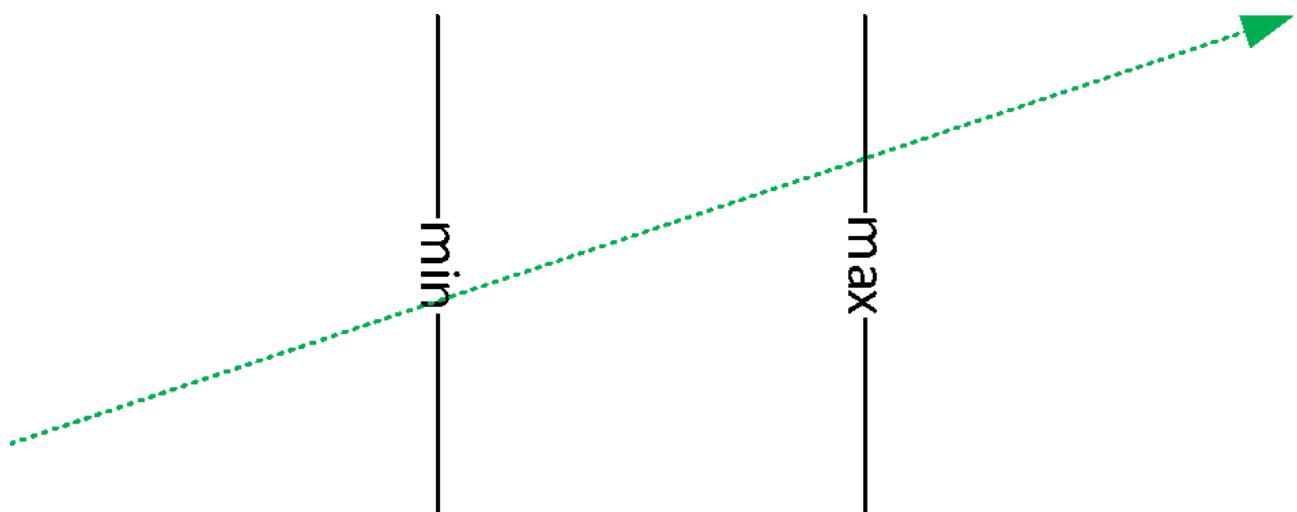


Figure 50.2: Ray Intersecting with Minimal and Maximal Planes

The general idea is to test where the ray intersects on each of the planes defined by the minimal and maximal bounds on that axis. If you are unsure what a plane is, it is formally defined as extending infinitely in two directions in a 3D space. Therefore, we are concerning ourselves with where the ray intersects the plane on that axis, not whether it intersects with the box. However, as we test each plane, we can determine if the point is still valid from the point of view of being still within the box. We simply iterate through the three axes until either we determine no intersection has occurred, or are left with the distance to the object.

Algorithm 17 provides the general algorithm for testing ray and OOOB intersection. Note that the algorithm only details the x-axis test, but the other axes are similar using the different values. For the model matrix, you have to consider that it has been defined as follows:

$$\begin{bmatrix} U_1 & V_1 & W_1 & T_1 \\ U_2 & V_2 & W_2 & T_2 \\ U_3 & V_3 & W_3 & T_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Thus we have the three axes of the transformed object defined as U , V and W . Again, you will need to implement this algorithm in the exercise. This one is a bit trickier, so take your time to understand it.

50.7 Exercise

Your task in this lesson is to implement the two algorithms. As a hint, implement the intersection testing algorithm in stages - start with the x-axis check and test it, add the y and test it, then finally add the z and test it.

Inverting Matrices with GLM

GLM provides a simple method to invert a matrix - `inverse`. Calling this with a matrix will return its inverse. For example:

```
1 model_inverse = inverse(model);
```

Algorithm 17 Testing OOOB and Ray Intersection

```

1: function TEST_RAY_BOUNDING_BOX(origin, direction, aabb_min, aabb_max, model,  

   distance)
2:   t_min  $\leftarrow 0$                                       $\triangleright$  floating point
3:   t_max  $\leftarrow 100000$                                  $\triangleright$  floating point
4:   OOOB_pos_world  $\leftarrow T$ 
5:   delta  $\leftarrow OOB\_pos\_world - origin$                  $\triangleright$  direction and magnitude to box

                                          $\triangleright$  Test planes perpendicular to OOOB x-axis

6:   x_axis  $\leftarrow \hat{U}$ 
7:   e  $\leftarrow x\_axis \cdot delta$                           $\triangleright$  cosine of x_axis and direction to box
8:   f  $\leftarrow direction \cdot x\_axis$                     $\triangleright$  cosine of x_axis and ray
9:   if  $\|f\| > 0.001$  then
10:    t1  $\leftarrow \frac{e+aabb\_min.x}{f}$                    $\triangleright$  test if ray and x_axis are parallel
11:    t2  $\leftarrow \frac{e+aabb\_max.x}{f}$                    $\triangleright$  intersection distance from left plane
12:    if t1  $> t_2$  then                                 $\triangleright$  intersection distance from right plane
13:      SWAP(t1, t2)                                  $\triangleright t_1$  needs to be nearest
14:      t_max  $\leftarrow \text{MIN}(t\_max, t_2)$             $\triangleright t\_max$  is nearest far intersection
15:      t_min  $\leftarrow \text{MAX}(t\_min, t_1)$             $\triangleright t\_min$  is furthest near intersection
16:      if t_max  $< t\_min$  then                     $\triangleright$  if far is closer than near no intersection
17:        return false
18:      else if  $-e + aabb\_min.x > 0$  or  $-e + aabb\_max < 0$  then
19:        return false                                   $\triangleright$  ray and x_axis are almost parallel

                                          $\triangleright repeat$  for y-axis

                                          $\triangleright repeat$  for z-axis

20:   distance  $\leftarrow t\_min$                             $\triangleright t\_min$  is distance to object
21:   return true

```

We haven't gone into inverse matrices. For those of you studying Games Development you should already know how. For those of you not, then it isn't necessary to know beyond what we have covered here.

The output from this exercise isn't really visible on the main screen, but the console window will show you some output. When you click on the screen, the console window will determine which object you have intersected with, and how far away it is from the camera. As an example output see Figure 50.3.

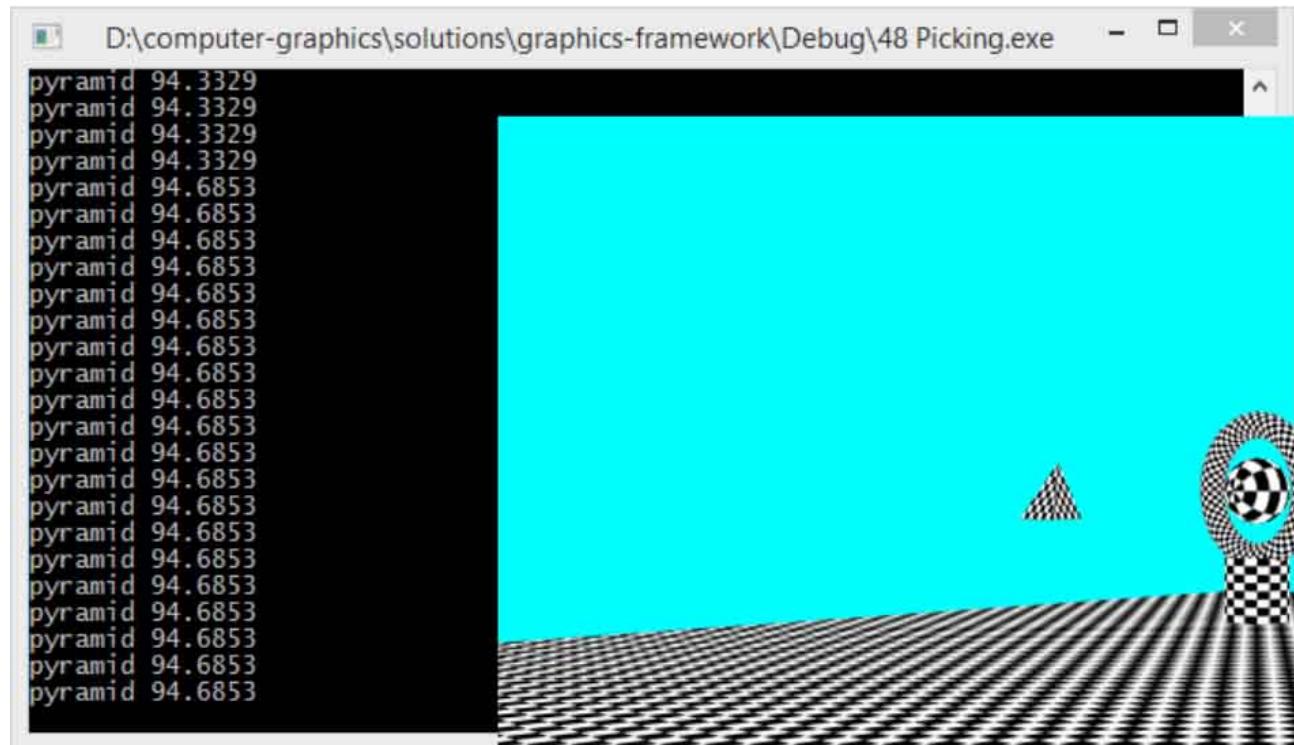


Figure 50.3: Output from Picking Lesson

Part VI

Lighting

Lesson 51

Lighting

We are now ready to start working on doing more useful effects. Over the next few lessons, we are going to focus on lighting, examining the different types, and implementing some lighting models in our shaders. We are building up current standard rendering techniques as we work now, and you will start to see your renders becoming more realistic as we work.

51.1 What is Lighting?

Lighting involves three pieces of physical information. These are:

1. Light emitters (for example, the Sun, a desk lamp, reflection from another surface)
2. Materials (describe how an object interacts with a light - what is absorbed, reflected)
3. Sensor (something to absorb the light - for example an eye, electronic sensor, film)

Over the next few lessons you will see that we use each of these pieces of physical information to create a lighting model for a particular rendered scene.

51.2 Light Source

A light source is something that emits rather than absorbing or scattering light. There are a number of different potential light sources in a scene, and we will explore these different sources over the next few lessons. As a simplistic model of light, we can consider the following three types:

Ambient light light that is everywhere in a scene at equal measure

Diffuse light light that has a direction and therefore reacts with an object's surface differently based on this direction

Specular light specular highlights - the shiny white reflection on a sphere for example. This requires knowing the position of the viewer and the direction of the light

Our simplistic model utilises RGB colour values, each of which has intensity. We have already been using colour in RGB values, so you should be familiar enough with the concepts involved. However, although we typically consider RGB colour to range in values from 0.0 to 1.0, it is quite possible for light to go above 1.0 in any of the components.

51.3 Materials

Materials describe the surfaces of objects. Up until now, we have considered our geometry as just that - a collection of geometric values that allow us to represent an object in 3D space. Materials are essentially a collection of colour values, textures, shaders and other information that allow us to describe what an object will look like, and how it will interact with the lighting of the scene.

Initially, we are just going to consider materials as a collection of colour values. These values react with our three lighting values defined in Section 51.2. We have the following:

Diffuse reflection describes how ambient and diffuse light of the scene interacts with the object

Specular material describes how the specular light of the scene interacts with the object

Shininess describes how shiny an object looks

We have used the term interacts in these definitions. There are basically two properties we are concerned with when thinking about how light interacts with an object. These are scattering and absorption.

51.3.1 Scattering

Scattering causes light to change direction, without reducing the amount of light involved. To put it simply, when light interacts with a mirror, you can consider the effect involved as a change in direction of the incoming light to the mirror. Essentially, a scattering effect causes the light to come in two different directions –light into the surface (called refraction) and light out of the surface (called reflection). This is an important concept when working with surfaces such as water. Figure 51.1, taken from Real-Time Rendering [1], should help illustrate reflection and refraction.

Absorption

Absorption relates to the light that is captured by an object, converted into some form of energy, and therefore no longer affects the lighting of a scene. Absorbed light doesn't change direction.

51.4 Sensors

For an object to be visible, some of the light that has interacted with the object needs to be absorbed by a light sensor. A light sensor will actually be composed of numerous smaller sensors, each of which detect a light signal, and convert it into a colour value (we can take a simplistic view here, and consider each pixel we render to be an individual light sensor). There is much more to a light sensor than this (apertures, lens, etc.). The recommended reading will explain further.

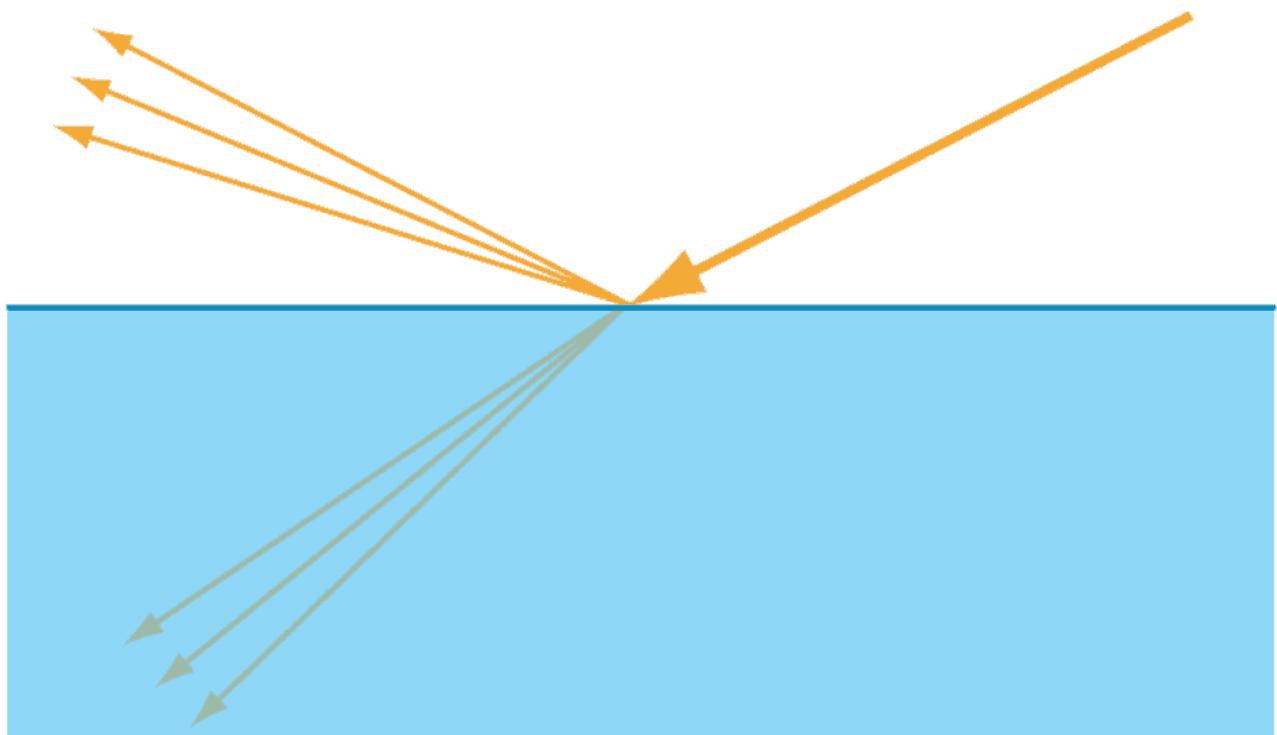


Figure 51.1: **Reflection and Refraction**

51.5 Recommended Reading

Chapter 5 of Real-Time Rendering [1] covers Visual Appearance, which includes lighting and other techniques.

Lesson 52

Simple Ambient Lighting

With our basic understanding of light, we can start working on the simplest form of lighting ambient light. Ambient light is light that is considered uniform across the whole scene, and does not take into account concepts such as light direction. This is in fact a *very* simplistic view of ambient light, but it will serve our purpose at the moment.

52.1 Examples

In Section 51.3, we described the idea of light being reflected from one object's surface onto the rest of the scene. This is essentially how we are able to see objects in the first place (we see the reflected light from the object). Obviously, not all the light that interacts with an object is captured by a light sensor. Light that interacts with an object may be redirected, and interact with another object, etc. This means that although there may only be one light source in a scene, light can interact with an object that is not in the same direction as the light source. This provides light that can be considered almost background to the scene. This is ambient light.

As an example, consider the desk you are likely sitting at. It will likely only have light shining directly down upon it. There will be areas under the desk that have no direct path to the light in the room, yet you can still see under the desk - that is there is still a light interacting with the surfaces under the desk. This is the ambient light of the room.

52.2 Ambient Light Equation

Ambient lighting is the easiest lighting to model in our simple lighting model. Given an ambient light for the scene:

$$\mathcal{A} = \langle \mathcal{A}_r, \mathcal{A}_g, \mathcal{A}_b, \mathcal{A}_a \rangle$$

And how the object reacts with the ambient light (the diffuse reflection component of the material):

$$\mathcal{D} = \langle \mathcal{D}_r, \mathcal{D}_g, \mathcal{D}_b, \mathcal{D}_a \rangle$$

The equation for working out the colour of the surface gained from the ambient light interacting with it is:

$$\mathcal{DA} = \langle \mathcal{D}_r \mathcal{A}_r, \mathcal{D}_g \mathcal{A}_g, \mathcal{D}_b \mathcal{A}_b, \mathcal{D}_a \mathcal{A}_a \rangle$$

That is, we just multiply each component of the light value by the equivalent component of the material value. For example, consider the following light and material values:

$$\begin{aligned}\mathcal{A} &= \langle 0.97, 1.0, 0.4, 1.0 \rangle \\ \mathcal{D} &= \langle 0.2, 0.2, 0.2, 1.0 \rangle\end{aligned}$$

Then the ambient colour of the object will be:

$$\mathcal{DA} = \langle 0.194, 0.2, 0.08, 1.0 \rangle$$

Ambient light is the easiest to deal with in our simplistic lighting model, but ambient light itself is more complex than we have discussed here.

52.3 Ambient Lighting Shader

We are going to build a shader now that performs the ambient lighting equation. As the equation is simple, the algorithm we will use to calculate the ambient colour value will also be simple.

The ambient shader works using the following pipeline:

$$\textit{position} \Rightarrow \textit{vertex_colour} \Rightarrow \textit{colour}$$

Figure 52.1 provides the structure for this shader.

52.4 Exercise

This lesson requires you to modify the two shader files - `simple_ambient.vert` and `simple_ambient.frag` - to calculate the ambient lighting colour to apply to the objects. You will also need to update the main method to set the `ambient_intensity` and `material_colour` values of the effect. When you run this application you will get the output shown in Figure 52.2 (you can also change the camera position using 1-4).

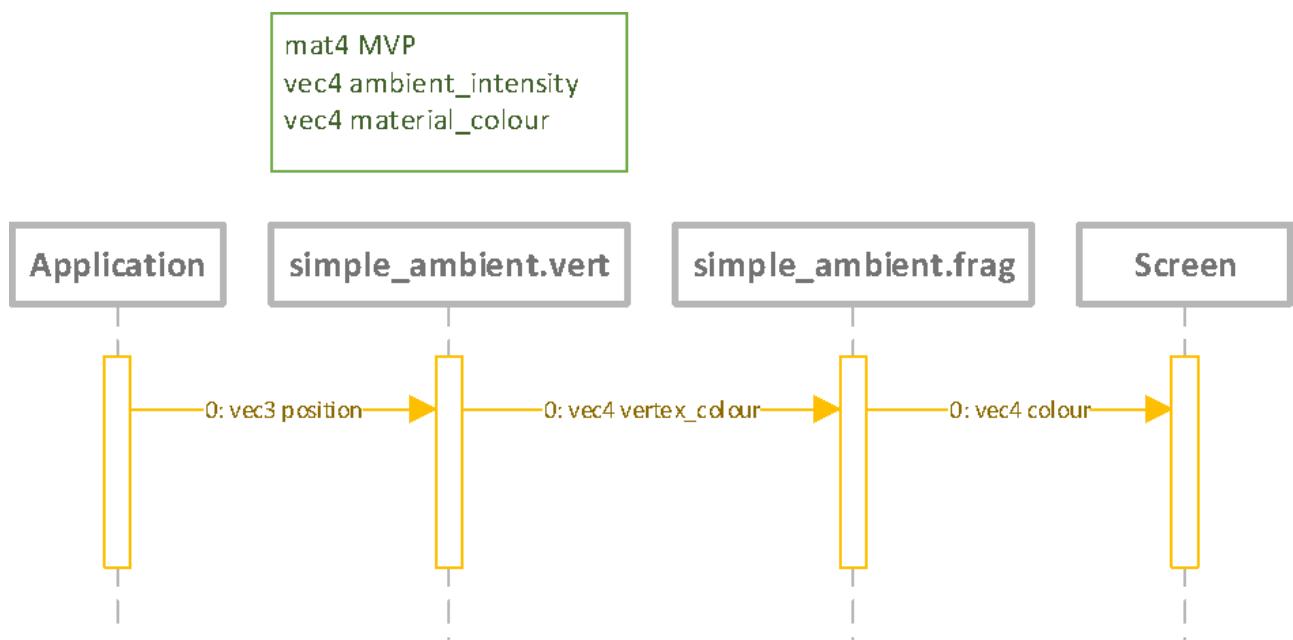


Figure 52.1: Ambient Shader Structure

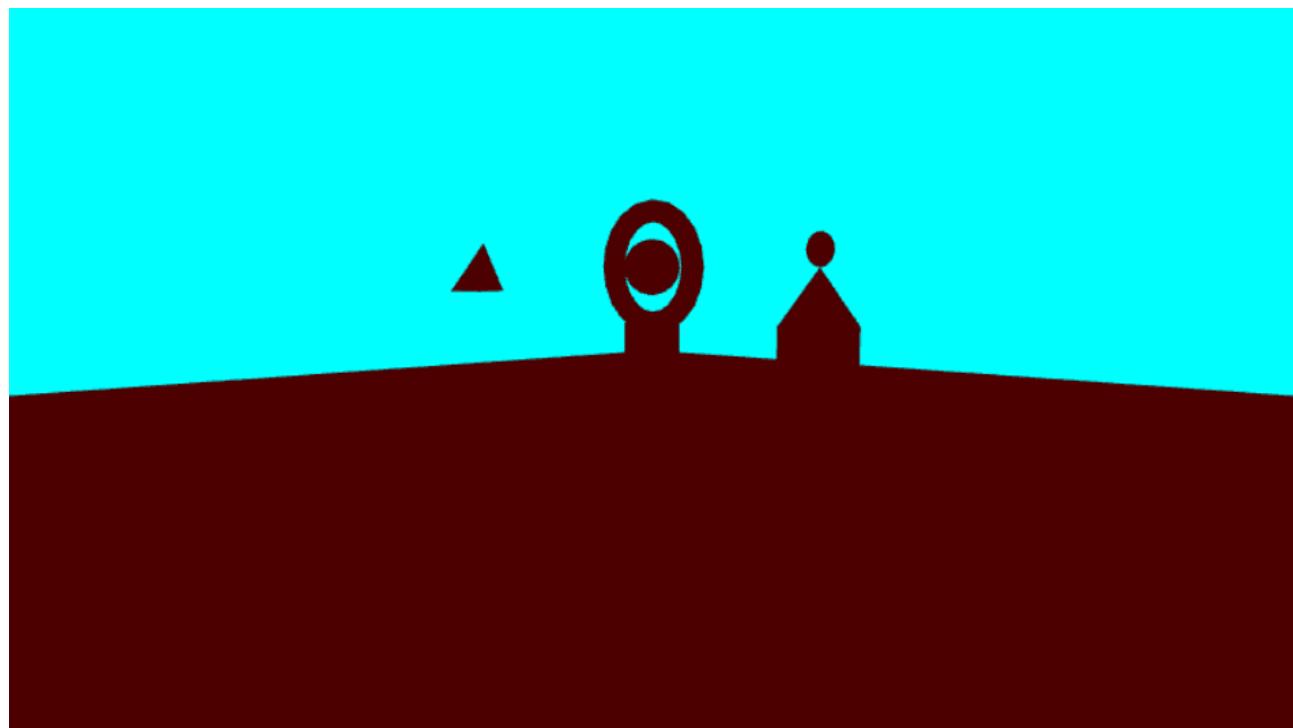


Figure 52.2: Output from Ambient Lighting Lesson

Lesson 53

Diffuse Lighting

Diffuse light is a far more interesting light type than ambient, and when we actually implement diffuse light in a couple of lessons time, you will see actual 3D effects on our objects (in other words, they will look like they have depth, etc.). Diffuse light is a little bit more complex to deal with than ambient light as we must take into account the direction of the light, and how it interacts with the surface of the object.

53.1 Examples

Diffuse light, from our point of view, is what gives an object a more realistic shape. Essentially, the adding of diffuse light to our render allows us to perceive the 3D look of an object without resorting to such approaches as colouring each side individually. With diffuse light, we can set a single colour for the object, and allow the light affecting the object to produce a 3D look. For example, compare the ambient effect (left) with the diffuse (right) in Figure 53.1.

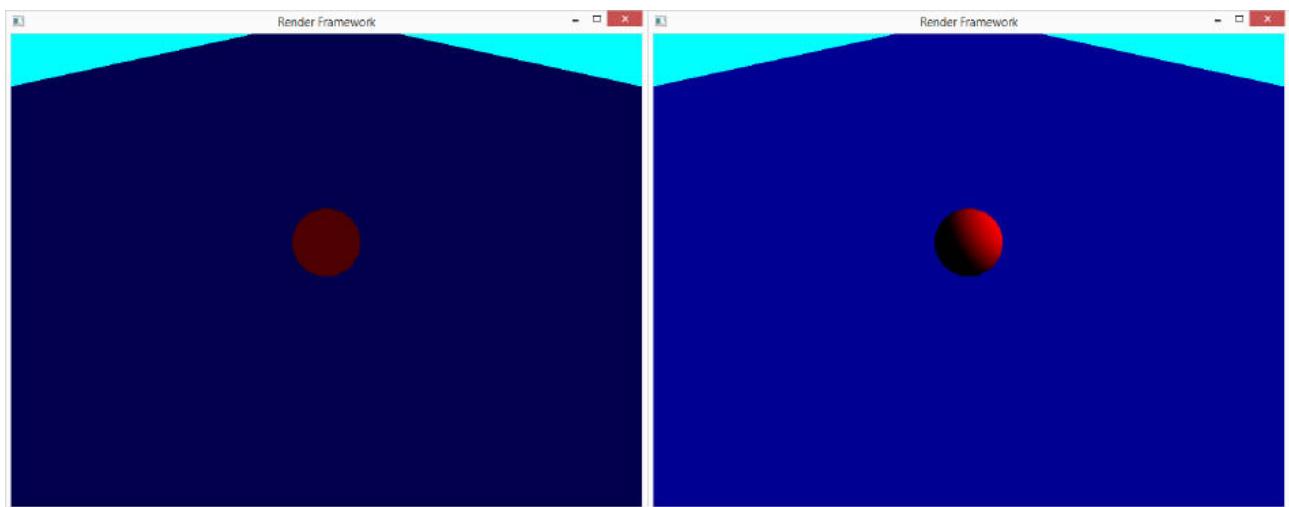


Figure 53.1: Ambient Diffuse Comparison

Notice that we have definite shape of the object when working with diffuse light, but not any when using just ambient light.

53.2 Diffuse Lighting Equation

The diffuse lighting equation is slightly more complex than the ambient light equation. As before, we need to define the light (\mathcal{C}) and diffuse reflection material (\mathcal{D}) values:

$$\begin{aligned}\mathcal{C} &= \langle C_r, C_g, C_b, C_a \rangle \\ \mathcal{D} &= \langle D_r, D_g, D_b, D_a \rangle\end{aligned}$$

We also need to define the direction that the light is travelling in. This must be a normalized vector (i.e. unit length), and is just the direction to the light source:

$$\mathbf{L} = \langle L_x, L_y, L_z \rangle$$

We also need the surface normal for the triangle we are working on. This is set on a per-vertex basis, as you will soon see. However, at the moment, just consider the normal to be a 3-dimensional vector of unit length.

$$\mathbf{N} = \langle N_x, N_y, N_z \rangle$$

The diffuse light calculation works on the principle that the cosine of the angle between two vectors ranges from 1.0 (vectors facing the same direction) to 0.0 (vectors are perpendicular or at $\frac{\pi}{2}$ radians). We can consider this to be 100% to 0% lit. We already know the relationship between the cosine of the angle between two vectors and the dot product of the two vectors. Therefore, we just use the fast dot product operation to calculate our light. The diffuse light equation is:

$$Diffuse = (\mathcal{D}\mathcal{C}) \max(\mathbf{N} \cdot \mathbf{L}, 0.0)$$

For example, let us consider the following values:

$$\begin{aligned}\mathcal{C} &= \langle 0.97, 1.0, 0.4, 1.0 \rangle \\ \mathcal{D} &= \langle 0.2, 0.2, 0.2, 1.0 \rangle\end{aligned}$$

This gives us the same calculated colour as before:

$$Diffuse = \langle 0.194, 0.2, 0.08, 1.0 \rangle$$

Let us also consider the direction to the light to be as follows (that is, it is up to the right from the object):

$$\mathbf{L} = \langle \frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}, 0 \rangle$$

Now let us calculate the diffuse light for the six sides of a cube. The normals are therefore:

$$\begin{aligned}
\mathbf{N}_1 &= \langle 1, 0, 0 \rangle \\
\mathbf{N}_2 &= \langle -1, 0, 0 \rangle \\
\mathbf{N}_3 &= \langle 0, 1, 0 \rangle \\
\mathbf{N}_4 &= \langle 0, -1, 0 \rangle \\
\mathbf{N}_5 &= \langle 0, 0, 1 \rangle \\
\mathbf{N}_6 &= \langle 0, 0, -1 \rangle
\end{aligned}$$

The dot products of each of these six normal with the light direction is as follows:

$$\begin{aligned}
\mathbf{L} \cdot \mathbf{N}_1 &= \frac{1}{\sqrt{2}} \\
\mathbf{L} \cdot \mathbf{N}_2 &= -\frac{1}{\sqrt{2}} \\
\mathbf{L} \cdot \mathbf{N}_3 &= \frac{1}{\sqrt{2}} \\
\mathbf{L} \cdot \mathbf{N}_4 &= -\frac{1}{\sqrt{2}} \\
\mathbf{L} \cdot \mathbf{N}_5 &= 0 \\
\mathbf{L} \cdot \mathbf{N}_6 &= 0
\end{aligned}$$

Our equation uses the max function to ensure that the minimum value we use is zero. This means we are performing the following six calculations:

$$\begin{aligned}
&\langle 0.194, 0.2, 0.08, 1.0 \rangle \times \frac{1}{\sqrt{2}} = \langle 0.137, 0.141, 0.057, 0.707 \rangle \\
&\langle 0.194, 0.2, 0.08, 1.0 \rangle \times 0 = \langle 0, 0, 0, 0 \rangle \\
&\langle 0.194, 0.2, 0.08, 1.0 \rangle \times \frac{1}{\sqrt{2}} = \langle 0.137, 0.141, 0.057, 0.707 \rangle \\
&\langle 0.194, 0.2, 0.08, 1.0 \rangle \times 0 = \langle 0, 0, 0, 0 \rangle \\
&\langle 0.194, 0.2, 0.08, 1.0 \rangle \times 0 = \langle 0, 0, 0, 0 \rangle \\
&\langle 0.194, 0.2, 0.08, 1.0 \rangle \times 0 = \langle 0, 0, 0, 0 \rangle
\end{aligned}$$

So only two faces (the faces pointing up and pointing left) are lit. In all instances, we will actually set the alpha component of the colour to 1 so that the objects rendered are opaque.

Diffuse light is actually quite easy to deal with. Really all we need to deal with is the calculation of the surface normals, and we do this during our geometry building at application start up. Let us look at normals next.

Lesson 54

Surface Normals

We have mentioned a few times now the concept of surface normals. A surface normal simply lets us define which direction a surface is facing. There are actually simple calculations for this, but generally speaking we want to define the surface normal at the start of the application so we do not have to calculate this value every frame.

54.1 What is a Surface Normal?

Figure 54.1 illustrates what we mean by a surface normal.

It is a vector that is perpendicular to the surface we are working on. The normal also has unit length (i.e. it has a length of 1). This allows us to use it in lighting calculations easily.

54.2 Calculating Surface Normals

Surface normal calculation is actually very simple –we need to calculate the cross product between the two defining vectors. For example, for Figure 54.1 the surface normal can be defined as:

$$n = (V1 - V0) \times (V2 - V0)$$

The render framework takes care of the surface normals for the basic geometric shapes provided, and most models you can load also come with surface normals. You might need to calculate them yourself sometimes, but it isn't that likely.

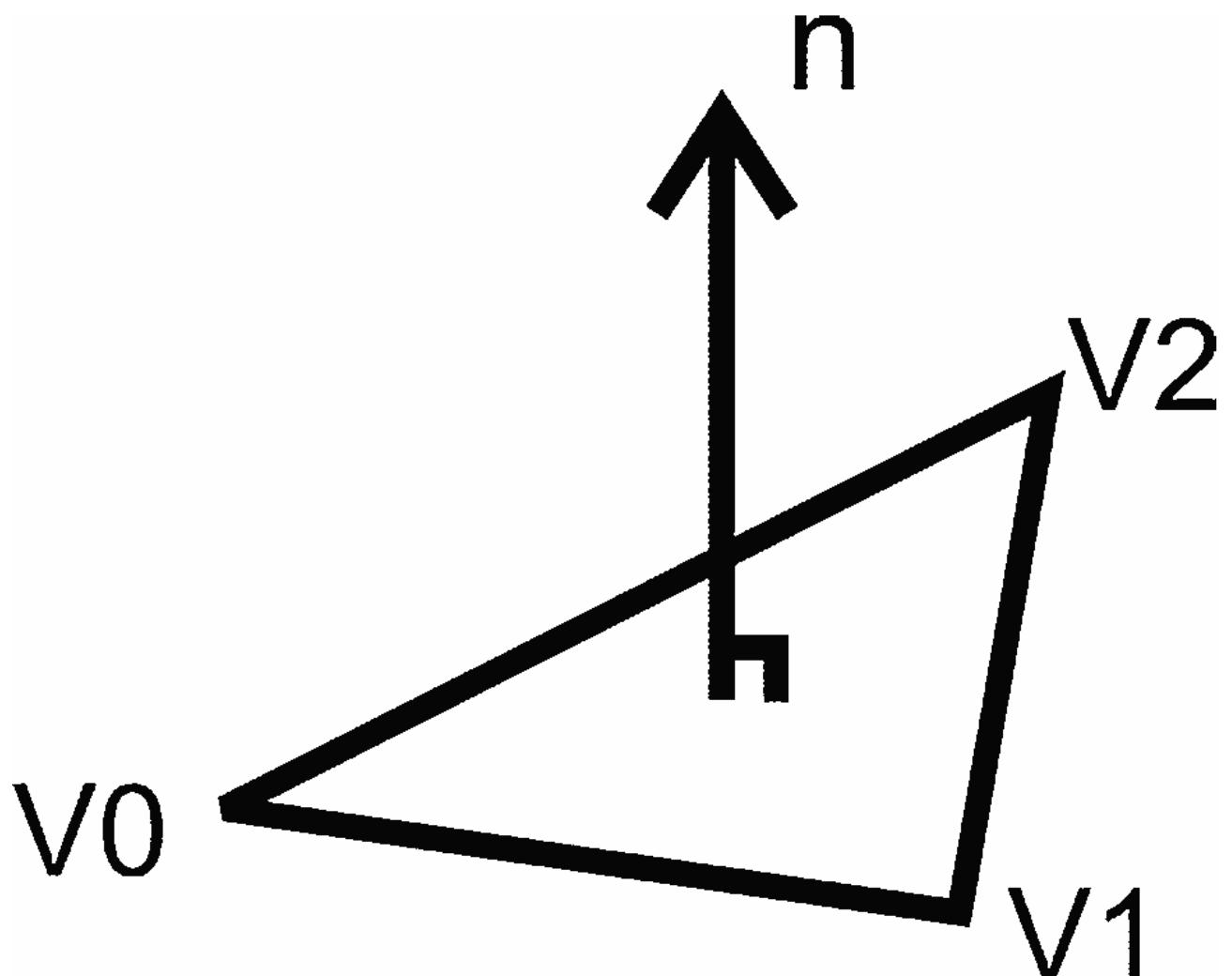


Figure 54.1: Surface Normal

Lesson 55

Diffuse Lighting Attempt 1

We are ready to create our diffuse light shader. As before, we have the vertex and the fragment shader to implement. The vertex shader is where all the work is being done, just as in our ambient shader.

The pipeline for our shader this time is as follows:

$$\text{position, normal} \Rightarrow \text{vertex_colour} \Rightarrow \text{colour}$$

The structure of the shader is given in Figure 55.1.

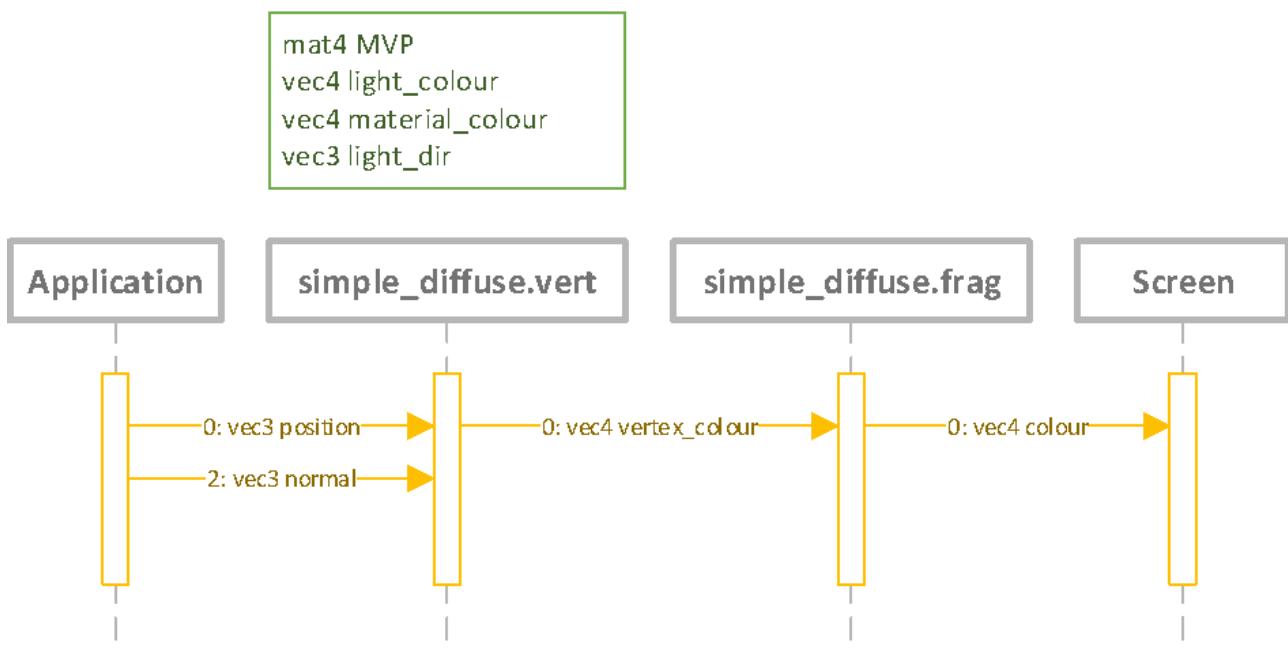


Figure 55.1: Simple Diffuse Shader Structure

Your goal in this lesson is to implement the diffuse shader and set the relevant uniforms in the main application. For the vertex shader, we will split the equation up as follows:

$$k = \max(\text{normal} \cdot \text{light_dir}, 0)$$

$$\text{diffuse} = k \cdot (\text{material_colour} * \text{light_colour})$$

Figure 55.2 illustrates the output from this lesson. Note that the sphere is rotating, but the light seems to rotate with the light. We will fix that in the next lesson.

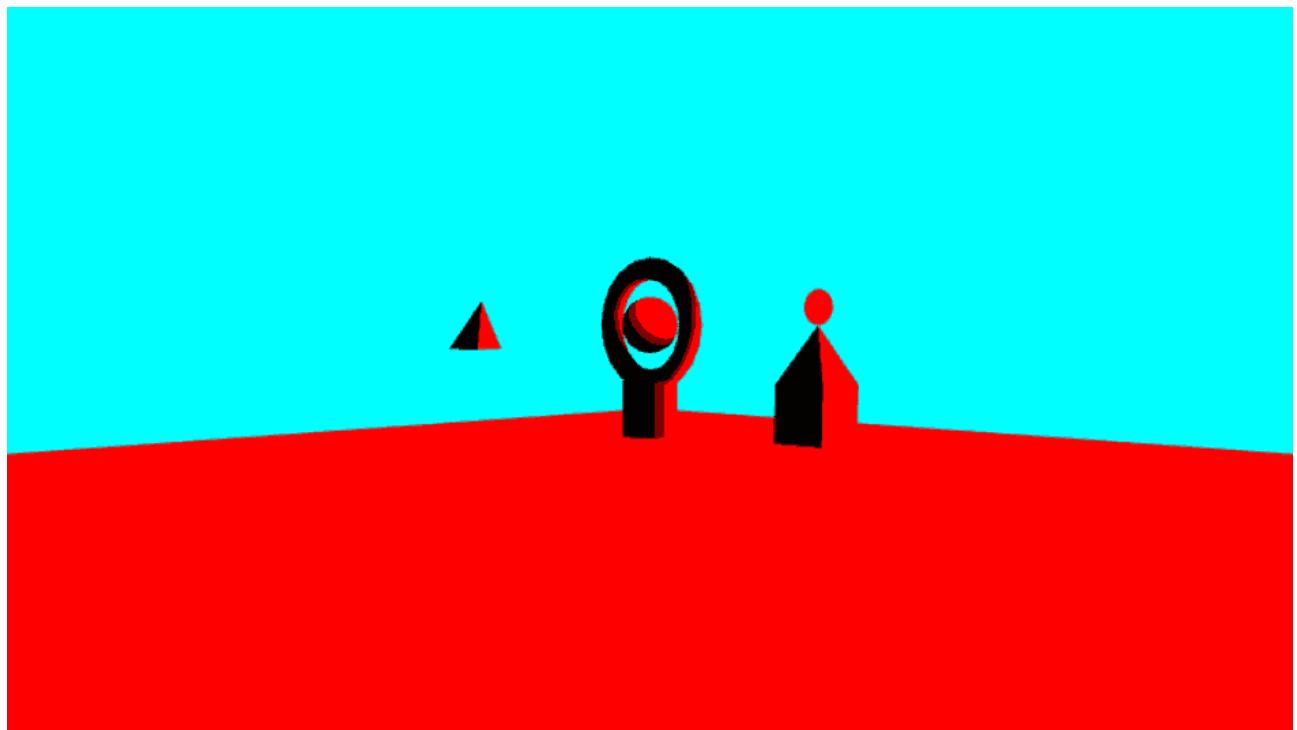


Figure 55.2: **Output from First Attempt at a Diffuse Shader**

Lesson 56

Diffuse Lighting Attempt 2

Now we need to fix what went wrong in our diffuse shader. The problem actually comes from the fact that we are not transforming our normals to match the fact that we have transformed our geometry. This is not a case of just applying the same transformation matrix (model-view-projection) as before, but using a new matrix. The matrix we have to use is called normal-matrix, which just needs to take into account the rotation of the object (if we implemented other transformations such as sheering we would have to work a bit harder). Luckily the render framework can get this for you.

56.1 Working Diffuse Shader

The new structure for our diffuse shader is shown in Figure 56.1. The only difference to our previous diffuse shader (Figure 55.1) is the addition of a new uniform \mathbf{N} . This is our normal transformation matrix.

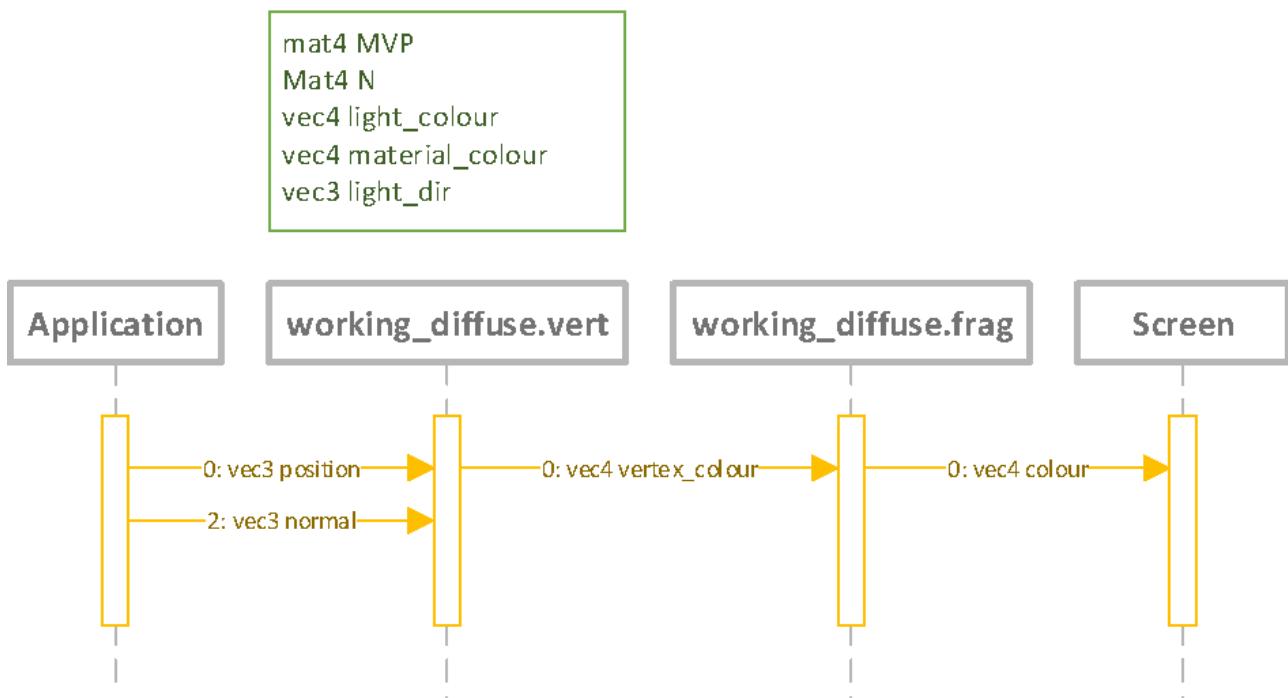


Figure 56.1: Working Diffuse Shader Structure

All we need to do to change our diffuse shader is to calculate a *transformed_normal* value to use in the lighting calculation. We do this by multiplying our **normal** value by our **N** value. The following equation gives the general idea.

$$\text{transformed_normal} = N \times \text{normal}$$

$$k = \max(\text{transformed_normal} \cdot \text{light_dir}, 0)$$

$$\text{diffuse} = k \cdot (\text{material_colour} * \text{light_colour})$$

The output from this shader is now correct, as shown in Figure 56.2.

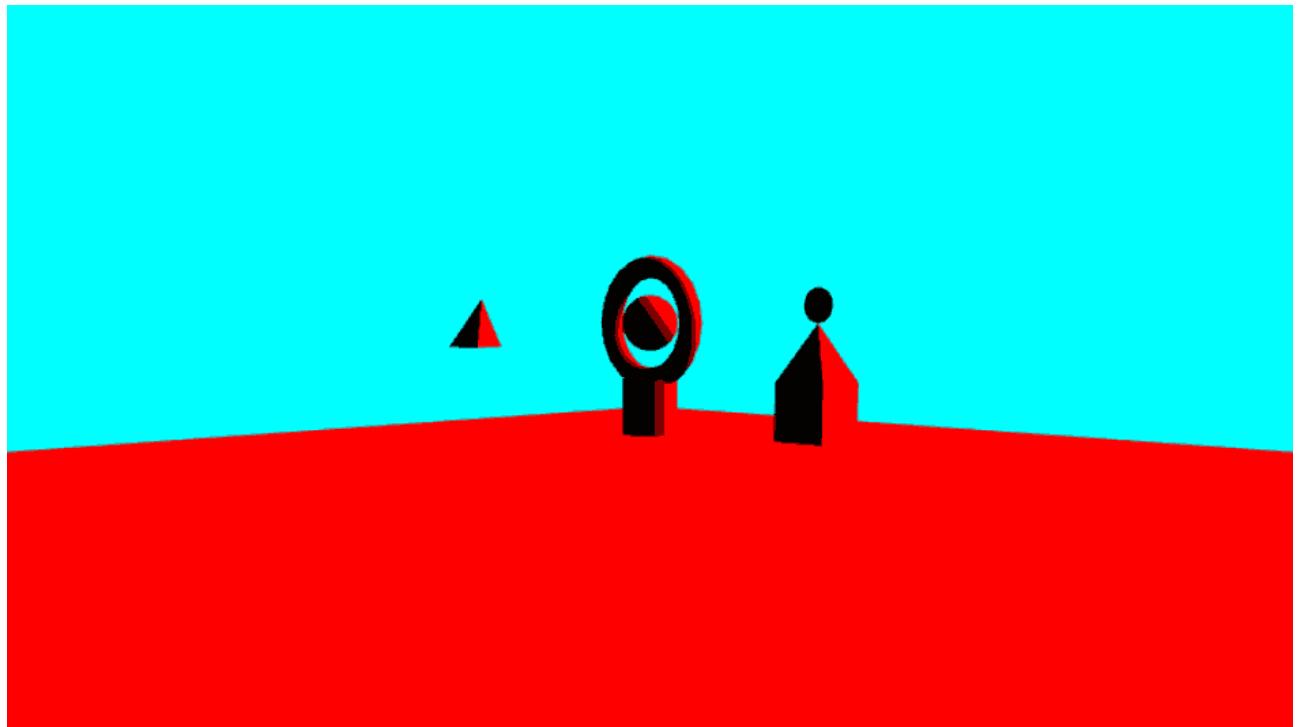


Figure 56.2: Output from Working Diffuse Shader

Lesson 57

Specular Lighting

Our final form of basic lighting is specular lighting. Specular lighting follows many of the characteristics of diffuse lighting it is based on the direction that the light is travelling in. However, unlike diffuse light, we are also interested in the position of the viewer. This is because we are creating an effect that provides highlights to an object. For example examine Figure 57.1.

You will see a couple of points on the rendered spheres where it looks like the light is reflecting from the surface of them. This is what is known as a specular highlight. It is determined by taking the direction of the light, the position of the eye, and using the surface normal to determine where the highlight appears on the object. Figure 57.2.

57.1 Specular Lighting Equation

The specular lighting equation is again more complex than the diffuse equation, although it does build upon it. As before, we need to define the light and material values. This time we still use the light colour (\mathcal{C}) but we use a different specular reflection value \mathcal{S} . We also use a shininess value m .

$$\begin{aligned}\mathcal{C} &= \langle \mathcal{C}_r, \mathcal{C}_g, \mathcal{C}_b, \mathcal{C}_a \rangle \\ \mathcal{S} &= \langle \mathcal{S}_r, \mathcal{S}_g, \mathcal{S}_b, \mathcal{S}_a \rangle\end{aligned}$$

We need to define the direction that the light is travelling in:

$$\mathbf{L} = \langle \mathbf{L}_x, \mathbf{L}_y, \mathbf{L}_z \rangle$$

We also need the eye position. This is just a 3-dimensional position in space - the same position as our camera. It is defined as follows:

$$e = \langle e_x, e_y, e_z \rangle$$

Unlike the diffuse shader, we will work with the world position of the vertex as well:

$$p = \langle p_x, p_y, p_z \rangle$$

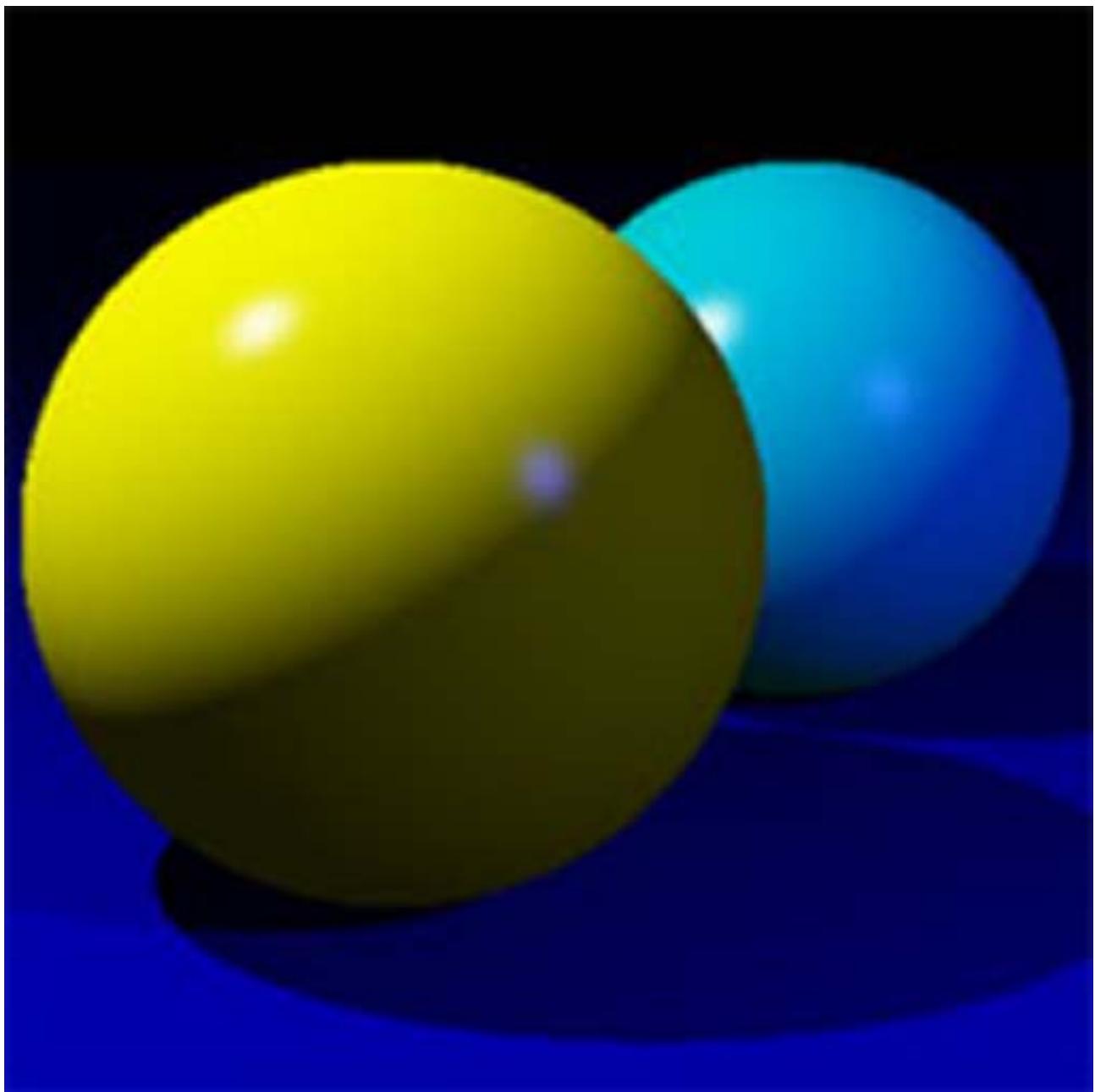


Figure 57.1: Specular Highlights

To calculate the world position, we have to transform the model position (the data that comes into the shader) into world space. We have to use the model matrix to do this. A more accurate calculation of the position is therefore:

$$p = M \times \langle p'_x, p'_y, p'_z, 1 \rangle$$

As diffuse, the surface normal for the triangle we are working on is required:

$$\mathbf{N} = \langle \mathbf{N}_x, \mathbf{N}_y, \mathbf{N}_z \rangle$$

One final value we need is the shininess of the object. This is just a scalar value m .

The specular lighting calculation requires us to calculate the half vector \mathbf{H} between the light direction and the direction to the eye. First we need to calculate the eye direction (a normalized vector):

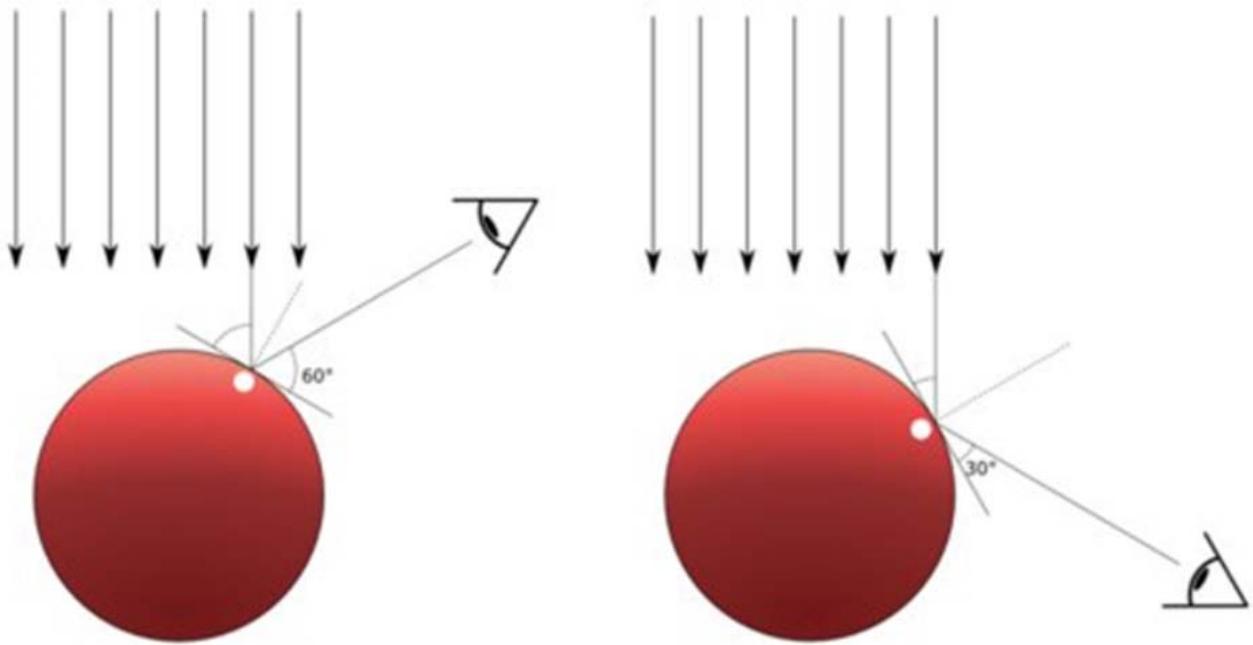


Figure 57.2: Eye-Light Interaction

$$\mathbf{E} = \frac{\mathbf{e} - \mathbf{p}}{\|\mathbf{e} - \mathbf{p}\|}$$

We then calculate \mathbf{H} as follows:

$$\mathbf{H} = \frac{\mathbf{L} + \mathbf{E}}{\|\mathbf{L} + \mathbf{E}\|}$$

We can calculate the specular value based on the angle between the half vector \mathbf{H} and the surface normal \mathbf{N} just as we did for the diffuse value:

$$\text{specular} = (\mathcal{SD}) \max(\mathbf{N} \cdot \mathbf{H}, 0.0)^m$$

We will not calculate an example here as there is too much data for this page.

57.2 Specular Shader

As before we need to have a vertex shader and a fragment shader. Also, as before, the fragment shader is the same as the diffuse and ambient versions. For the vertex shader you will have to calculate the specular intensity value. This was the second half of the equation above:

$$s = \max(\mathbf{N} \cdot \mathbf{H}, 0.0)^m$$

The pipeline through the shader is the same as the diffuse shader:

position, normal \Rightarrow *vertex_colour* \Rightarrow *colour*

The structure of the shader is roughly the same, but with the addition of extra uniforms. Figure 57.4 presents the structure of the specular shader.

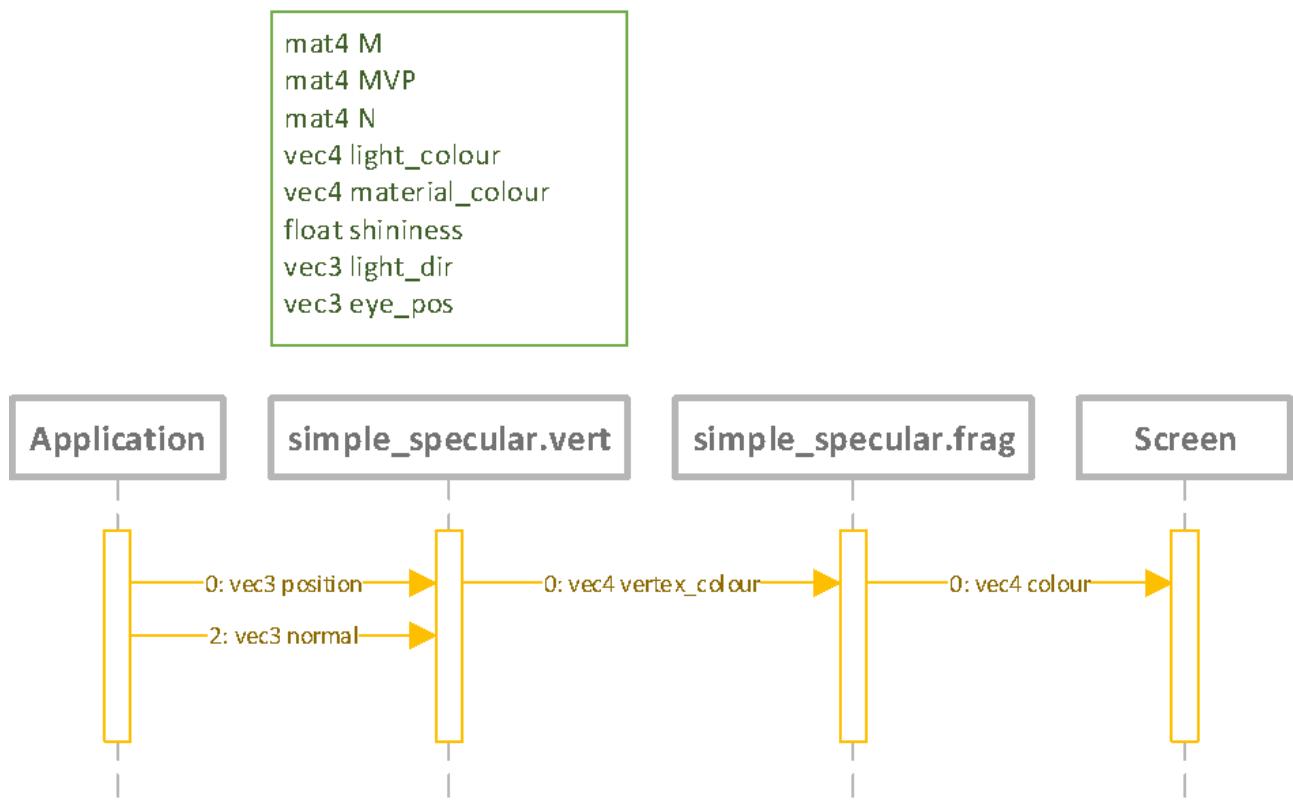


Figure 57.3: Specular Shader Structure

57.3 Exercise

For this lesson you have to implement the specular shader. You have to perform the same tasks as the previous lighting lessons - complete the shaders and set the uniforms in the main application. Figure 57.5 provides an example output from this lesson.

Some other exercises for you to work on:

1. Try manipulating the shininess value and see what results you achieve. Try and determine the relationship between the shininess value, and what you see on the screen.
2. Try different types of geometry with the specular shader. Can you see the relationship between the eye position and where the reflection appears?
3. Try manipulating the light direction, and further understand how the light direction and eye position effects the specular light you see.

```
mat4 M
mat4 MVP
mat4 N
vec4 light_colour
vec4 material_colour
float shininess
vec3 light_dir
vec3 eye_pos
```

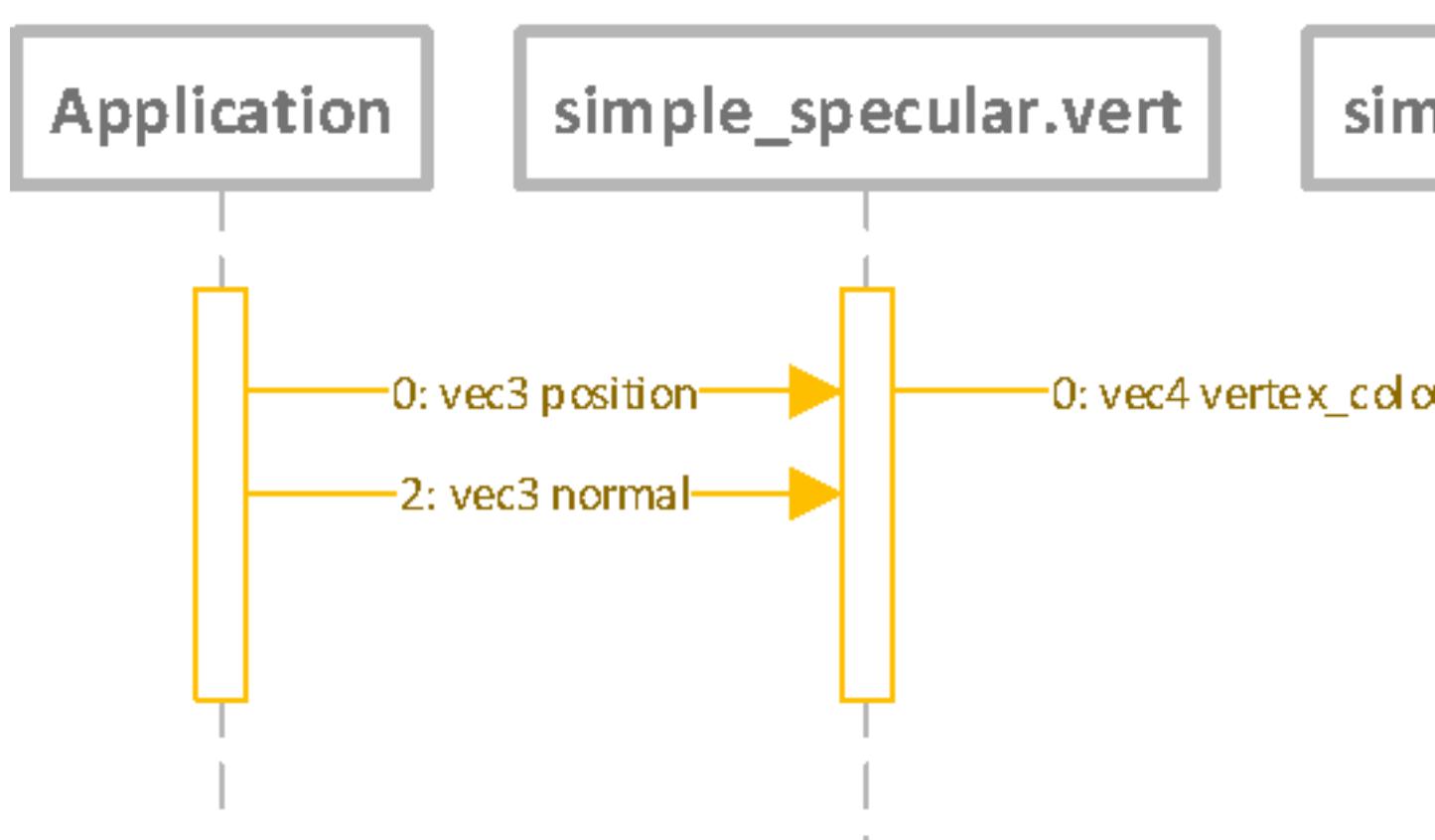


Figure 57.4: Specular Shader Structure

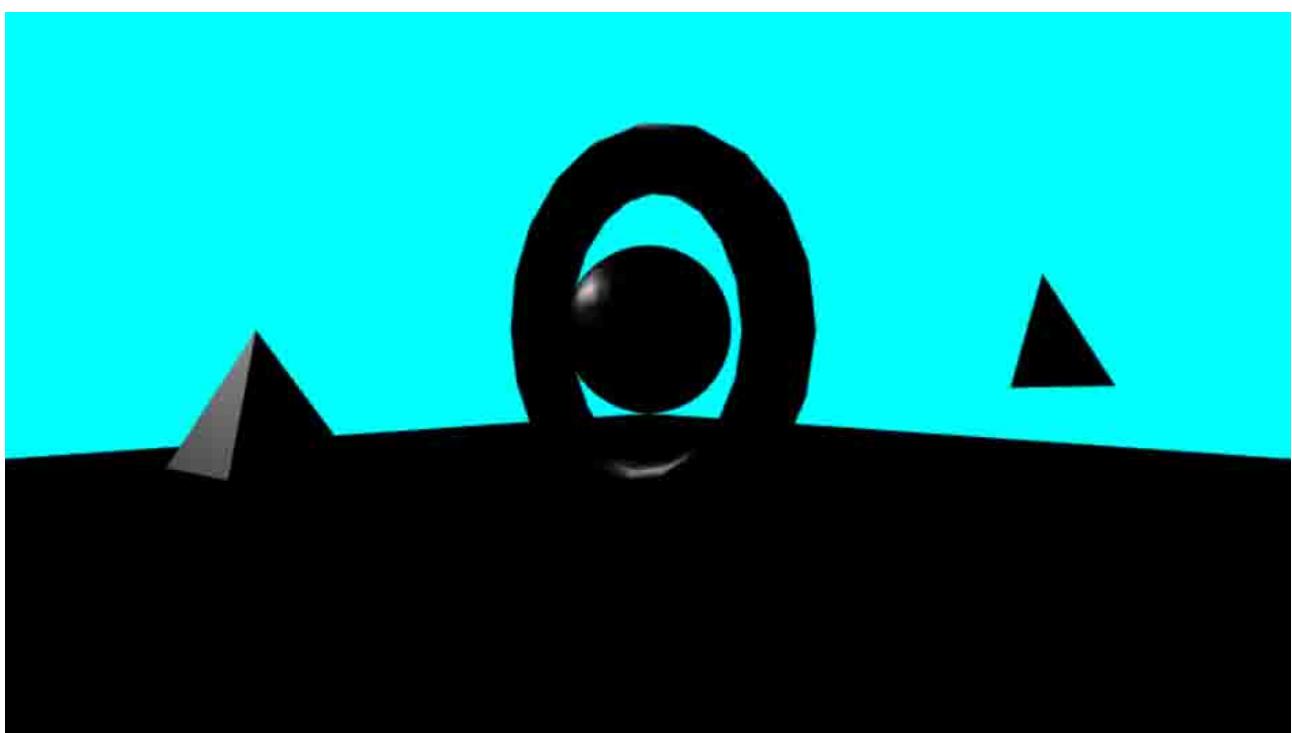


Figure 57.5: Output from Specular Shader

Lesson 58

Combined Lighting

OK, we have made three separate shaders to illustrate the three different lighting types. Let us now move onto combining the three lighting effects. That is, we are going to calculate the following equation (where \mathcal{K} is the final colour reaching the eye):

$$\mathcal{K} = \mathcal{DA} + \mathcal{DC} \max(\mathbf{L} \cdot \mathbf{N}, 0.0) + \mathcal{SC} \max(\mathbf{N} \cdot \mathbf{H}, 0.0)^m$$

This might look complicated, but all we are really doing is adding the three lighting components together.

Your task in this lesson is to complete the necessary shaders and update the main application. Our pipeline through the shader remains the same:

$$position, normal \Rightarrow vertex_colour \Rightarrow colour$$

Our structure combines the three previous lighting shader into one (with renaming accordingly). Figure 58.1 illustrates.

Just to help you along, the complete algorithm for combined lighting is shown in Algorithm 18.

Algorithm 18 Combined Lighting Algorithm

```
1: procedure COMBINED_LIGHTING( $M$ ,  $MVP$ ,  $N$ , ambient_intensity, light_colour,  
    light_dir, diffuse_reflection, specular_reflection, shininess, eye_pos, position, normal)  
2:    $gl\_Position \leftarrow MVP \times position$   
3:    $ambient \leftarrow diffuse\_reflection * ambient\_intensity$   
4:    $transformed\_normal \leftarrow N \times normal$   
5:    $diffuse \leftarrow \max(transformed\_normal \cdot light\_dir, 0) \cdot (diffuse\_reflection * light\_colour)$   
6:    $world\_position \leftarrow M \times position$   
7:    $view\_dir \leftarrow eye\_pos - world\_position$   
8:    $half\_vector \leftarrow light\_dir + view\_dir$   
9:    $specular \leftarrow \max(transformed\_normal \cdot half\_vector, 0)^{shininess} \cdot (specular\_reflection * light\_colour)$   
10:   $vertex\_colour \leftarrow ambient + diffuse + specular$ 
```

The output is shown in Figure 58.2.

```
mat4 M
mat4 MVP
mat4 N
vec4 ambient_intensity
vec4 light_colour
vec3 light_dir
vec4 diffuse_reflection
vec4 specular_reflection
float shininess
vec3 eye_pos
```

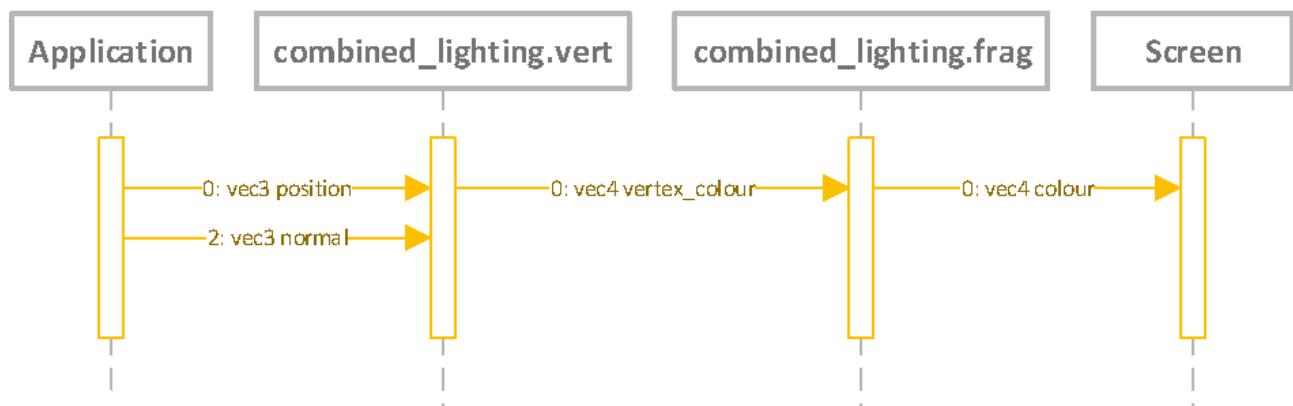


Figure 58.1: Structure of Combined Lighting Shader

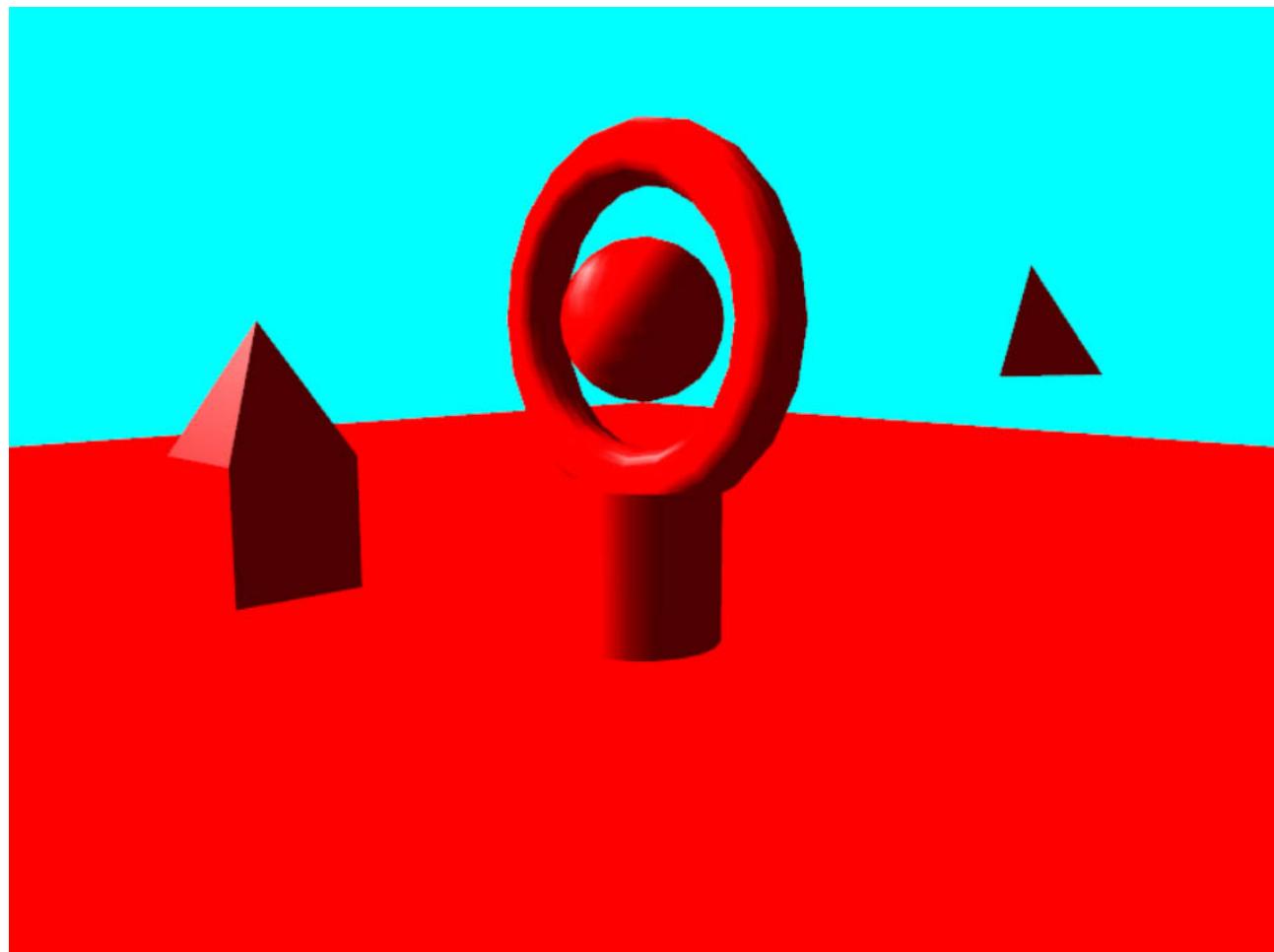


Figure 58.2: Output from Combined Lighting Lesson

Lesson 59

Emissive Lighting

So far, we have been treating objects simply as light absorbers and reflectors in our scene. However, objects can also emit light as well. Let us add one other component to our scene –using emissive light.

59.1 What is Emissive Light?

Emissive light is light that emits from a rendered object in our scene. Figure 59.1 provides an example of emissive light.

However, we will not be dealing with emissive light quite in this manner. We will consider emissive material to be the colour that a surface emits, without taking into account the different lighting values. If we also want to emit a light from the object, we can use a point light for this - more on this in later lessons.

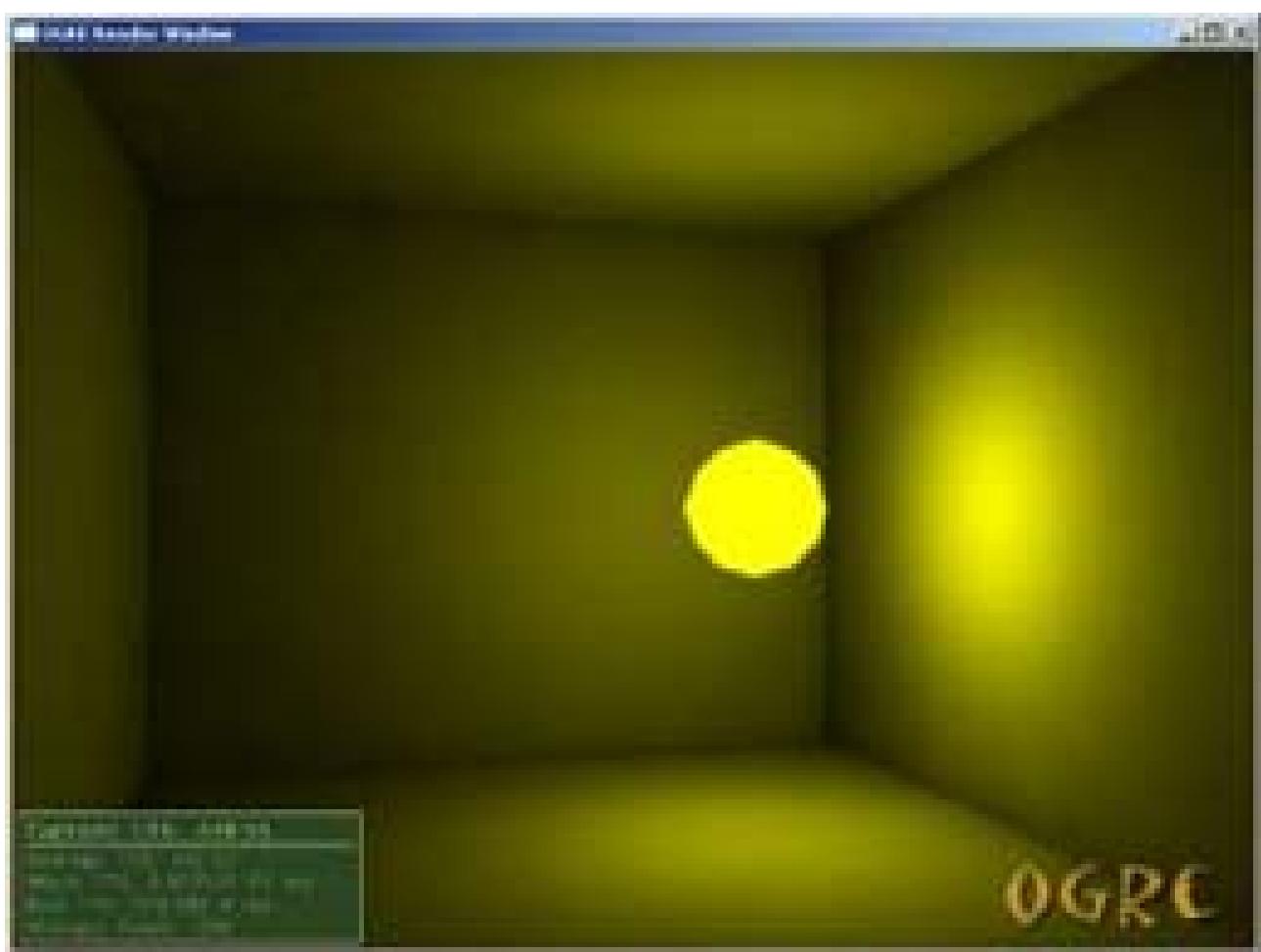


Figure 59.1: **Emissive Light**

Lesson 60

Gouraud Shading

By introducing emissive light, and also taking into account texturing, we are able to utilise one of the main shading and lighting models - Gouraud shading. In this model, we take our current understanding of lighting and combine them together with emission and texturing to create a realistic looking scene.

Gouraud shading uses the following equations to calculate a colour:

$$\begin{aligned}\mathcal{K}_{primary} &= \mathcal{E} + \mathcal{D}\mathcal{A} + \mathcal{D}\mathcal{C} \max(\mathbf{L} \cdot \mathbf{N}, 0.0) \\ \mathcal{K}_{secondary} &= \mathcal{S}\mathcal{C} \max(\mathbf{N} \cdot \mathbf{H}, 0.0)^m \\ \mathcal{K} &= \mathcal{K}_{primary} \mathcal{T}_1 \mathcal{T}_2 \dots + \mathcal{K}_{secondary}\end{aligned}$$

In other words we take the emissive-ambient-diffuse component of the lighting and multiply it by and texture samples (the \mathcal{T} values) and then add the specular colour. This means our vertex shader now calculates two values - $\mathcal{K}_{primary}$ and $\mathcal{K}_{secondary}$ - and sends them to the fragment shader for calculating the final colour based on texture sampling. Our pipeline now becomes the following:

```
position      primary_colour  
normal    ⇒ secondary_colour ⇒ colour  
tex_coord      tex_coord
```

Before moving onto implementing the Gouraud shader let us look at how we can simplify our shader code by creating user defined values using **structs**.

60.1 structs in Shaders

We have now covered quite a number of basics in doing lighting and shading, and you may have noticed that we have quite a bit of information that we can pass to a shader to get the lighting that we want. However, this does mean that we have quite a bit of code in our main operation (which is not the best situation). To rectify this, we are going to work with some useful data structures to simplify our work.

GLSL is a C based language. Because of this we can use certain C function - most importantly **struct**. What we are going to do is create a **struct** to represent our reusable parts of rendering information - that is the material and the light. We will create two **structs** to support this.

For those of you unfamiliar with a **struct** this is very similar to a class in standard object-orientation. From a C++ point of view, a **struct** is identical to a **class** except that the default protection level of attributes is **public** rather than **private**. Typically however we consider a **struct** as a combination of values to create a new data type (like a record), with the possibility of some methods defined. We don't normally use inheritance and other object-oriented concepts. For example, if we want to declare a new data structure to represent a student we could declare the following.

```
1 struct student
2 {
3     unsigned int matric_number;
4     string name;
5     string programme;
6 };
```

We can then declare variable of this type in our code:

```
1 student a;
```

We can get and set values of the struct by using dot notation:

```
1 string s = a.name;
2 a.programme = "BSc Games Development";
```

Within GLSL, the same capabilities exist. We can declare new data types and then get and set values using the dot notation. This is also true for **uniform**, **in** and **out** values (except you cannot set values in a **uniform**). We will start using **struct** values from now. This is especially useful for some of the values we have started using.

60.1.1 Declaring structs in Shaders

We are going to declare two **structs** in our Gouraud vertex shader - **directional_light** and **material**. These new data types will contain the information required to describe the directional light of a scene and the material properties of an object.

If we consider these objects in turn, our directional light has the following information:

- ambient intensity
- light colour
- light direction

Our material description has the following values:

- emissive colour
- diffuse reflection colour
- specular reflection colour

- shininess

Looking at these as values in our shader, we declare the following for a directional light:

```
1 // A directional light structure
2 struct directional_light
3 {
4     vec4 ambient_intensity;
5     vec4 light_colour;
6     vec3 light_dir;
7 };
```

The following is the definition of our material:

```
1 // A material structure
2 struct material
3 {
4     vec4 emissive;
5     vec4 diffuse_reflection;
6     vec4 specular_reflection;
7     float shininess;
8 };
```

All we need to do now is declare `uniform` values of these types to allow us to complete the shader:

```
1 // Directional light for the scene
2 uniform directional_light light;
3 // Material of the object
4 uniform material mat;
```

Our shader is then written using these data types rather than individual values. This takes care of our shader, but what about the main application? How do we set the uniform values now?

60.1.2 Setting Uniform Values in structs

From our OpenGL point of view, all that has happened in the shader is that we have renamed some of the `uniform` values. As OpenGL does not understand what these data structures are, it looks at the internal representation of the `struct` to see the uniforms created and expands these out. For example, although GLSL code has a declaration:

```
1 uniform directional_light light;
```

When compiled and loaded by OpenGL the following `uniform` values are visible:

```
1 uniform vec4 light.ambient_intensity;
2 uniform vec4 light.light_colour;
3 uniform vec3 light.light_dir;
```

If we had another `directional_light` `uniform` called `other` then we would also have the following `uniform` values declared:

```

1 uniform vec4 other.ambient_intensity;
2 uniform vec4 other.light_colour;
3 uniform vec3 other.light_dir;
```

This means that if we want to set our uniform values using the render framework we would do the following:

```

1 glUniform4fv(eff.get_uniform_location("light.ambient_intensity"), 1, ←
    value_ptr(ambient_intensity));
2 glUniform4fv(eff.get_uniform_location("light.light_colour"), 1, ←
    value_ptr(light_colour));
3 glUniform3fv(eff.get_uniform_location("light.light_dir"), 1, ←
    value_ptr(light_dir));
```

Simple! However, this would start getting tedious after some time (particularly when we introduce other data types later). Therefore, the render framework provides a helper method that takes care of this for you:

```
1 renderer::bind(light, "light");
```

We just use the name *light* as the uniform name and the graphics framework does the rest. The *light* value here is of type `directional_light` which is part of the graphics framework. It is declared as follows:

```

1 /*
2 Object that describes a directional light
3 */
4 class directional_light
5 {
6     private:
7     // Ambient intensity of the light
8     glm::vec4 _ambient;
9     // The colour of the light
10    glm::vec4 _colour;
11    // The direction the light is facing
12    glm::vec3 _direction;
13    public:
14    // Creates a directional light with some default colour values
15    directional_light() :
16        _ambient(glm::vec4(0.7f, 0.7f, 0.7f, 1.0f)),
17        _colour(glm::vec4(0.9f, 0.9f, 0.9f, 1.0f)),
18        _direction(glm::vec3(1.0f, 0.0f, 0.0f))
19    {
20    }
21    // Creates a directional light with provided values
22    directional_light(const glm::vec4 &ambient_intensity,
23                      const glm::vec4 &light_colour,
24                      const glm::vec3 &direction) :
25        _ambient(ambient_intensity),
26        _colour(light_colour),
27        _direction(direction)
28    {
29    }
```

```

30 // Default copy constructor and assignment operator
31 directional_light(const directional_light &other) = default;
32 directional_light& operator=(const directional_light &rhs) = ←
33     default;
34     // Gets the ambient intensity of the light
35     glm::vec4 get_ambient_intensity() const { return _ambient; }
36     // Sets the ambient intensity of the light
37     void set_ambient_intensity(const glm::vec4 &value) { _ambient = ←
38         value; }
39     // Gets the colour of the light
40     glm::vec4 get_light_colour() const { return _colour; }
41     // Sets the colour of the light
42     void set_light_colour(const glm::vec4 &value) { _colour = value; }
43     // Gets the direction of the light
44     glm::vec3 get_direction() const { return _direction; }
45     // Sets the direction of the light
46     void set_direction(const glm::vec3 &value) { _direction = value; }
47     // Rotates the light using the given Euler angles
48     void rotate(const glm::vec3 &rotation);
49     // Rotates the light using the given quaternion
50     void rotate(const glm::quat &rotation);
51 };

```

As you can see it comes with some helper methods. You will find a variable of type `directional_light` at the start of this lesson. There is a similar data type declared for material, with a similar helper method to bind it to the renderer.

60.2 Gouraud Shader

Now that we have covered `struct` values in shaders we can move on to completing the Gouraud shader. The structure of the shader is given in Figure 60.1. Note the use of `material` and `directional_light` values.

As we are now producing two colour values to pass to the fragment shader, we will look at the necessary algorithms. The vertex shader for Gouraud shading is given in Algorithm 19. The fragment shader algorithm is provided in Algorithm 20.

We are now ready to complete this lesson.

60.3 Exercise

We need to build both the vertex shader and the fragment shader this time. We aren't doing anything different as such - it is how we use the data that is changing.

Figure 60.2 illustrates the expected output for the Gouraud shader. Notice that the material colour is overwriting the white of the checked texture, but that the black remains (think about the equation here). We have also got some realistic looking renders happening now that we are mixing lighting and texturing.

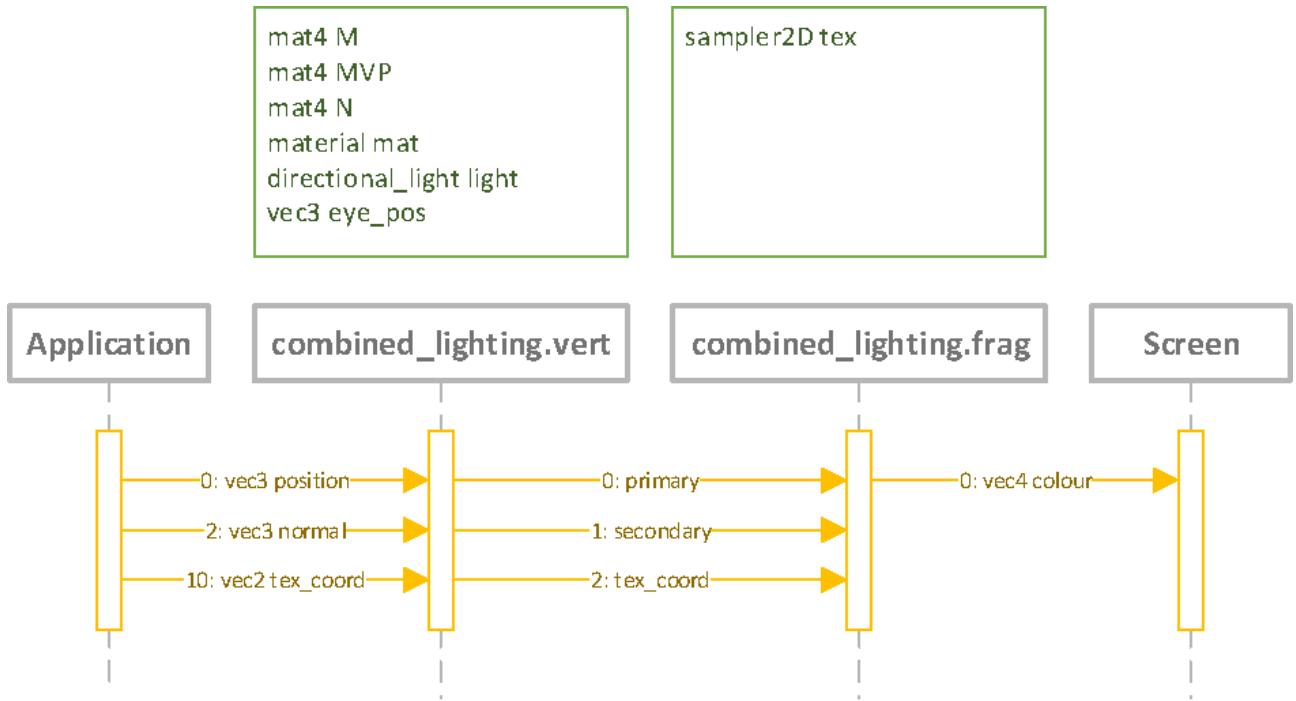


Figure 60.1: Gouraud Shader Structure

Algorithm 19 Gouraud Vertex Shader

```

1: procedure GOURAUD_VERTEX( $M, MVP, N, material, light, eye\_pos, position, normal$ )
2:    $ambient \leftarrow material.diffuse * light.ambient\_intensity$ 
3:    $transformed\_normal \leftarrow N \times normal$ 
4:    $diffuse \leftarrow \max(transformed\_normal \cdot light.light\_dir, 0) \cdot (material.diffuse * light.light\_colour)$ 
5:    $world\_position \leftarrow M \times position$ 
6:    $view\_dir \leftarrow (eye\_pos - world\_position)$ 
7:    $half\_vector \leftarrow (light.light\_dir + view\_dir)$ 
8:    $specular \leftarrow \max(transformed\_normal \cdot half\_vector, 0)^{material.shininess} \cdot (material.specular\_reflection * light.light\_colour)$ 
9:    $primary \leftarrow material.emissive + ambient + diffuse$ 
10:   $secondary \leftarrow specular$ 
  
```

Algorithm 20 Gouraud Fragment Shader

```

1: procedure GOURAUD_FRAGMENT( $tex, primary, secondary, tex\_coord$ )
2:    $tex\_colour \leftarrow TEXTURE(tex, tex\_coord)$ 
3:    $colour \leftarrow primary * tex\_colour + secondary$ 
  
```

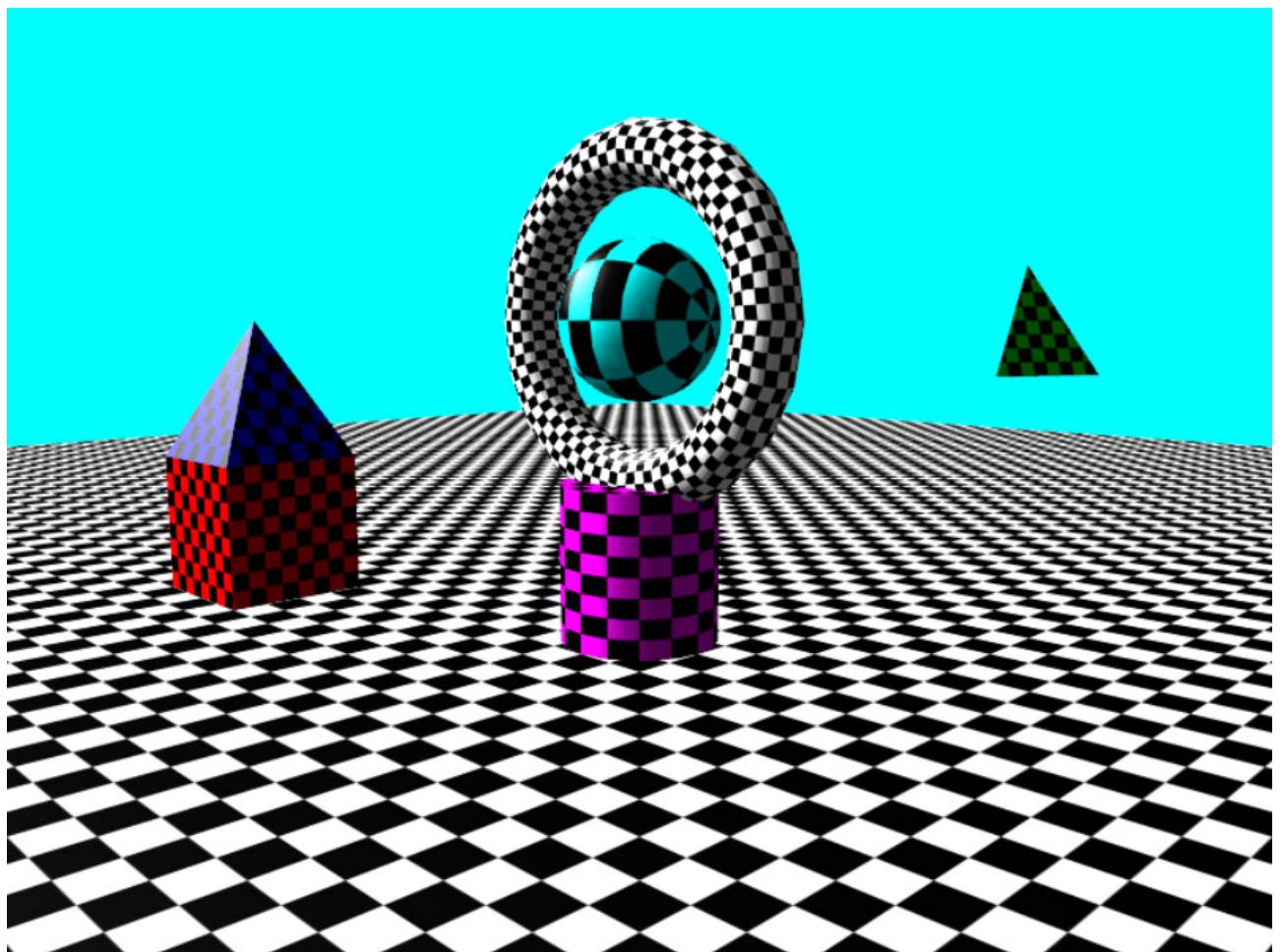


Figure 60.2: Output from Gouraud Shading Lesson

Lesson 61

Phong Shading

Our last main form of lighting, and the type we will actually use from now on, is Phong Shading. Phong Shading adds little from the point of view of how we calculate the light colour. However, rather than calculate light on the vertex shader a per-vertex lighting solution we will calculate the light on the fragment shader a per-pixel lighting solution.

Figure 61.1 illustrates the difference between flat (per-quad or per-triangle), Gouraud (per-vertex - what we have been doing up till now), and Phong (per-pixel) shading. You should be able to notice the difference between these three implementations.

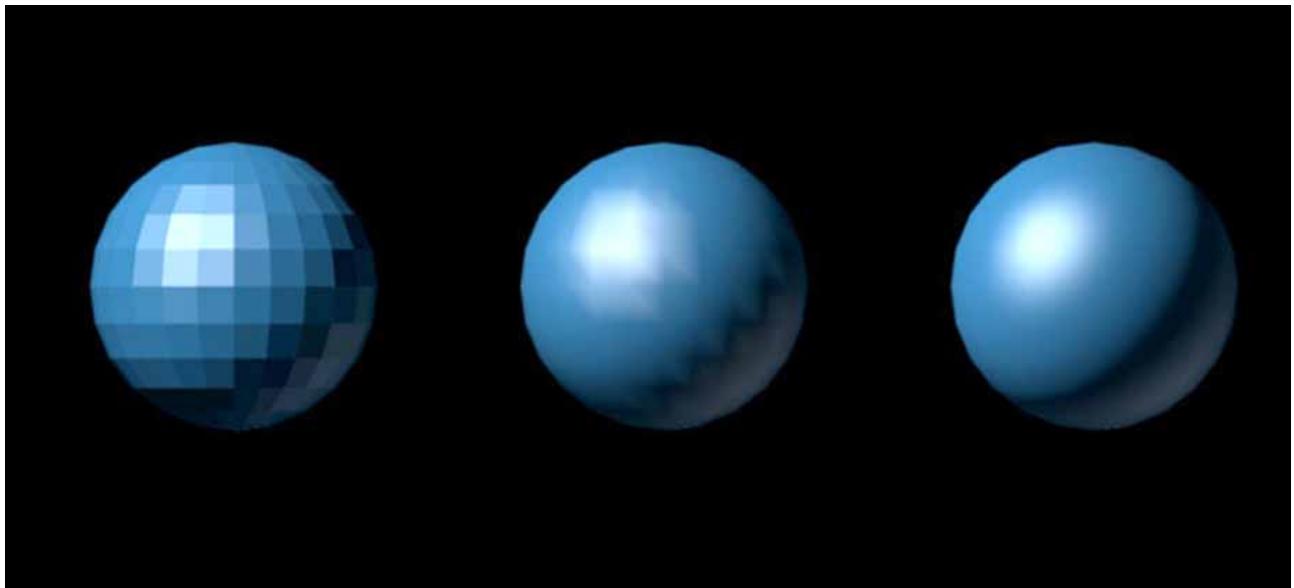


Figure 61.1: Flat, Gouraud and Phong Shading

Phong shading tends to be the approach taken in modern graphics applications, as although expensive in comparison to Gouraud shading, it is still relatively cheap to process. There are no new equations here, as the previous equations work as before. However, we will be reprogramming our vertex shader and fragment shader to suit.

61.1 Phong Shader

As we are not changing the general lighting equation we will not represent that here. The key is to realise that we are now doing most of the calculation in the fragment shader. You will find that our Phong shader has the same uniforms just in different places. Our pipeline now becomes:

<i>position</i>	<i>world_position</i>
<i>normal</i>	\Rightarrow <i>transformed_normal</i> \Rightarrow <i>colour</i>
<i>tex_coord</i>	<i>tex_coord</i>

The point now is that we will be forwarding much of our data from the vertex shader to the fragment shader. You have to understand a little about what this means. When you pass a value from the vertex shader (a per-vertex value) down the pipeline, when it reaches the fragment shader interpolation of the values occur across the surface. Figure 61.2 illustrates the interpolation of normals across a surface from one vertex to the other.

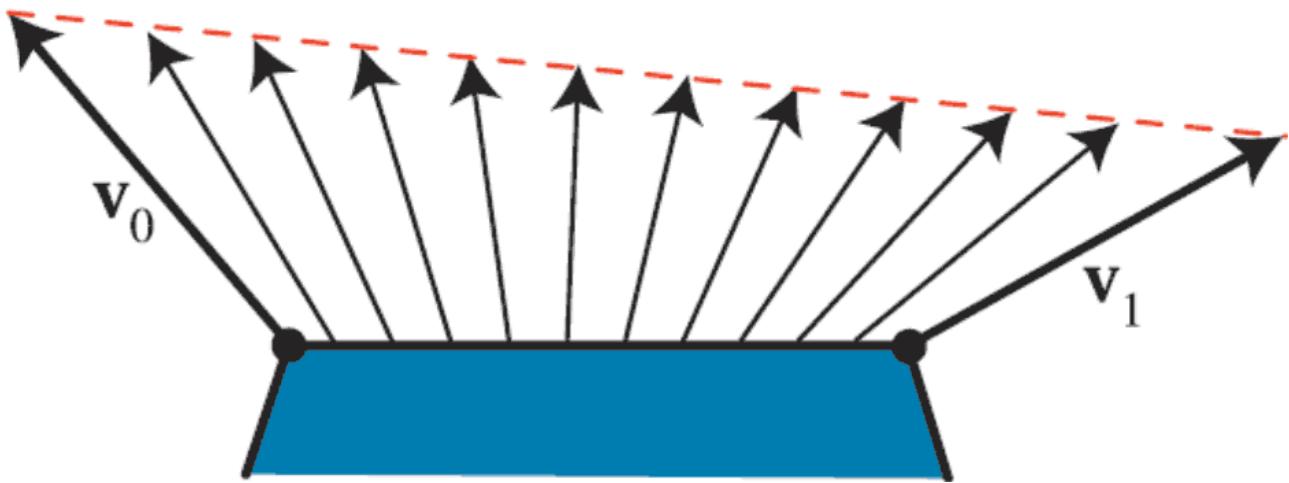


Figure 61.2: Normal Interpolation Across a Surface

As the normals are interpolated, when we calculate the lighting value for a fragment we have a slightly different normal. This provides our per-pixel lighting effect.

The structure of the shader is provided in Figure 61.3. And that is all you are getting for this lesson. You have to complete the Phong vertex and fragment shaders and update the main application accordingly.

The output from this lesson should be almost identical to the previous one. So much so that an image here will not do the difference justice. However, if you look closely at the curved surfaces you will see that the lighting looks smoother with less triangle edges being visible.

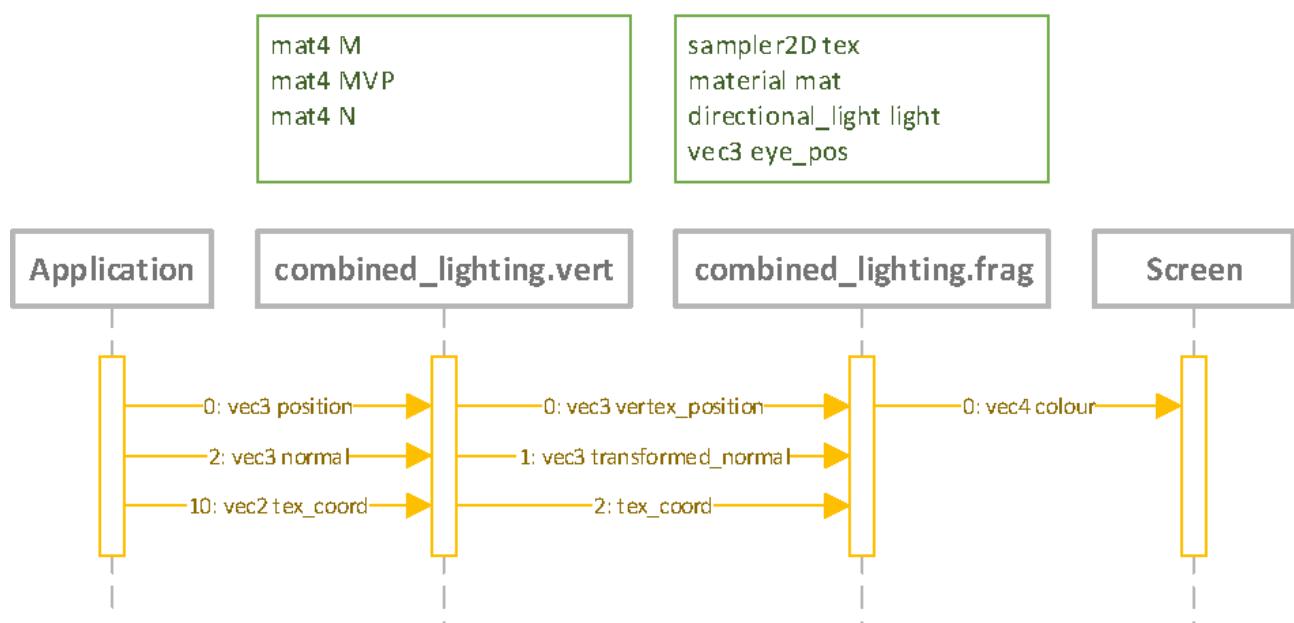


Figure 61.3: Phong Shader Structure

Lesson 62

Point Lights

Let us now try and add some other lighting types to our scene. The first lighting type we will examine is a point light.

62.1 What is a Point Light?

Point light works in many regards to normal lighting. However, now our light source has a position and it has a area which it lights. Figure 62.1 illustrates a basic point light.

Let us go back to our standard directional lighting calculation first, and then think about how we would implement a point light.

62.2 Standard Directional Light

Our standard directional light uses the following calculation:

$$\mathcal{K}_{primary} = \mathcal{E} + \mathcal{D}\mathcal{A} + \mathcal{D}\mathcal{C} \max(\mathbf{L} \cdot \mathbf{N}, 0.0)$$

$$\mathcal{K}_{secondary} = \mathcal{S}\mathcal{C} \max(\mathbf{N} \cdot \mathbf{H}, 0.0)^m$$

$$\mathcal{K} = \mathcal{K}_{primary} \mathcal{T}_1 \mathcal{T}_2 \cdots + \mathcal{K}_{secondary}$$

Our point light will fundamentally use the same calculation, but we have to rethink some of these values. The most obvious one is the light direction value \mathbf{L} . Light no longer has the same direction for the entire scene (we are no longer working with the Sun). Instead, light has a source, and the direction of the light to the surface is dependant on the position of the light relative to the surface. Therefore, we now calculate the light direction in the shader as follows:

$$\mathbf{L} = (\widehat{light_position - vertex_world_position})$$

That is the first change to our light equation. If we simply changed our lighting equation to this, we would achieve the effect shown in Figure 62.2.

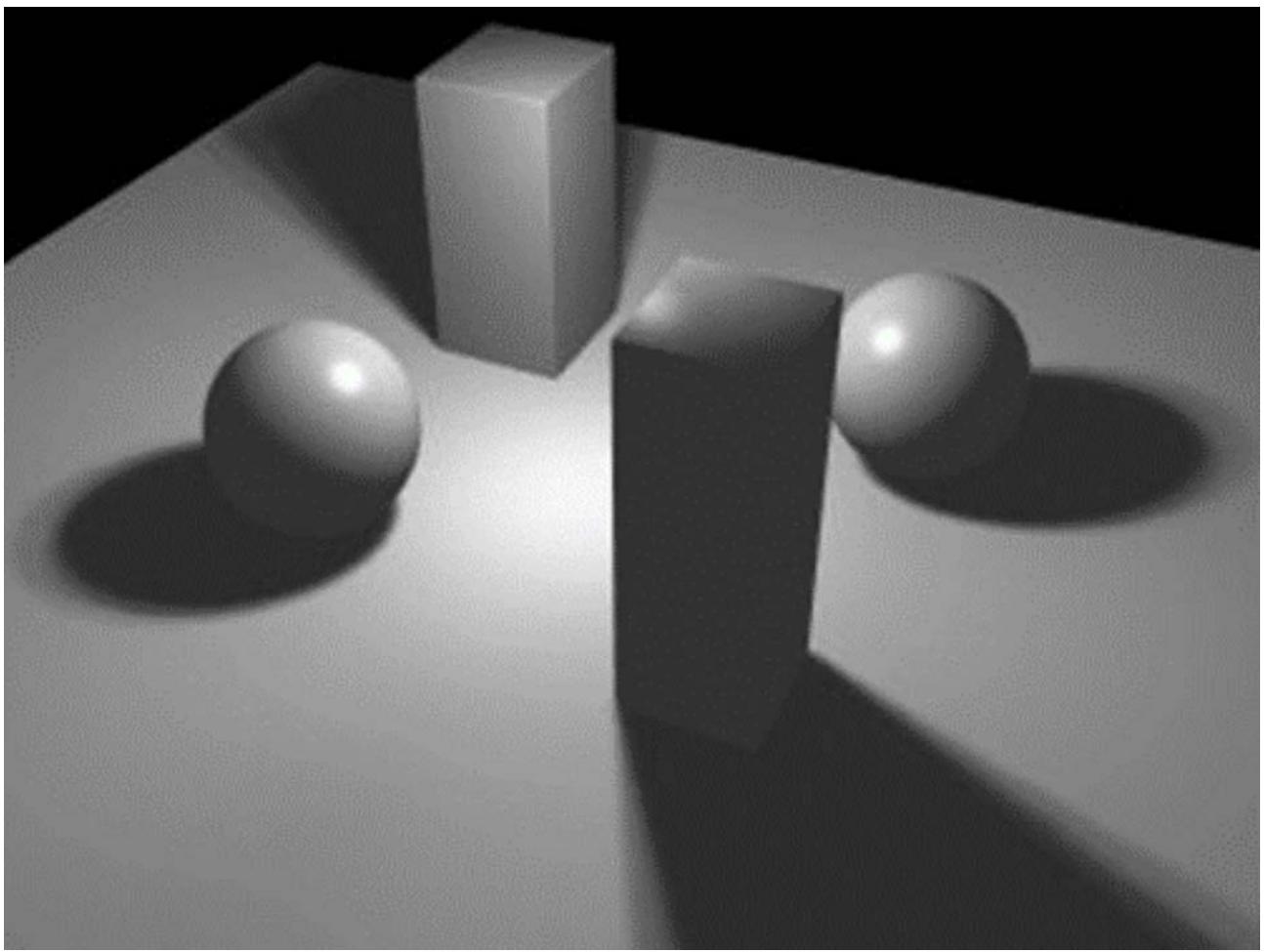


Figure 62.1: Point Light Example

This isn't really correct. We are not taking distance into account. A surface facing the light source will be lit no matter how far it is from the light source. This is obviously not what we want, so let us look at how we can define a point light.

62.3 Working with Distance

Let us now take distance into account when working with the point light. GLSL actually provides a function that allows us to get the distance between two points:

```
1 float d = distance(point1, point2);
```

We will start working with this distance value to try and create the lighting effect we want.

62.3.1 Attempt 1

For our first attempt, let us just check the distance and if it is within a certain range we use the light colour, otherwise we use black. A quick way of doing this in the shader is as follows:

```
1 float d = distance(point.position, position);
2 vec4 light_colour;
3 if (d < range)
```

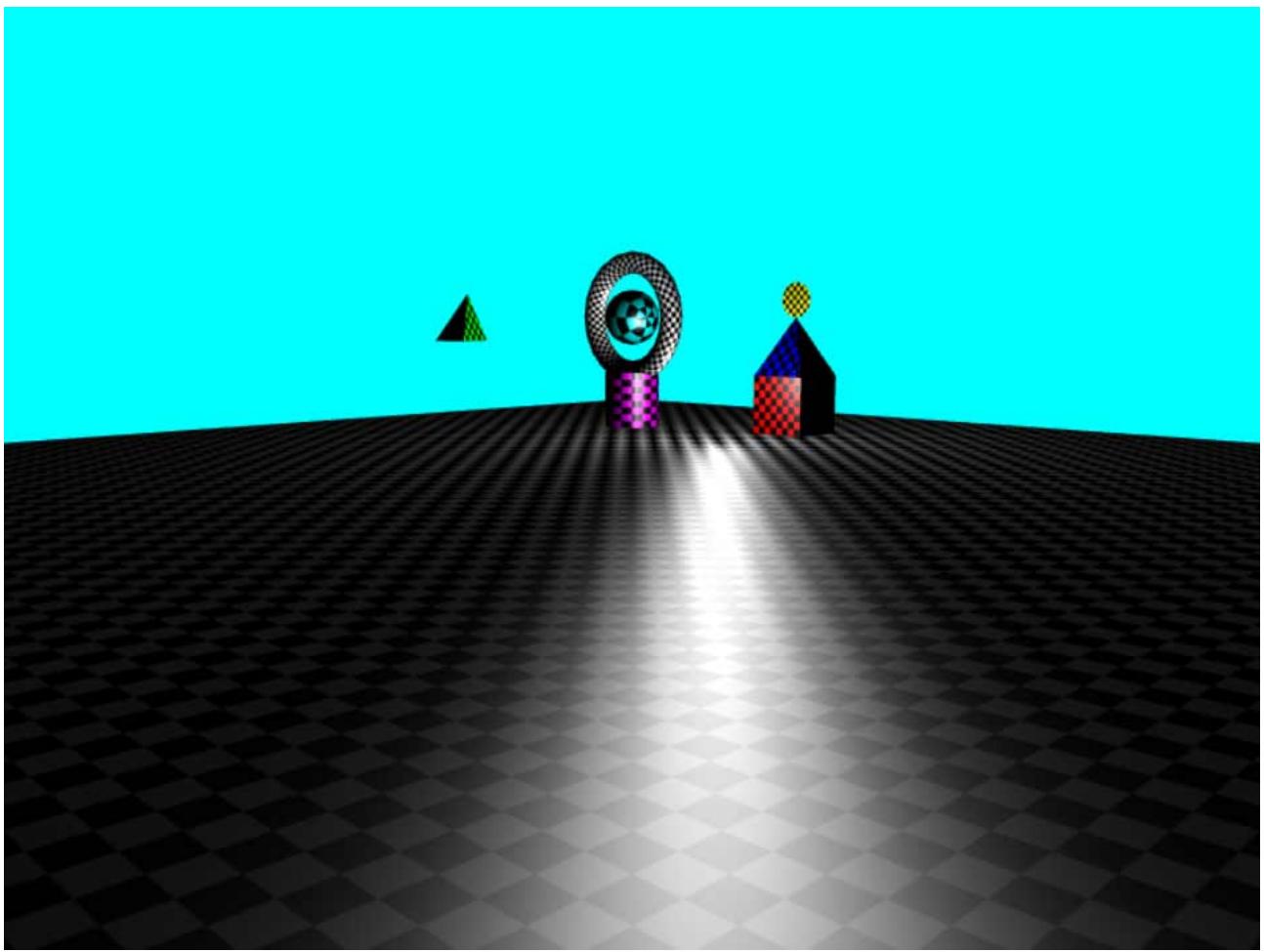


Figure 62.2: Point Light with No Distance

```

4     light_colour = point.colour;
5 else
6     light_colour = vec4(0.0, 0.0, 0.0, 1.0);

```

If we run an application with this shader we get the output shown in Figure 62.3.

So in our first attempt to include distance, we ended up making it look worse. This is because we have a cutoff point for the light. Light does not work like this. It diminishes over distance. It does not just stop. Let us rethink our approach.

62.3.2 Attempt 2

If we want to scale the light by a given range, we need to change our calculation. We can do this by scaling the light range by the distance, and multiplying the light colour by the scalar result. This would give us the following equation:

$$\text{light_colour} = \text{point_colour} \times \frac{\text{range}}{\text{distance}}$$

Running this attempt will give us the output shown in Figure 62.4.

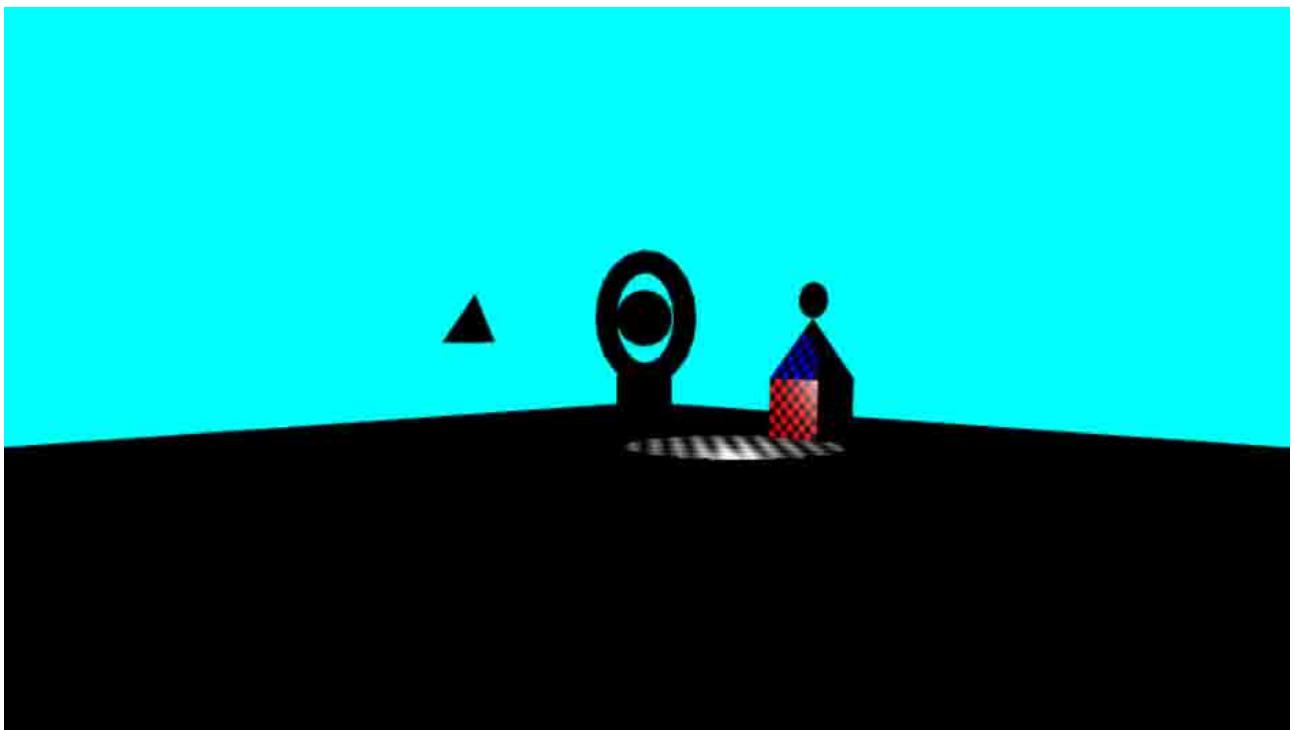


Figure 62.3: Point Light with Range

62.3.3 Attempt 3

Attempt 2 looks pretty good, and allows us to take range into account for our light calculation. We could just stop at that, but we actually want to allow some manipulation of the light value based on its brightness and how far it can light. To do this, we introduce three values to our point light:

constant a value used to determine how bright the light is. A low number (less than 1) provides a bright light. A large number (greater than 1) provides a dim light.

linear a value used to determine how the light degrades based on range. A high number here means that the light cuts off at a short range. A low number provides a light that travels far.

quadratic a values used to determine how quickly the light degrades over range. This value takes into account the square of the distance from the light. A high number here means the light will cutoff quickly (even with a low linear). A low value means that light will travel far.

This will provide us with the light value that we are interested in. These values are called *attenuation* factors. We will use them to build the equation for the light colour for a particular point.

62.3.4 Point Light Equation

The point light equation is as follows:

$$C = \frac{1}{k_c + k_l d + k_q d^2} C_0$$

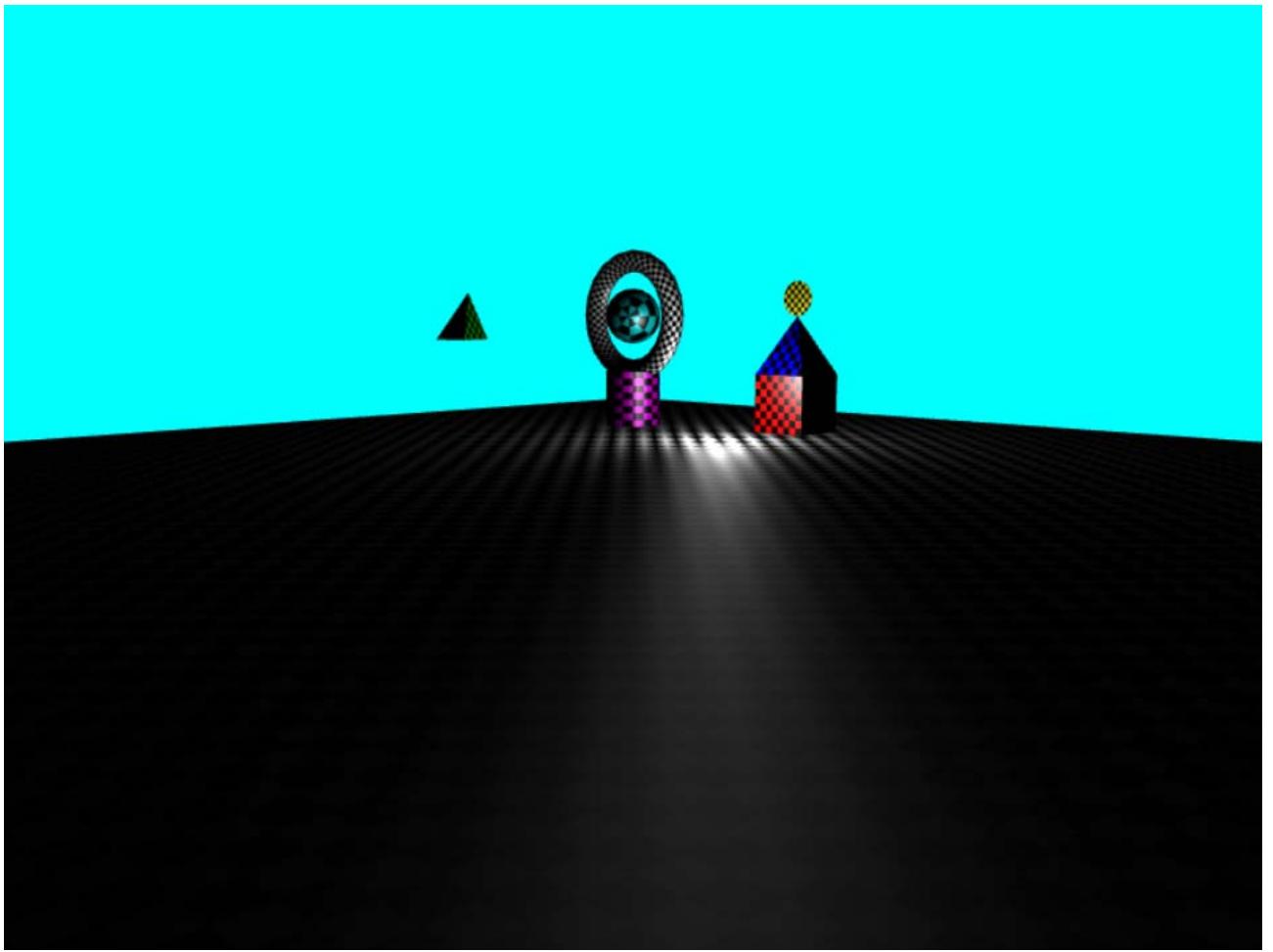


Figure 62.4: Scaled Point Light

Where \mathcal{C}_0 is the colour of the light, d is the distance to the light source from the vertex, and the constant values k_c, k_l and k_q are the attenuation values.

62.4 Point Light Data in the Render Framework

As with the directional light, the render framework provides a helper class to manage data representing a point light. The outline of this code is as below. The complete code can be found in `point_light.h`.

```

1 class point_light
2 {
3     private:
4         // The colour of the point light
5         glm::vec4 _colour;
6         // The position of the point light
7         glm::vec3 _position;
8         // The constant factor of the attenuation
9         float _constant;
10        // The linear factor of the attenuation
11        float _linear;
12        // The quadratic factor of the attenuation
13        float _quadratic;
```

```

14 public:
15     // Creates a point light with a default colour
16     point_light() :
17         _colour(glm::vec4(0.9f, 0.9f, 0.9f, 1.0f)),
18         _position(glm::vec3(0.0f, 0.0f, 0.0f)),
19         _constant(0.5f),
20         _linear(0.2f),
21         _quadratic(0.01f)
22     {
23     }
24     // Other constructors
25     // ...
26     // Getter and setter methods
27     // ...
28     // Sets the range of the point light
29     void set_range(float range)
30     {
31         _linear = 2.0f / range;
32         _quadratic = 1.0f / (powf(range, 2.0f));
33     }
34     // Moves the light by the given vector
35     void move(const glm::vec3 &translation)
36     {
37         _position += translation;
38     }
39 };

```

You should familiarise yourself with the default lighting values and the two helper methods - `set_range` and `move`. The `set_range` in particular is useful to allow a simple method of setting the general range of a light. You can then set the `constant` value with a standard setter.

To use the point light, the renderer provides a bind method:

```
1 renderer::bind(point_light, "name");
```

This works in the same manner as the directional light. Your goal in this exercise is to complete the point light shader and then set the necessary values to get the application to work. Figure 62.5 provides a sample output.

62.5 Exercises

1. Try and replicate the different effects achieved through the point light attempts undertaken in this lesson. This will involve some changes to the fragment shader to achieve the effect you are after.
2. Experiment with the different attenuation values to see the result. Try for bright lights with short ranges, and dim lights with long ranges.

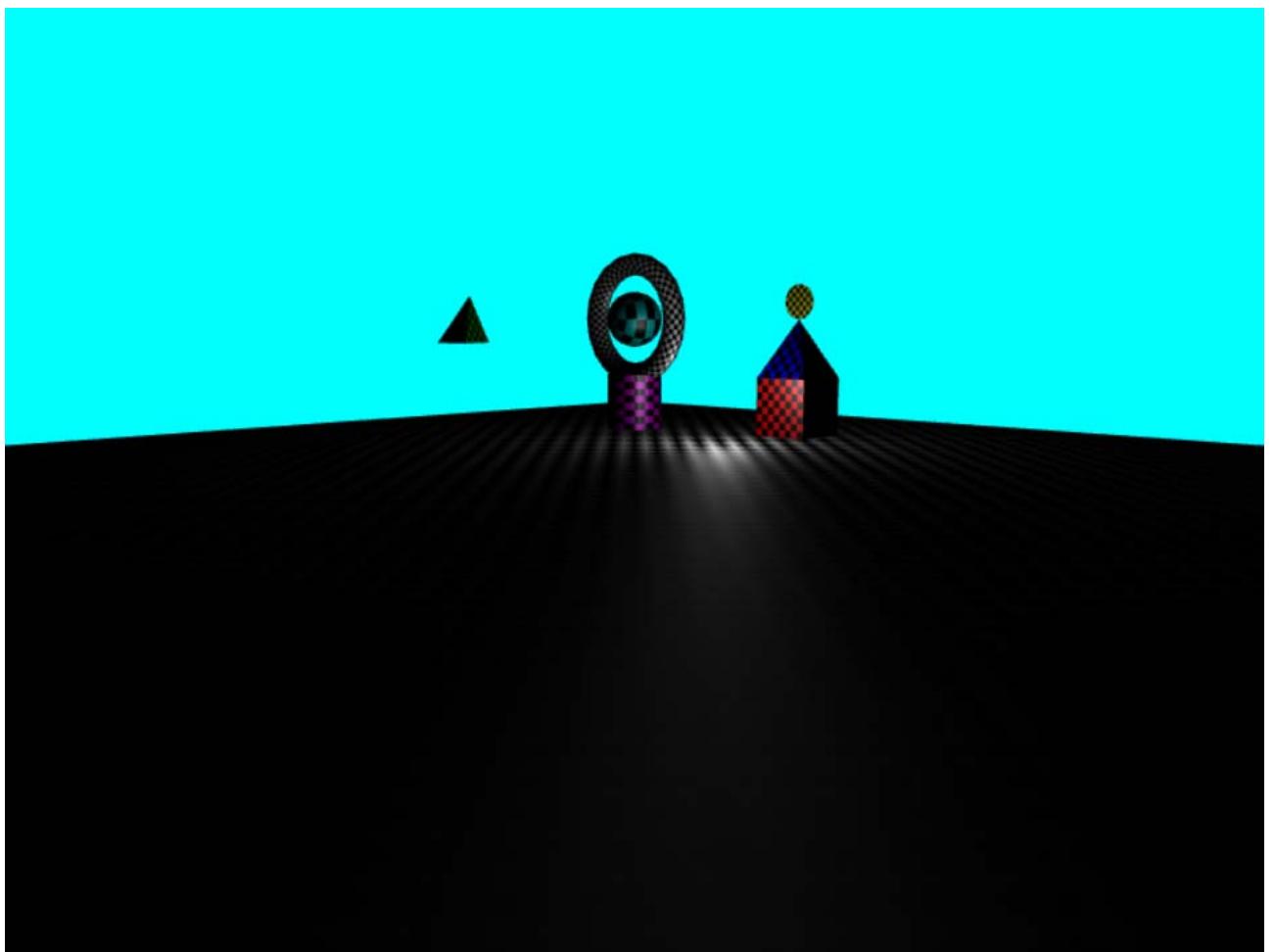


Figure 62.5: Output from Point Light Lesson

Lesson 63

Spot Lights

Our next form of light is a spot light. A spot light shares many similar properties to the point light, but now we have to consider the direction that the light is facing. As such, we will reuse much of our point light equation to emulate a spot light. This means attenuation is a factor again.

63.1 What is a Spot Light?

A spot light shares the properties of a point light (it has position and distance), but now we have a direction to consider as well. Figure 63.1 shows an example of a spot light.

Apart from that, we are still dealing with the same concepts as our other lighting calculations.

63.2 Standard Point Light

Remember from the previous lesson that we defined the point light colour equation as follows:

$$C = \frac{1}{k_c + k_l d + k_q d^2} C_0$$

We have our attenuation factors (k_c, k_l, k_q) and the colour of the point light (C_0). A spot light operates the same, except we must take into account the direction of the light. Luckily, we have done this before for our directional light equation.

63.3 Spot Light Equation

To calculate the spot light colour we use the following equation:

$$C = \frac{\max(-\mathbf{R} \cdot \mathbf{L}, 0)^p}{k_c + k_l d + k_q d^2} C_0$$



Figure 63.1: Spot Light

We are now using the direction of the light to the object (\mathbf{L}) and the direction the spot light faces (\mathbf{R}) to work out the light intensity. The calculation is based on the angle between these two vectors (using the dot product), and a value (p) that defines the power of the spot light. You should be able to understand this equation and how we derived it by now.

63.4 Spot Light Data in the Render Framework

A spot light in the render framework operates just like a point light, with `bind` methods to use them in an effect. The structure of the spot light data is shown below.

```

1 // An object representing spot light data
2 class spot_light
3 {
4 private:
5     // Colour of the spot light
6     glm::vec4 _colour;
7     // Position of the spot light.
8     glm::vec3 _position;
9     // Direction that the spot light faces
10    glm::vec3 _direction;
11    // The constant factor of the attenuation
12    float _constant;
```

```

13 // The linear factor of the attenuation
14 float _linear;
15 // The quadratic factor of the attenuation
16 float _quadratic;
17 // The power of the spot light
18 float _power;
19 public:
20     // Creates a spot light with a default colour
21     spot_light() :
22         _colour(glm::vec4(0.9f, 0.9f, 0.9f, 1.0f)),
23         _position(glm::vec3(0.0f, 0.0f, 0.0f)),
24         _direction(0.0f, 0.0f, -1.0f),
25         _constant(0.5f),
26         _linear(0.2f),
27         _quadratic(0.01f),
28         _power(10.0f)
29     {
30 }
31 // Other constructors
32 // ...
33 // Getter and setters
34 // ...
35 // Sets the range of the point light
36 void set_range(float range)
37 {
38     _linear = 2.0f / range;
39     _quadratic = 1.0f / (powf(range, 2.0f));
40 }
41 // Moves the light by the given vector
42 void move(const glm::vec3 &translation)
43 {
44     _position += translation;
45 }
46 // Rotates the light
47 void rotate(const glm::quat &rotation)
48 {
49     // Calculate new orientation
50     auto rot = glm::mat3_cast(rotation);
51     _direction = rot * _direction;
52 }
53 // Rotates the light
54 void rotate(const glm::vec3 &rotation)
55 {
56     rotate(glm::quat(rotation));
57 }
58 };

```

Your task in this lesson is to complete the necessary shader and complete the main code file. Essentially you are following the same procedure from the point light lesson. An example output from this lesson is shown in Figure 63.2. This position and state of the spot light has been achieved through the use of implemented controls to move and rotate the light.

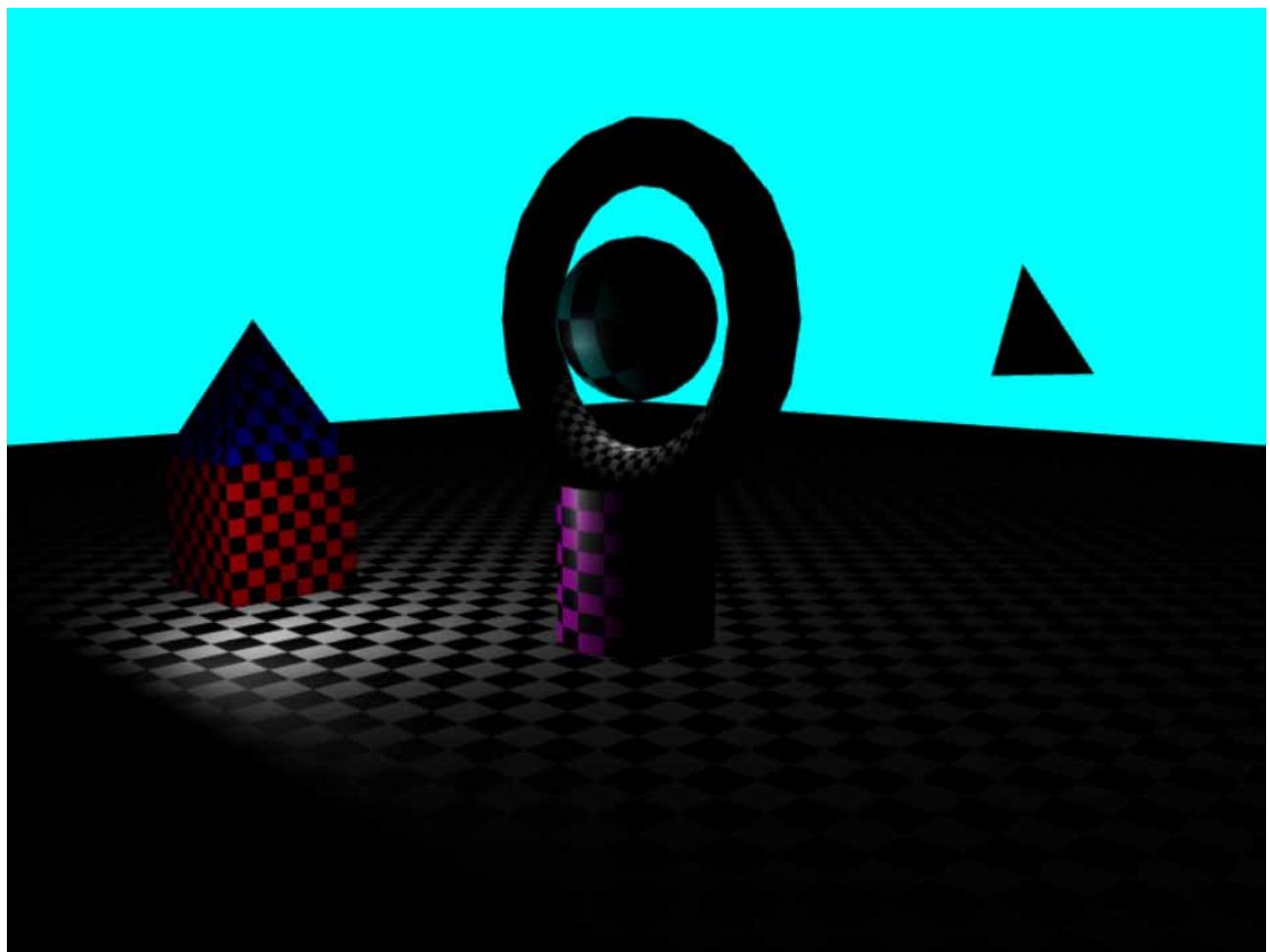


Figure 63.2: **Output from Spot Light Lesson**

Lesson 64

Multiple Lights

We will end our initial work on lighting by looking at how we can implement multiple lights in our scene. This requires us to use arrays for our shader uniforms and functions in our shaders. Let us look at array uniforms first.

64.1 Array Uniforms

So far, we have been declaring single value uniforms of the form:

```
1 uniform vec4 light_colour;
```

This is good for simple values, but causes us issues when we want to use more than one value to represent something. For example, we do not really want to write:

```
1 uniform vec4 light_colour_0;
2 uniform vec4 light_colour_1;
3 uniform vec4 light_colour_2;
4 // ... and so on
```

Thankfully, we can create uniform arrays quite happily in GLSL. Therefore, instead of the above, we can write:

```
1 uniform vec4 light_colour[10];
```

This makes our life much easier. However, you should note that GLSL does not allow runtime sized arrays (i.e. we cannot allocate an array with a non-constant value). Therefore, our array sizes our fixed at compile time. Normally you should ensure you have either provided adequate space to store the array, or you should know the maximum size and never go beyond it.

For our purposes, we need to create arrays of lights. Therefore, we have declarations as follows:

```
1 // Point light information
2 struct point_light
3 {
4     vec4 light_colour;
5     vec3 position;
6     float constant;
```

```

7   float linear;
8   float quadratic;
9 };
10
11 // Spot light data
12 struct spot_light
13 {
14   vec4 light_colour;
15   vec3 position;
16   vec3 direction;
17   float constant;
18   float linear;
19   float quadratic;
20   float power;
21 };
22
23 // Point lights being used in the scene
24 uniform point_light points[4];
25 // Spot lights being used in the scene
26 uniform spot_light spots[5];

```

Therefore we have 4 point lights we can allocate and 5 spot lights we can allocate.

64.1.1 Array Uniform Naming

There are a couple of different ways we can set these uniform arrays. We are going to use the simplest to understand (but not the most efficient) in our approach. As we discussed before, we can set the individual values in our structs by combining the name of the uniform and the name of the data value in the struct. So, for example, the position value of the point light structure would have the following uniform name:

```
1 uniform vec3 point.position;
```

For arrays, each member is given a unique name based on its index. If we go back to our original light colour example:

```
1 uniform vec4 light_colour[10];
```

This is actually a short cut for declaring the following uniforms in the shader:

```

1 uniform vec4 light_colour[0];
2 uniform vec4 light_colour[1];
3 uniform vec4 light_colour[2];
4 uniform vec4 light_colour[3];
5 uniform vec4 light_colour[4];
6 uniform vec4 light_colour[5];
7 uniform vec4 light_colour[6];
8 uniform vec4 light_colour[7];
9 uniform vec4 light_colour[8];
10 uniform vec4 light_colour[9];

```

Realise that these are names of uniforms we are using here. They are not array declarations.

For our lighting structures, this means we have the following uniform names:

```
1 uniform vec4 points[0].light_colour;
2 uniform vec4 points[1].light_colour;
3 // ... and so on
```

From the point of view of our render framework, this allows us to set these individual uniform values by their extended names:

```
1 auto idx = eff.get_uniform_location("points[0].light_colour");
2 glUniform4fv(idx, 1, value_ptr(points[0].light_colour));
```

This is useful information to know if you want to extend beyond the functionality provided by the render framework. However, for point and spot lights a helper method is provided to take care of the naming conventions. We will look at this at the end of the lesson.

64.2 Functions in Shaders

The other concept we need to introduce in this lesson is the use of functions in shaders. GLSL is a C based language, and as such we can use many of the basic concepts from C. This means we can write a function that returns a value. For example, we could write:

```
1 vec4 black()
2 {
3     return vec4(0.0, 0.0, 0.0, 1.0);
4 }
5
6 void main()
7 {
8     // ... do some other rendering stuff
9     vec4 temp = black();
10    // ... do more rendering stuff
11 }
```

This enables us to write reusable code and also write structured code (which makes our life easier). For our lesson, you will need to write a function that takes some values and returns a lighting value. There are two functions to implement - one for point lighting and one for spot lighting.

64.2.1 Looping in Shaders

This lesson also requires you to write a `for` loop. A `for` loop in GLSL is exactly the same as in standard C. For example, to sum all the point lights in our shader we would write:

```
1 for (int i = 0; i < 4; ++i)
2     colour += calculate_point(points[i], mat, position, normal, ←
        view_dir, tex_colour);
```

64.3 Binding Vectors of Lights

As mentioned, our render framework enables the binding of collections of point and spot lights to our shader. This operates exactly as binding a single light source:

```
1 vector<point_light> points;
2 // ... set up point lights
3 renderer::bind(points, "points");
```

The renderer will assume the `points` uniform is an array and will attempt to bind it as such. It will iterate through the `vector` until all the lights are bound.

64.4 Exercise

Your goal in this lesson is to complete the shaders and the main application to enable the use of multiple light sources. Figure 64.1 provides an example output.

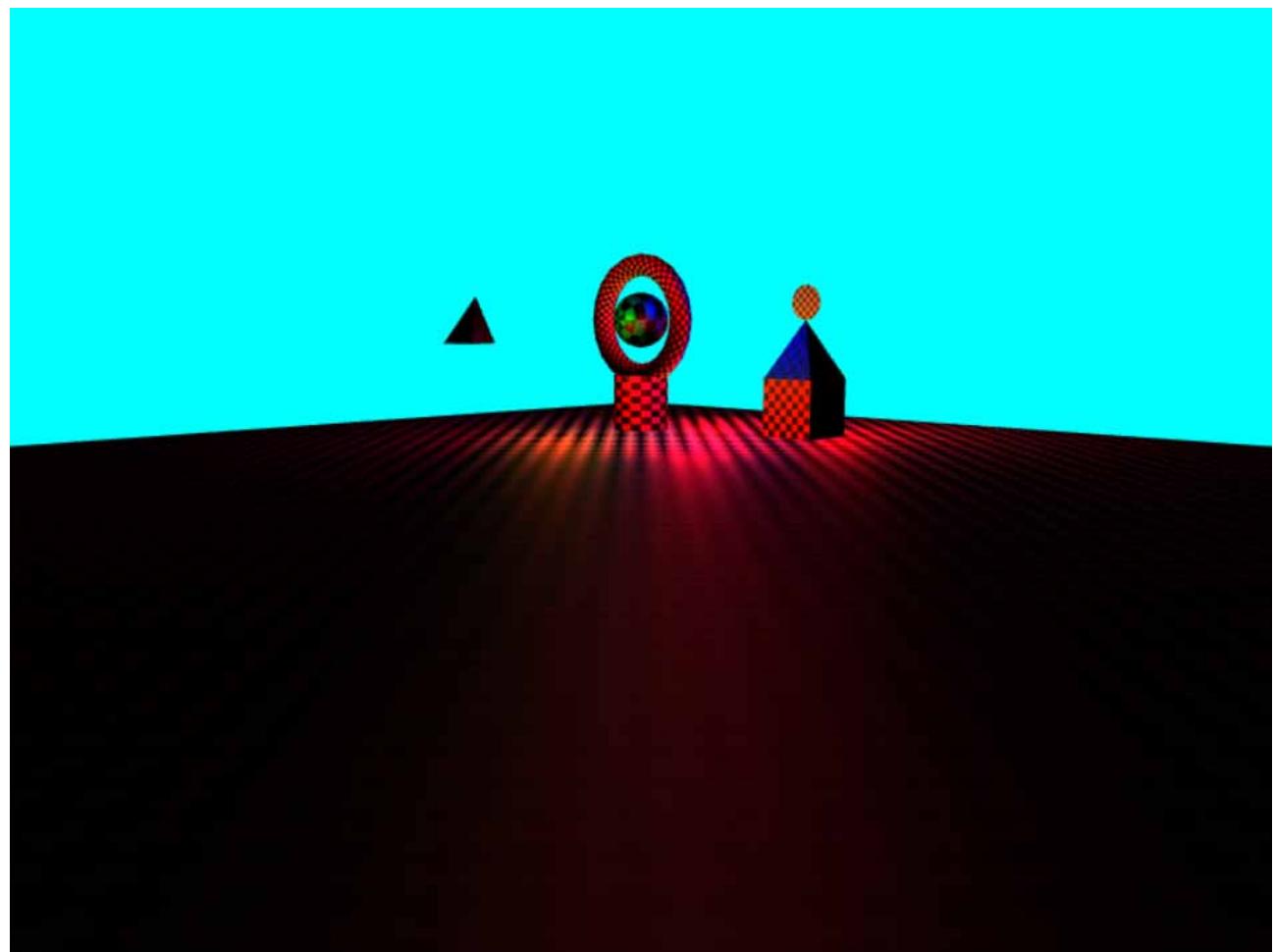


Figure 64.1: Output from Multiple Lights Lesson

Part VII

Lighting Effects

Lesson 65

Multi-file Shaders

Before starting on the other lighting effects let us look at how we can break our shader programs into separate parts. This allows us to reuse existing pieces of effects to construct larger complete effects.

65.1 What is a GLSL Program?

So far our work with shaders has involved us loading two files - a *vertex* shader and a *fragment* shader. This is the minimal requirement for writing shader programs. However, there is nothing stopping us loading multiple files of each type - as long as there are no conflicts. Figure 65.1 illustrates the general idea.

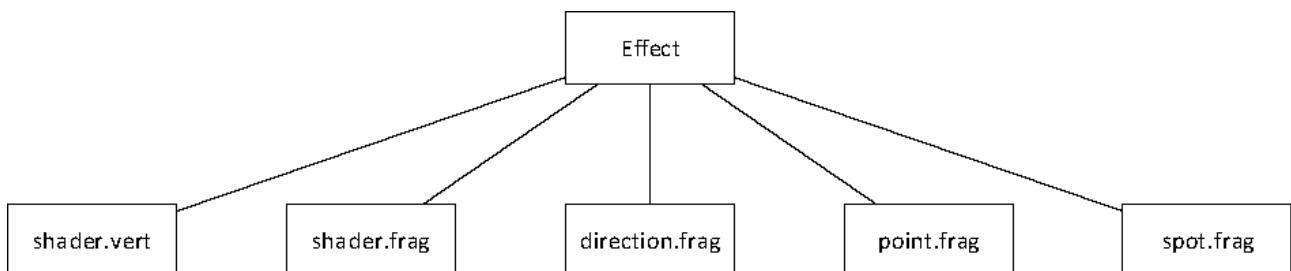


Figure 65.1: Structure of an Effect with Multiple Parts

Of course, this is fairly standard in programming - we split our code across multiple files. However, GLSL is not the most intuitive for doing this, so we have to do a bit more work. This is mainly because we are treating each file as an individual compilation unit rather than using file inclusion (i.e. header files) to enable this.

65.2 Loading Multiple Shaders in the Graphics Framework

To get started, let us look at how we can load multiple shaders into one effect in the graphics framework. We do this as follows:

```
1 // Name of fragment shaders required
```

```

2 vector<string> frag_shaders
3 {
4     "shader.frag",
5     "...\\resources\\shaders\\parts\\direction.frag",
6     "...\\resources\\shaders\\parts\\point.frag",
7     "...\\resources\\shaders\\parts\\spot.frag"
8 };
9 // Load shaders
10 eff.add_shader(frag_shaders, GL_FRAGMENT_SHADER);

```

We could also just make multiple calls to `add_shader` with each file name - it is effectively the same thing. This is the easy part of using multiple shader files. Let us now look at the hard part.

65.3 Ensuring we Don't Have Multiple Definitions

As each shader we load in GLSL is considered a separate compilation unit we need declare our `structs` in multiple files. This however will lead to multiple definition errors. Therefore we need to use *definition* guards to ensure that these new types only exist once. An example for our `directional_light` `struct` is shown below.

```

1 #ifndef DIRECTIONAL_LIGHT
2 #define DIRECTIONAL_LIGHT
3 struct directional_light
4 {
5     vec4 ambient_intensity;
6     vec4 light_colour;
7     vec3 light_dir;
8 };
9#endif

```

65.4 Example - Directional Light Shader

With the previous rule in mind let us look at how our `direction.frag` fragment shader part looks. This is shown below:

```

1 // Directional light structure
2 #ifndef DIRECTIONAL_LIGHT
3 #define DIRECTIONAL_LIGHT
4 struct directional_light
5 {
6     vec4 ambient_intensity;
7     vec4 light_colour;
8     vec3 light_dir;
9 };
10#endif
11
12 // A material structure
13#ifndef MATERIAL

```

```

14 #define MATERIAL
15 struct material
16 {
17     vec4 emissive;
18     vec4 diffuse_reflection;
19     vec4 specular_reflection;
20     float shininess;
21 };
22 #endif
23
24 // Calculates the directional light
25 vec4 calculate_direction(in directional_light light, in material mat←
26 , in vec3 normal, in vec3 view_dir, in vec4 tex_colour)
27 {
28     // Code for calculating directional light colour
29 }

```

Apart from the definitions at the top, this is just as you would expect. However, we have now got ourselves a reusable shader part for multiple effects. Much better. However, we still have another problem to address.

65.5 Defining that a Function Exists

Because we have no such thing as a header file in GLSL, the main shader part (e.g. `shader.frag`) has no idea what the `calculate_direction` function is. Therefore we have to declare it in our main shader file. This is just a standard function definition as follows:

```

1 // Forward declarations of used functions
2 vec4 calculate_direction(in directional_light light, in material mat←
3 , in vec3 normal, in vec3 view_dir, in vec4 tex_colour);

```

This indicates to the compiler that the function will exist, and when we build the effect (the linking part of the process) the function will be available (from the compiled `direction.frag` file). This is all just standard software development, just a bit more obtuse than we are familiar with.

65.6 Exercise

Your job in this lesson is to complete the shader parts -

- `direction.frag`
- `point.frag`
- `spot.frag`

You will also need to complete the shaders `shader.vert` and `shader.frag` to allow you to use these shaders. These are based on the previous example so you should be able to do so easily. Finally, the main program needs to be updated to use these shaders and set their

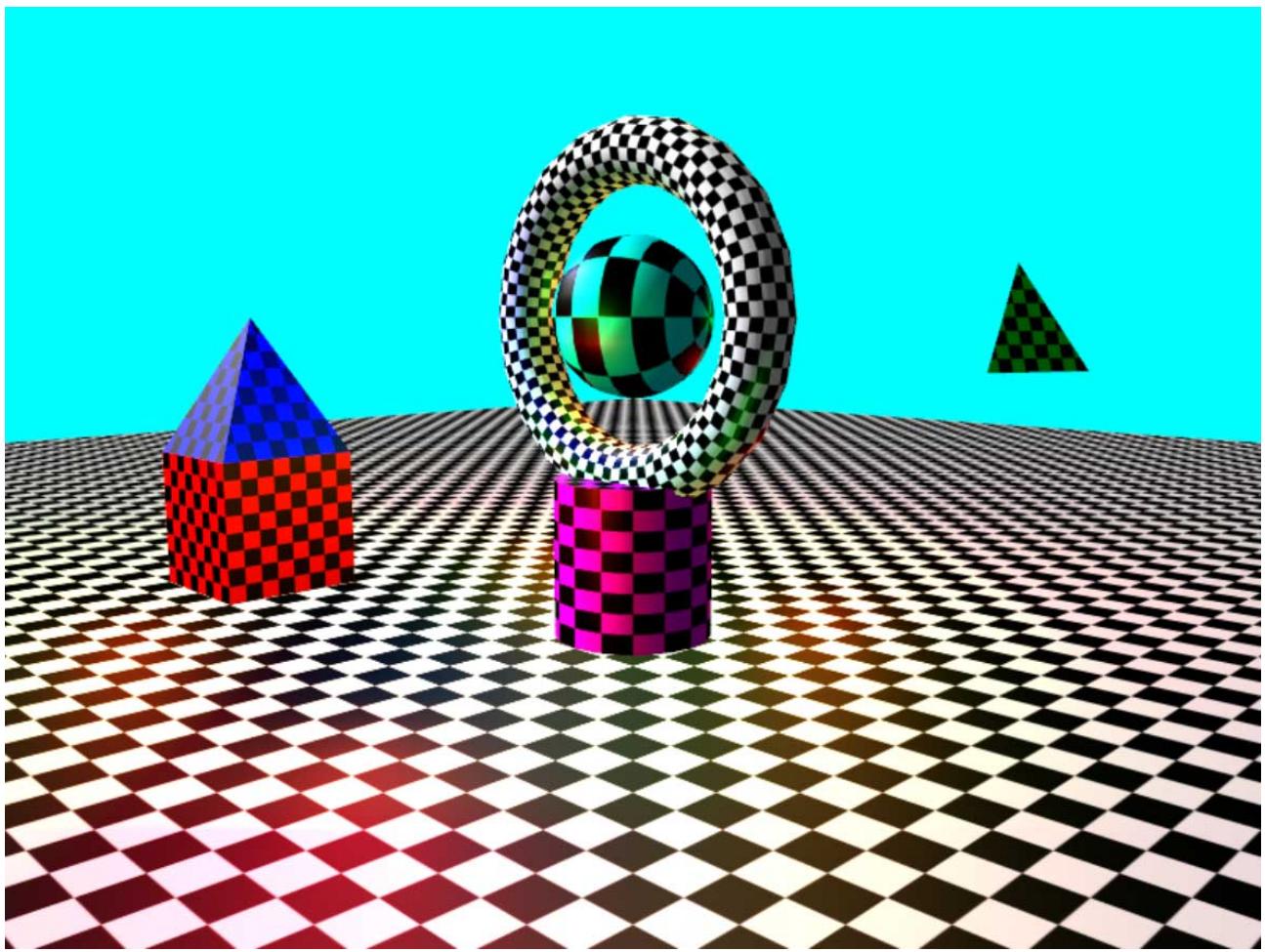


Figure 65.2: Output from Multi-file Shader Lesson

uniforms correctly. This application is the same as the last one, and the output is shown in Figure 65.2.

This is not the only way we could have approached multiple shader files. We could have joined all the different files together into one large `string` and compiled that. The limitation with this approach is that the line numbers for errors will not be accurate and therefore makes debugging *a lot* harder.

Lesson 66

Shadow Mapping

In our first lesson on more lighting effects we aren't actually going to render anything to the screen. Although you may think of shadowing as a complicated process (which it can be), we are simply going to use the depth buffer to determine if an item is in shadow or not. The principle we take is that we render the scene from the point of view of the light source, gather the depth information, and then use this information to determine if an object is in shadow. This is a very simplified model of shadowing but suits our purposes here.

66.1 The Depth Buffer

The *depth buffer* is one of the buffers of information the GPU keeps track of to enable correct rendering. Its main purpose is to determine which object is closest to the camera to determine the pixel colour. However, it can also be used to capture depth information to allow us to determine if an object is in shadow.

The depth buffer can be represented as a greyscale image with the pixel colour determining the depth of the pixel at that location. These can easily be examined using CodeXL as shown in Figure 66.1.

The technique we will be using to capture the depth buffer involves us capturing a *render pass* using a *render buffer*. We will use this method far more when we look at post-processing. For the moment now let us look at the two parts of the graphics framework that will support shadowing in this technique.

66.2 Creating a shadow_map in the Graphics Framework

The graphics framework comes with two objects that can be used to support our shadow mapping approach - `depth_buffer` and `shadow_map`.

66.2.1 `depth_buffer`

The `depth_buffer` object is responsible for maintaining our depth information. It has two parts:

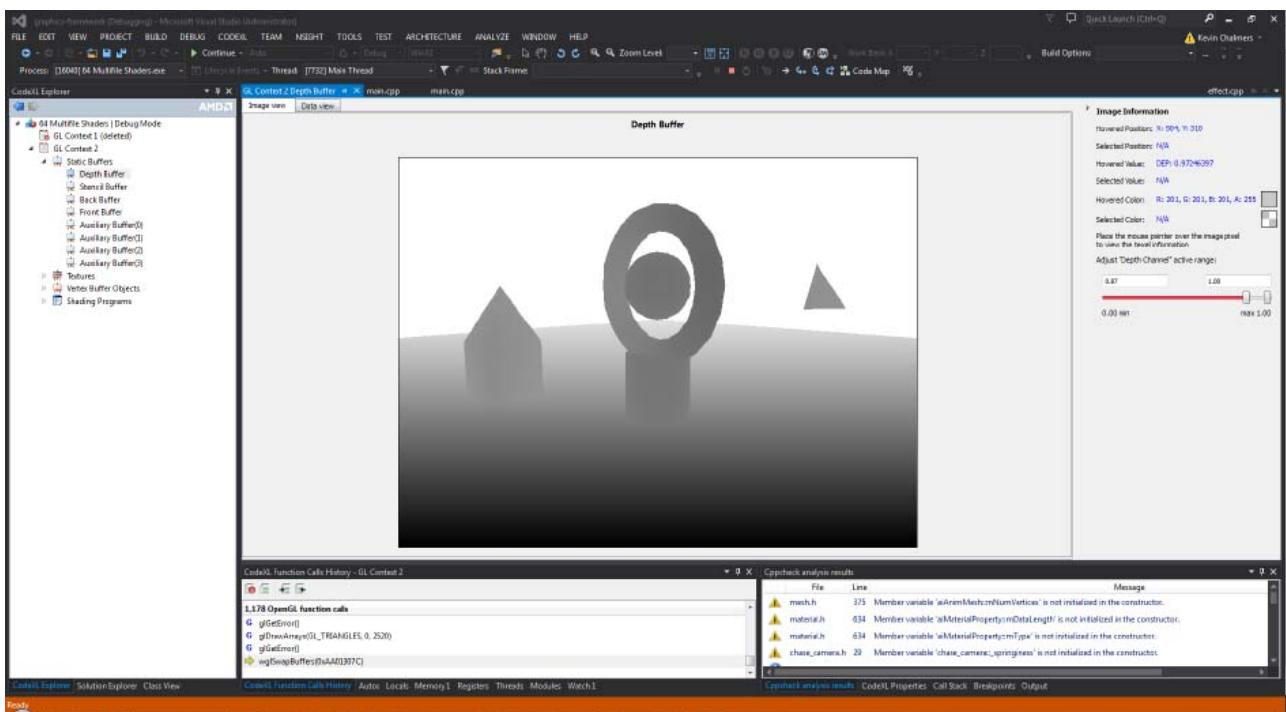


Figure 66.1: CodeXL Depth Buffer View

1. The OpenGL ID for the buffer to render to
2. A `texture` object allowing access to this buffer's data

The `depth_buffer` is simply a place holder allowing us to select it as a render target rather than rendering to the screen. This means we are rendering to a location in memory and using this data (actually the screen is just memory anyway - we are just redirecting the render). The `texture` object is what allows us access to the depth information.

66.2.2 shadow_map

The `shadow_map` object contains our captured `depth_buffer` but also information regarding how the depth was rendered. This involves us requiring the location of the light being used and its direction - essentially the same as a spot light. This allows us to determine if an object is within the shadow's coordinate space. We will look at this in the next lesson.

66.3 Rendering to a Depth Buffer

To create a `shadow_map` in the graphics framework we use the following code:

```
shadow = shadow_map(width, height);
```

In general the `shadow_map` should be the width and height defined for our screen.

For our technique we will be using a spot light to determine the shadow. Therefore our shadow map should use the same position and direction as the spot light:

```
shadow.light_position = spot.get_position();
shadow.light_dir = spot.get_direction();
```

This should be done in the update method.

For the method of shadowing we are using we capture the depth by rendering the back of the geometry only. This makes sense as it is only the back of the object that would create a shadow. To do this we have to change the faces that OpenGL culls from back to front. This is actually quite simple to do.

When it comes to rendering to our shadow map we perform the following steps:

1. Bind shadow map as render target (effectively binds the depth buffer)
2. Clear the depth buffer
3. Set culling to front face
4. Render scene using shadow effect
5. Set culling to back face
6. Set render target to the screen

To bind the shadow map we use `renderer::set_render_target` with the `shadow_map`. To set the render target as the screen again simply use `set_render_target` with 0.

To clear a buffer in OpenGL we use `glClear`. For the depth buffer we pass the argument `GL_DEPTH_BUFFER_BIT` into this call.

To change the face we are culling we use `glCullFace`. The two options here are `GL_FRONT` and `GL_BACK`.

66.4 Completing the Lesson

You have to complete the lesson using the techniques described above. There are actually only a few lines to change.

This lesson provides no output to the screen as we are capturing that render. However, we can see what is happening. When `s` is pressed during the running of the application the depth buffer texture is saved as “`test.png`” in the project folder. An example of the output is shown in Figure 66.2.



Figure 66.2: Captured Depth Buffer

Lesson 67

Shadowing

The second stage of our shadow mapping approach involves us rendering the shadows. This just involves us writing a shadow shader that takes in our depth texture and uses it to determine if a fragment is in shadow or not. Let us look at the shaders first.

67.1 The `shadow.frag` Shader

The high level view of our shader is illustrated in Figure 67.1.

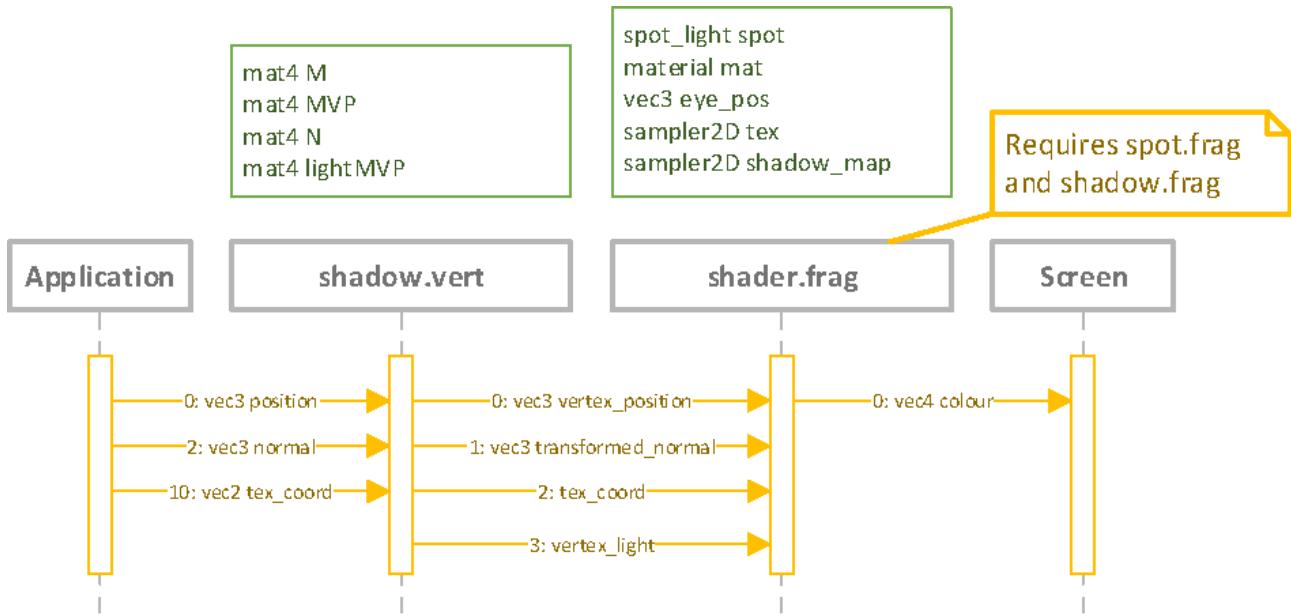


Figure 67.1: **Shadow Mapping Shader**

`shadow.frag` is a small shader file that needs to be included with other shaders to work. Its job is to determine if a particular fragment is in shade or not. It does this by taking the position of the fragment in the light space (in other words as if the viewer was located in the light's position) and using this to work out a texture coordinate to sample the depth buffer from. This allows us to determine if the given fragment is behind something that would cause a shadow. We use this to determine if we should return a shade value (0.5) or a lit value (1.0). Algorithm 21 provides an overview of this algorithm.

Algorithm 21 Calculate Shadow Algorithm

```

1: function CALCULATE_SHADOW(shadow_map, light_space_pos)
   ▷ Convert light_space_pos from homogenous coordinates
2:   proj_coords  $\leftarrow$  light_space_pos.xyz/light_space_pos.w
   ▷ Calculate shadow map coords to sample
3:   shadow_tex_coords.x  $\leftarrow$   $0.5 \times \text{proj\_coords.x} + 0.5$ 
4:   shadow_tex_coords.y  $\leftarrow$   $0.5 \times \text{proj\_coords.y} + 0.5$ 
   ▷ Check shadow coordinates in bounds [0, 1]
5:   if not in bounds then
6:     return 1.0
   ▷ Determine z coordinate of fragment
7:   z  $\leftarrow$   $0.5 \times \text{proj\_coords.x} + 0.5$ 
   ▷ Sample shadow map
8:   depth  $\leftarrow$  TEXTURE(shadow_map, shadow_tex_coords).x
9:   if depth = 0 then
   ▷ No depth value at the position
10:    return 1.0
11:   else if depth < z + epsilon then
   ▷ epsilon is for floating error (0.001)
12:     return 0.5
   ▷ in shade
13:   else
14:     return 1.0
   ▷ not in shade

```

67.1.1 Exercise

Complete the file `shadow.frag` in the `shaders/parts` folder. The algorithm above will help.

67.2 Calculating the Light Space Coordinate

The file `shadow.vert` (in the `shaders` folder) calculates the light space coordinate for our vector. This is actually no different than transforming the position in screen space - we just use a transformation for the light as a viewer instead:

$$P_{ls} = M_{ls} \cdot P$$

In the shader file you just need a line to perform this calculation using the `lightMVP` transformation matrix.

67.2.1 Exercise

Now complete the `shadow.vert` shader file. The calculation is very similar to the standard one for `gl_Position`.

67.2.2 Exercise 2 - Complete `shader.frag`

`shader.frag` contains all the comments you need to complete the effect. IT is just the spot light effect with an additional call to `calculate_shadow`.

67.3 Performing the Render

The main function for the lesson contains all the comments you need to complete it. The only thing you need to know is how to calculate the `lightMVP` matrix. This is done by getting the following matrices

`lM` - the standard model matrix for the object being rendered

`lV` - the view matrix taken from the `shadow_map` object

`lP` - the projection matrix from the camera

The rest is just a standard transformation matrix calculation. Once the lesson is completed and run you will get an output similar to that shown in Figure 67.2.

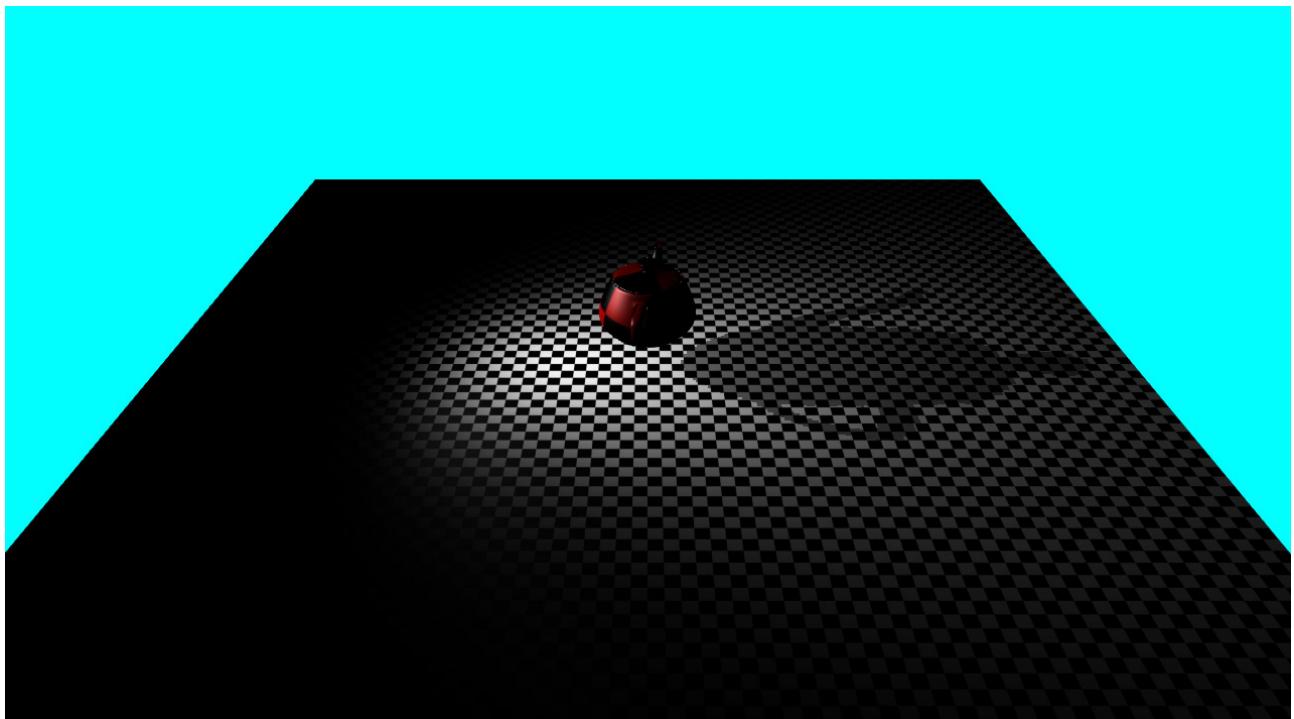


Figure 67.2: Output from Shadow Mapping Lesson

67.4 Exercise

Change the program to use a free camera and add code to move the light around to see the effect.

Lesson 68

Normal Mapping

Normal mapping is a technique that allows us to calculate the normals on a per-pixel level. This is done using a normal map, similar to the image shown in Figure 68.1.

The RGB values of the texture represent the $\langle x, y, z \rangle$ components of the normal at that point in the texture. The texture looks blue because most of the normals face towards us on the Z-axis.

To use the normal map, we need to work within a new coordinate space called the tangent space. The tangent space is essentially a coordinate space based on the normal at a particular point on the model. Figure 68.2 should help illustrate:

The normal vector becomes one axis in our coordinate space the bi-normal and the tangent being the others. This is actually quite trivial to work out. We need to create a transformation matrix as follows using these values:

$$TBN = \begin{bmatrix} T_x & B_x & N_x \\ T_y & B_y & N_y \\ T_z & B_z & N_z \end{bmatrix}$$

where T is the tangent vector, B the bi-normal vector, and N the normal for the surface. Remember that we will have to transform these by the normal matrix to get the actual surface values.

The TBN matrix is used to transform the sampled normal. The general algorithm for doing this is shown in Algorithm 22.

Algorithm 22 Calculating a Normal from a Normal Map

```
1: function CALC_NORMAL(normal, tangent, binormal, normal_map, tex_coord)
2:   sampled_normal  $\leftarrow$  TEXTURE(normal_map, tex_coord).xyz
                                 $\triangleright$  transform sampled_normal to [0, 1]
3:   sampled_normal  $\leftarrow$   $2.0 \times \text{sampled\_normal} - (1.0, 1.0, 1.0)$ 
                                 $\triangleright$  generate TBN
4:   TBN  $\leftarrow$  MAT3(tangent, binormal, normal)
                                 $\triangleright$  transform sampled_normal by TBN and return
5:   return TBN  $\times$  sampled_normal
```

The returned normal is what we use in our lighting calculations. These two new values - *tangent* and *bi-normal* can be calculated on the fly by our vertex shader. However, a better method is

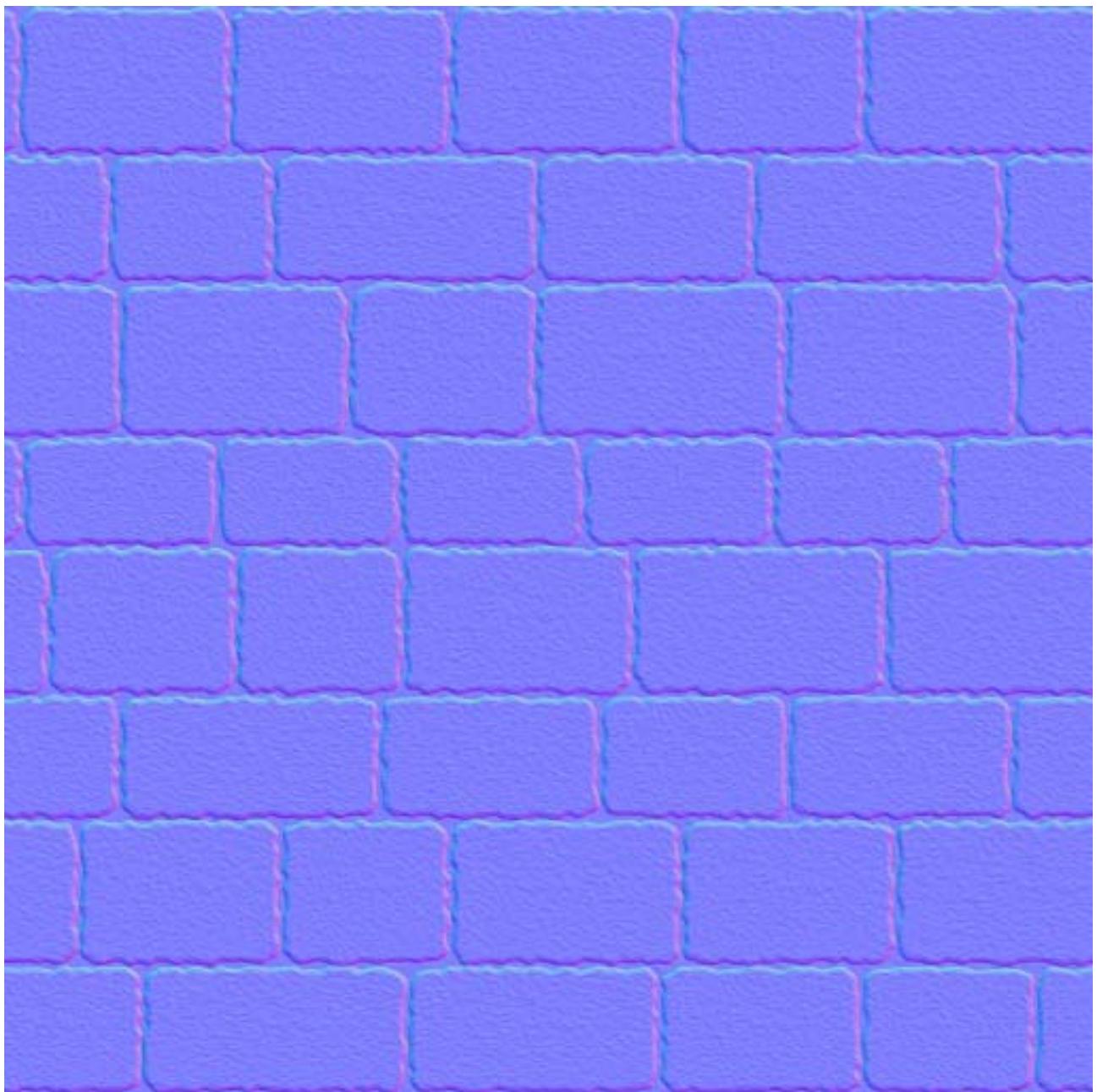


Figure 68.1: A Normal Map

to pre-calculate these. The graphics framework does so for any geometry it creates - **however loaded models only have this information if it is provided. You might have to calculate these yourself (doing some research).**

68.1 Shaders

Algorithm 22 has already provided you with the basic code for the shader part you need to complete for this lesson (`resources/shaders/parts/normal_map.frag`). The overall shader structure is shown in Figure 68.3.

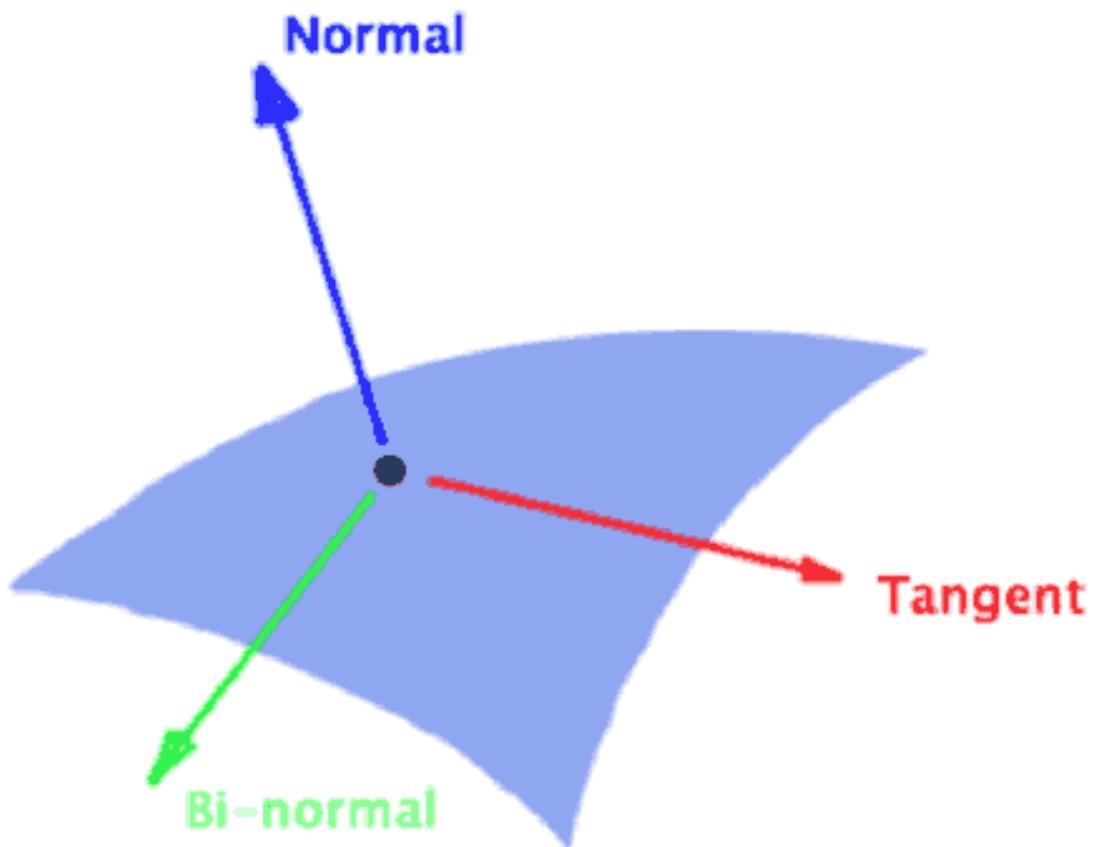


Figure 68.2: Tangent Space

68.2 Completing the Lesson

You will have to implement the algorithm in the shader file `normal_map.frag`. The other two shader files - `shader.vert` and `shader.frag` will also need completed to use the necessary data types. The main application only requires you to set in some values in `load_content` and set the `normal_map` uniform. Figure 68.4 provides the output from this lesson.

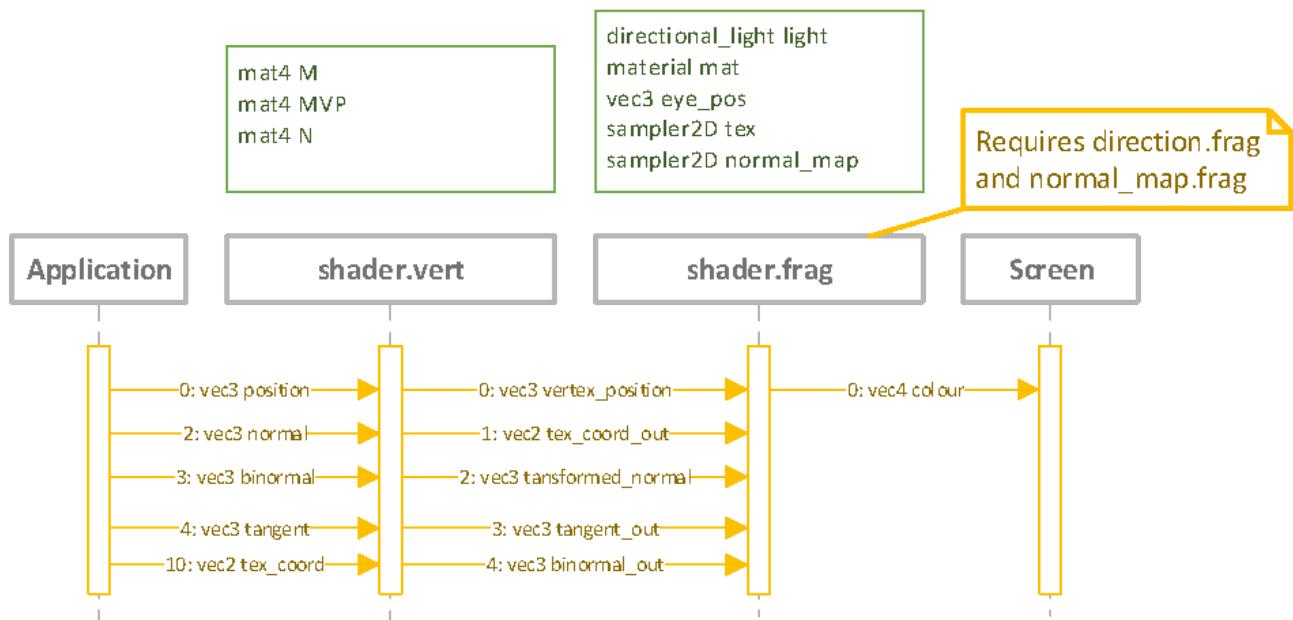


Figure 68.3: Normal Map Shader Structure



Figure 68.4: Normal Map Output

Lesson 69

Cube Maps

We are now going to use another form of texturing - cube maps. Cube maps are a form of 3D texturing which is commonly used for skyboxes. They provide a nice way of mapping a box to a texture. This is a useful for techniques such as skyboxes and environment maps which we will be doing shortly.

69.1 What is a Cubemap?

A cube map is simply six images joined together to create a texture that can be mapped onto a cube. Figure 69.1 provides an example.

This texture is made up of six parts, relating to the six planes that make up a cube:

- Positive X and negative X
- Positive Y and negative Y
- Positive Z and negative Z

What we need to do is use 3D texture coordinates to determine which part of the texture to use. We do this by using 3D texture coordinates.

69.2 3D Texture Coordinates

For a cube map, we need to determine which part of the texture to use based on a 3D texture coordinate. This actually works in a similar manner to 2D coordinates, but with some extra thought put in place.

Each individual piece of the cube map is just a texture with texture coordinates ranging from -1.0 to 1.0 on each axis. Therefore, if we want to use the positive X texture, we set the X component to 1 (i.e. $\langle 1, y, z \rangle$) and then work with texture coordinates as normal. This does make our life easy, particularly with skyboxes, which already have these coordinates developed for the positions.



Figure 69.1: An Example Cubemap

69.3 Loading a Cubemap in the Graphics Framework

The render framework has cubemap capabilities built in using the `cubemap` type. All you need to do is declare a `cubemap` value:

```
cubemap cube_map;
```

Then load it in your `load_content` method by providing the necessary names for the textures. An example is below:

```

1 array<string, 6> filenames =
2 {
3     "posx.png",
4     "negx.png",
5     "posy.png",
6     "negy.png",
7     "posz.png",
8     "negz.png"
9 };
10
11 cube_map = cubemap(filenames);

```

Note the order that the files are specified - **this is very important**. Doing these in the wrong order will cause your cubemap to be loaded incorrectly.

69.4 Completing the Lesson

To complete this lesson all you need to do is load in a cubemap as described above. We aren't using the cubemap yet. There is an example cubemap in `resources/textures/cubemaps/alien`. You will only see a sphere at the moment, but if you use CodeXL you will be able to see the loaded cubemap. Figure 69.2 provides an example.

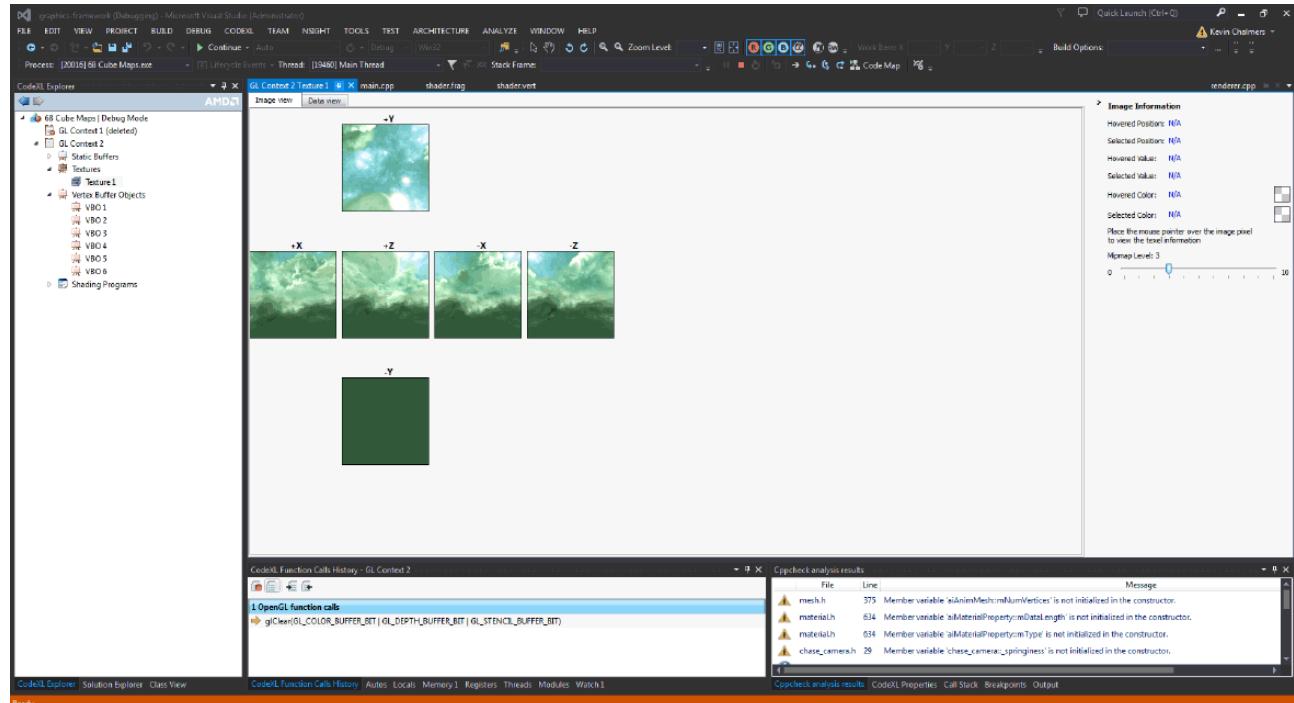


Figure 69.2: CodeXL Cubemap View

Lesson 70

Skybox

So we have loaded a cubemap. Now what? Well the main point of working with cubemaps is to generate skyboxes. To do this we need to add some geometry to our code.

70.1 Skybox Geometry

A skybox is a cube that the viewer is in the centre of. This means that the cube geometry is actually inside out from our normal point of view. We want the internal parts of the cube to be rendered rather than the external parts. You have already created a box the right way round for rendering - now you need to create one with the rendered faces on the inside (remember your winding order!).

70.1.1 Exercise - Create Skybox Geometry

It might have been a while since you had to generate your own geometry by hand but this will give you a nice refresher. Create a box geometry as specified and attach it to the `skybox` mesh. Then scale the mesh by 100 on each dimension to make the box big enough to sit in (*scale the mesh - don't just make the box dimensions 100*).

70.2 Rendering a Skybox

A skybox is just what should be on the horizon of the view. As such, it should be rendered first, but also rendered so that no depth information is provided (since it is the furthest away object). Therefore, to render a skybox we have to go through a number of stages:

1. Disable depth testing
2. Disable depth mask
3. Bind skybox effect (more on this soon)
4. Calculate *MVP* for skybox (it is just a mesh)
5. Bind cubemap
6. Render skybox

7. Enable depth mask
8. Enable depth testing
9. Render rest of scene

To disable depth testing in OpenGL we use `glDisable(GL_DEPTH_TEST)` and to enable it again we use `glEnable(GL_DEPTH_TEST)`. To enable and disable the depth mask we use `glDepthMask` with either `GL_TRUE` or `GL_FALSE`. Binding a cubemap is just the same as binding a texture.

70.3 Skybox Shader

We won't actually explain this much because it is really that easy. From the point of view of the texture coordinate we require to sample from we just use the position of the vertex. It is that easy. This coordinate maps to the expected texture coordinates for the cubemap at that position.

To sample the texture we just use the `texture` function as before but using the `cubemap` texture (which is of type `samplerCube`).

70.4 Completing the Lesson

You now have everything you need to render the skybox. There is a line of code that you need to add to the `update` method, but this is indicated. Once completed your output should be similar to that shown in Figure 70.1.

70.5 Exercise

Modify the code to use a free camera so you can look around the scene and determine that the skybox is working correctly.



Figure 70.1: **Output from Skybox Lesson**

Lesson 71

Environment Maps

The reason we are using the sphere in our scene is for our next shader effect - an environment map. An environment map is where an object reflects part of its environment using a cubemap to determine what that reflection looks like. Doing this is actually quite trivial, and we will only look at the shader you need to update. The main application you should be able to do yourself.

The structure of the shader is given in Figure 71.1.

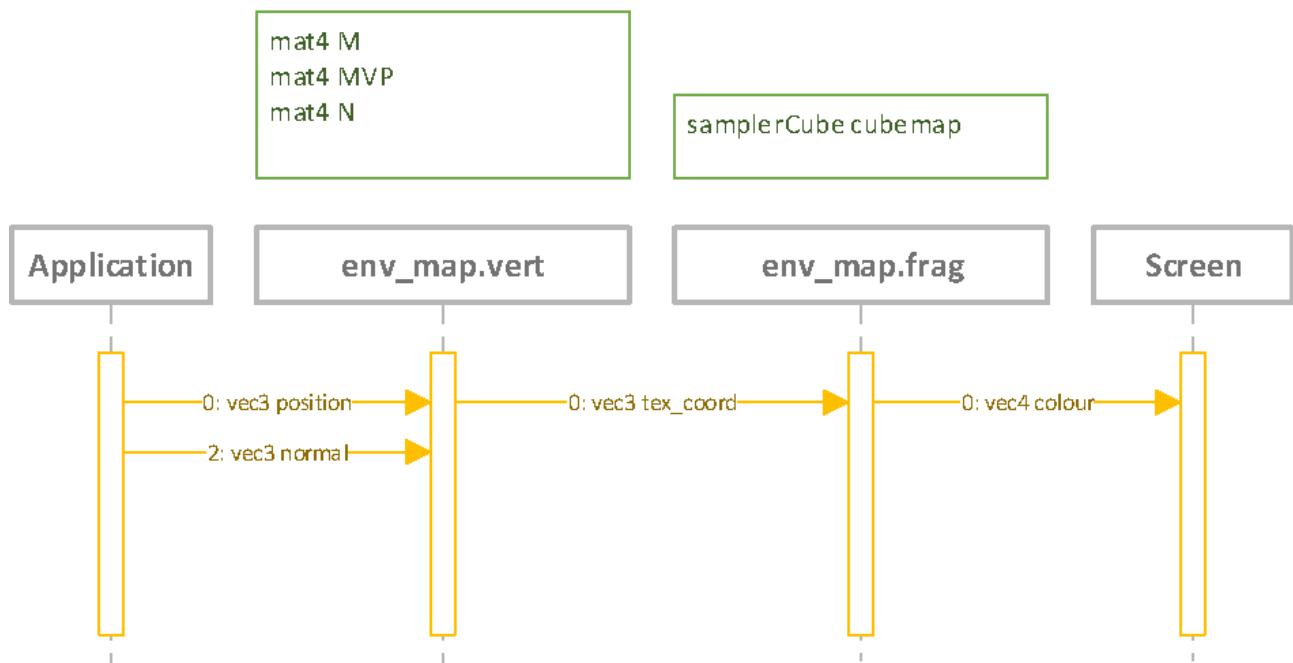


Figure 71.1: Environment Map Shader Structure

The only calculation you need to know about is how to calculate the texture coordinate. This is done by getting the reflection of eye direction and the transformed normal (normalized). Remember the eye direction is just the world space position of the vertex minus the eye position. This is enough of a description for you to complete this lesson - *although you should sketch out why this is the case!*

An example output from this lesson is shown in Figure 71.2.



Figure 71.2: Output from Environment Map Lesson

Lesson 72

Tarnished Object

We can combine our environment map code with a blending approach to create a tarnish effect by using the following:

$$\text{colour} = \text{environment_sample} \times \text{tarnish_sample}$$

This is actually a simple change. You will find the shader files `tarnish.vert` and `tarnish.frag` to complete this. The output from the lesson is shown in Figure 72.1.



Figure 72.1: Output from Tarnished Object Lesson

Lesson 73

Terrain

The last few lessons have been quite simple, so let us move onto a more substantial piece of work - generating terrain. Terrain rendering is something that provides us with some realistic looking images in a rather cheap manner. We will look first at the process we are going to use.

73.1 Generating Terrain Data

To generate terrain we use something called a heightmap. A heightmap is just a texture that contains height data. We convert the colour of the pixels to height data on a plane - the pixel colour represents the y-component of the vertex position. A heightmap is very simple. Figure 73.1 provides an example.

We simply read in our texture data then use it to generate our vertex positions - much like we can do to generate a sphere or our other pieces of geometry. This allows us to render the terrain just as simple geometry. Figure 73.2 provides an example.

There is more we will have to do to generate the normals for the terrain and also provide texturing information. We will do this in later. First we will just generate the geometry.

73.2 Part 1 - Generating Geometry

Your goal in this lesson is to complete the `generate_terrain` function. We will do this in stages. The first part is to actually generate the geometry data. This is done in the following stages:

1. Get texture data
2. Generate position geometry
3. Generate index data for geometry

Let us look at these substages in turn.



Figure 73.1: Heightmap Example

73.2.1 Getting Texture Data

OpenGL provides a simple method to access our texture data - the `glGetTexImage` operation. This operation takes the following form:

```
1 glGetTexImage(target, level, format, type, data) ;
```

These parameters are as follows:

target - based on the type of texture we are using. In this instance we will use `GL_TEXTURE_2D`

level - describes which mipmap level we want to access. This will be 0 in our case

format - How we want the colour data to appear. This will be `GL_RGBA` in our example

type - the number format for our data. We will use `GL_FLOAT`

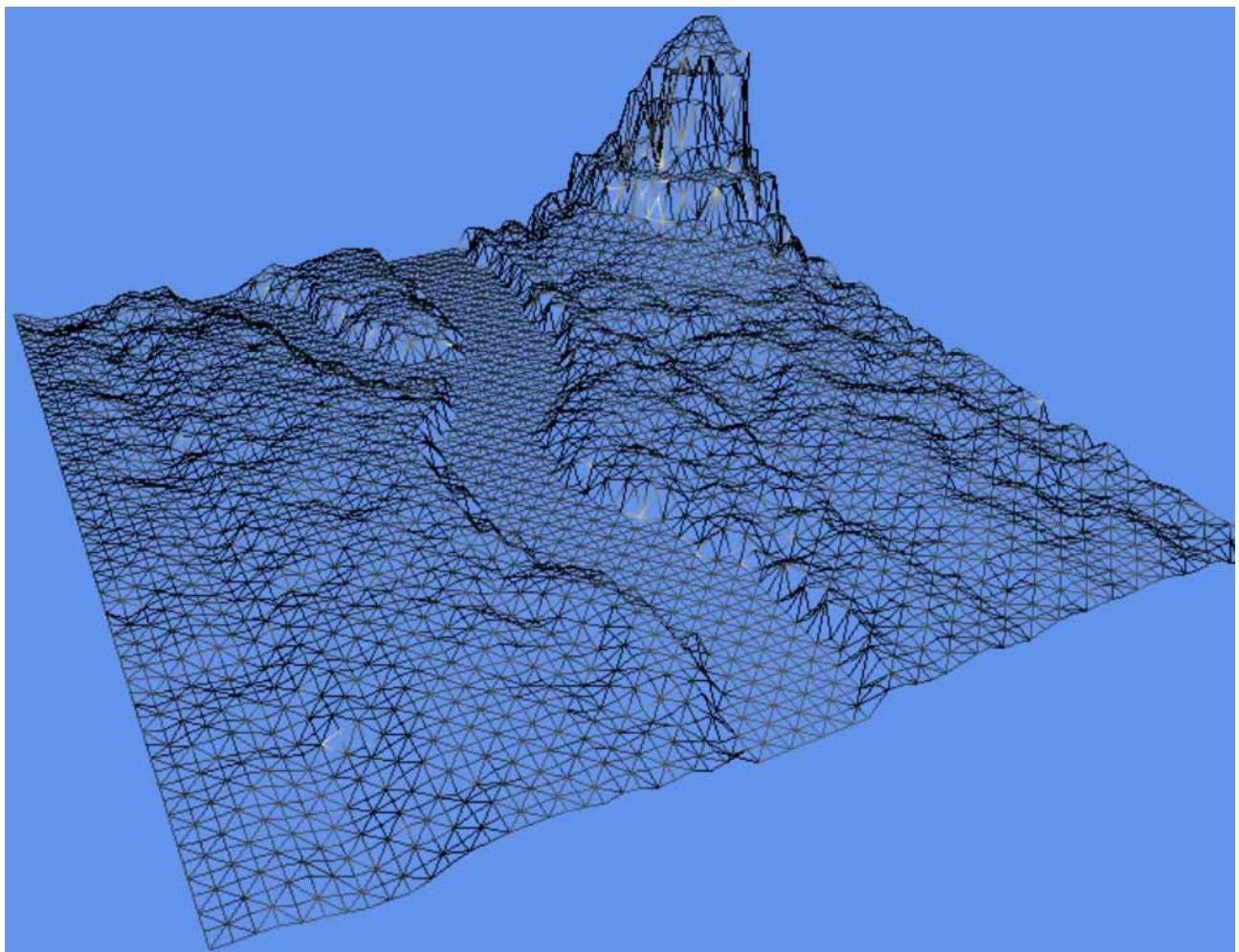


Figure 73.2: Terrain Mesh

data - a pointer to the location to store the captured texture data

The following code snippet shows how we capture the texture data:

```
1 glBindTexture(GL_TEXTURE_2D, id);
2 auto data = new vec4[width * height];
3 glGetTexImage(GL_TEXTURE_2D, 0, GL_RGBA, GL_FLOAT, (void*)data);
```

`glBindTexture` tells OpenGL which texture we want to use. This is what is happening in the renderer when we bind textures. The `id` value is the OpenGL identifier for the texture. You will find that the `texture` data type in the graphics framework provides this for you. The `width` and `height` values are also obtainable from the `texture` object.

73.2.2 Generate Position Geometry

To create our geometry we just need to iterate through each point and create the points accordingly. Algorithm 23 provides the basic algorithm.

All we are doing is creating a plane (take a look at the `geometry_builder` to see the similarity there) but setting the `y` component to the value sampled from the height map.

Algorithm 23 Generating Geometry Data for Terrain

```

1: procedure GENERATE_VERTICES(width, depth, width_point, depth_point,
   height_map_data, height_map_width, height_map_height, height_scale)
2:   for x = 0 → height_map_width do
3:      $P_x = -(width/2) + (width\_point * x)$ 
4:     for z = 0 → height_map_height do
5:        $P_z = -(depth/2) + (depth\_point * z)$ 
6:        $P_y = height\_map\_data[z \times height\_map\_width + x].y \times height\_scale$ 
           ▷ Add P to position buffer

```

73.2.3 Generate Index Data

The other thing we have to do is generate index data to define the triangles. Remember that we used this technique way back at the start when working with a cube. Here, we just need to determine the triangle coordinates and act accordingly. Algorithm 24 provides the basic algorithm.

Algorithm 24 Generating Index Data for Terrain Geometry

```

1: procedure GENERATE_INDICES(height_map_width, height_map_height)
2:   for x = 0 → height_map_width − 1 do
3:     for y = 0 → height_map_height − 1 do
4:        $top\_left \leftarrow (y \times height\_map\_width) + x$ 
5:        $top\_right \leftarrow (y \times height\_map\_width) + x + 1$ 
6:        $bottom\_left \leftarrow ((y + 1) \times height\_map\_width) + x$ 
7:        $bottom\_right \leftarrow ((y + 1) \times height\_map\_height) + x + 1$ 
           ▷ Add triangle 1 to indices
           ▷ top_left, bottom_right, and bottom_left
           ▷ Add triangle 2 to indices
           ▷ top_left, top_right, and bottom_right

```

73.2.4 Completing Part 1

All you need to do is implement the code to get the texture data and implement the necessary algorithms. Two other pieces of code are required. You also need to attach the generated buffers to the geometry (at the bottom of the function) and also clean up the *data* value that you allocated.

Once you have part 1 working you should have an output similar to that shown in Figure 73.3.

73.3 Part 2 - Generating Normals

So our last terrain example was a bit dull - we couldn't actually make out our terrain. Our next stage is to generate the normal information for our terrain. We already know how to do this - we use the cross product.

Our new process will be as follows:



Figure 73.3: Output form Part 1 of Terrain Rendering

1. Get texture data
2. Generate position geometry
3. Generate index data for geometry
4. **Add normal data**

To calculate a normal, we just need to access each triangle (our index data will let us do this) and then use the cross product of the vectors making up the triangle to calculate the surface normal (we have discussed this enough in the lectures that you should understand this).

The trick here is that each vertex belongs to more than one triangle. As such, we need to sum each normal for each triangle that vertex is involved in. Once we have summed the normals in this manner, we then need to normalize each vertex's normal to get the final normal.

Algorithm 25 provides the pseudocode for our normal generation.

73.3.1 Completing Part 2

You need to add the code for the normal generation to our `generate_terrain` procedure (remember to add the buffer). Once this is done, you will also need to change the shader to a lighting one (Phong should do) and create the necessary uniforms. Figure 73.4 provides the expected output.

73.4 Part 3 - Generating Texture Coordinates

The next stage is the simplest one - generating texture coordinates for the terrain. This is pretty easy and the pseudocode is given in Algorithm 26.

Algorithm 25 Generating Normals for Terrain

```

1: procedure CALCULATE_NORMALS(positions, normals, indices, num_indices)
2:   for i = 0 → num_indices/3 do
      idx1 ← indices[i * 3]                                ▷ get indices for next triangle
      idx2 ← indices[i * 3 + 1]
      idx3 ← indices[i * 3 + 2]
      side1 ← positions[idx1] − positions[idx3]          ▷ calculate sides of triangle
      side2 ← positions[idx1] − positions[idx2]
      n ← (side2 × side1)                                ▷ calculate normal based on sides - remember to normalize
      normals[idx1] ← normals[idx1] + n
      normals[idx2] ← normals[idx2] + n
      normals[idx3] ← normals[idx3] + n                                ▷ now normalize each normal
8:   for all normals do
9:     NORMALIZE(normal)

```

Algorithm 26 Generating Texture Coordinates for Terrain

```

1: procedure      CALCULATE_TEX_COORDS(height_map_width,           height_map_height,
                                         width_point, depth_point)
2:   for x = 0 → height_map_width do
3:     for z = 0 → height_map_height do
4:       vx ← width_point × x
5:       vy ← depth_point × z                                ▷ Add v to texture coordinate buffer

```

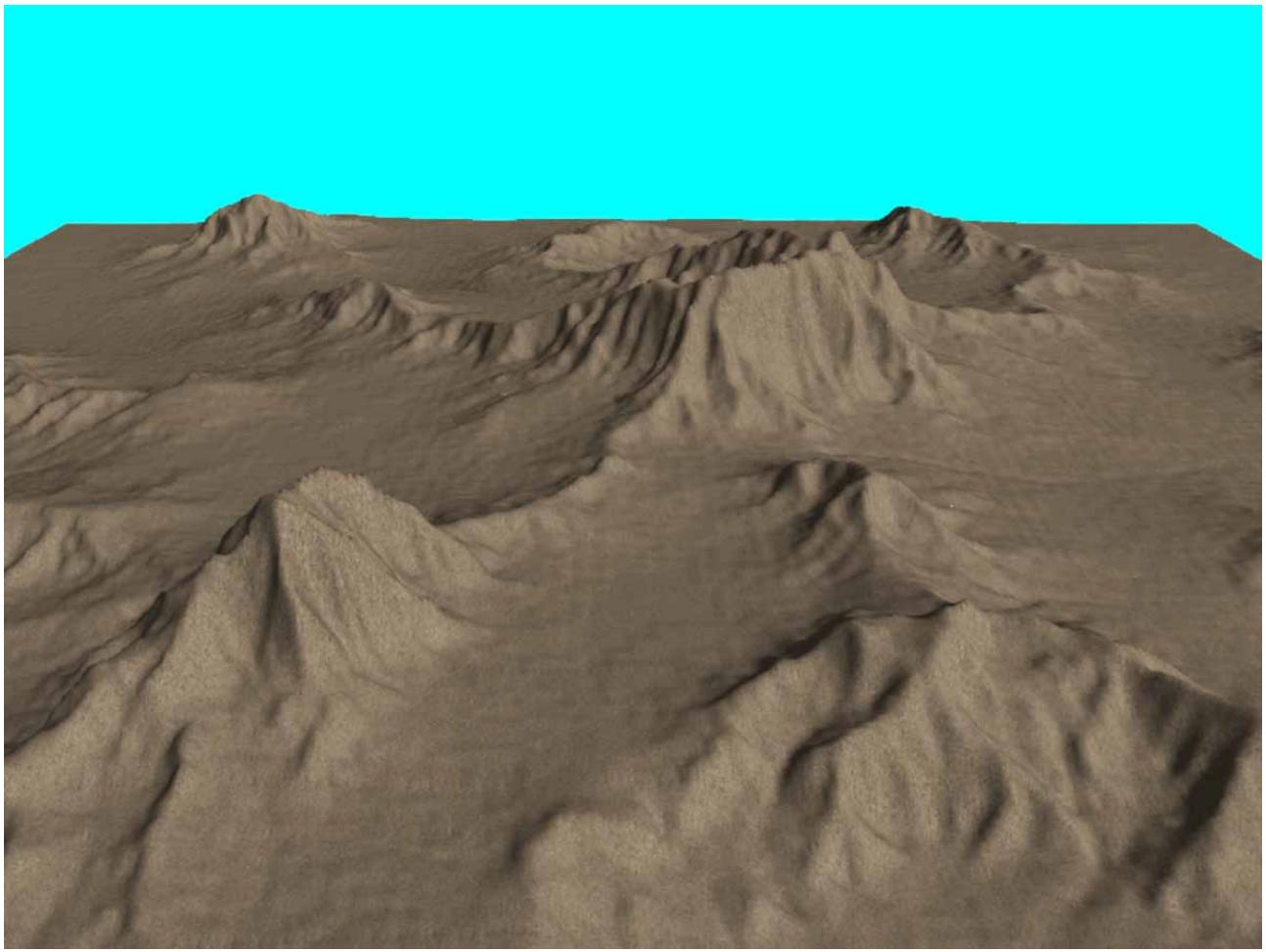


Figure 73.4: **Correctly Lit Terrain**

You will now need to set a texture for the Phong shader to see the result. Figure 73.5 provides an example output.

73.5 Part 4 - Generating Texture Weights

We are now going to return to one of our earlier techniques to create our final terrain render - multi-texturing. To do this we are going to add some new data to our geometry - texture weights. Each piece of geometry can have texture weights associated with its vertices. We will look at how we can generate these texture weights based on the height of the terrain. First of all we will look at how we can create a shader to exploit the texture weights.

73.5.1 Terrain Shader

The terrain shader high level structure is provided in Figure 73.6. It is similar to a standard lighting shader but with the addition of a new vertex parameter - *tex_weight*. This parameter (on channel 11) contains our texture weight data.

The other addition is a new shader part called `weighted_texture`. You will have to complete this shader. All it does is sample each texture, summing these values scaled by the relevant weight. For example, for `tex[0]` you would perform the following calculation:

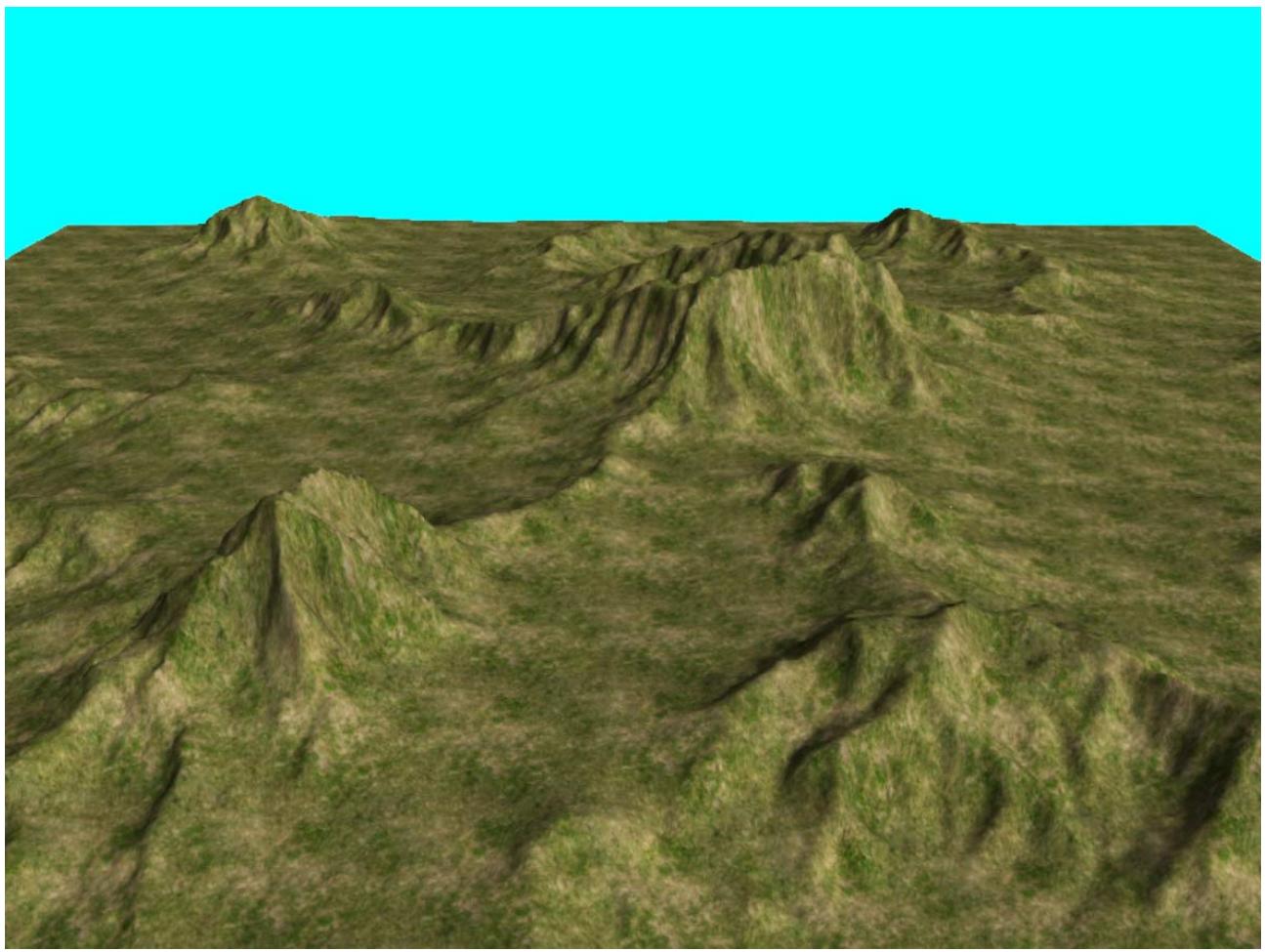


Figure 73.5: **Textured Terrain**

```
1 tex_colour += texture(tex[0], tex_coord) * weights.x;
```

Exercise

Complete the `weighted_texture.frag` and `terrain.frag` shaders now.

73.5.2 Generating Texture Weights

Each vertex requires us to determine the different weights of the 4 textures we are going to use. This requires us to determine some kind of weighting depending on the height of the vertex. The general equation is as follows:

$$weight = 1 - \frac{|y - factor|}{0.25}$$

Remember that $|a|$ signifies the *size* (or *absolute*) of the value. The *factor* value we set depending on the weight we wish to give to that texture at different heights.

The other thing we must do is *clamp* (using the `clamp` function) these values into the range [0, 1]. The pseudocode for the approach we are going to use is given in Algorithm 27.

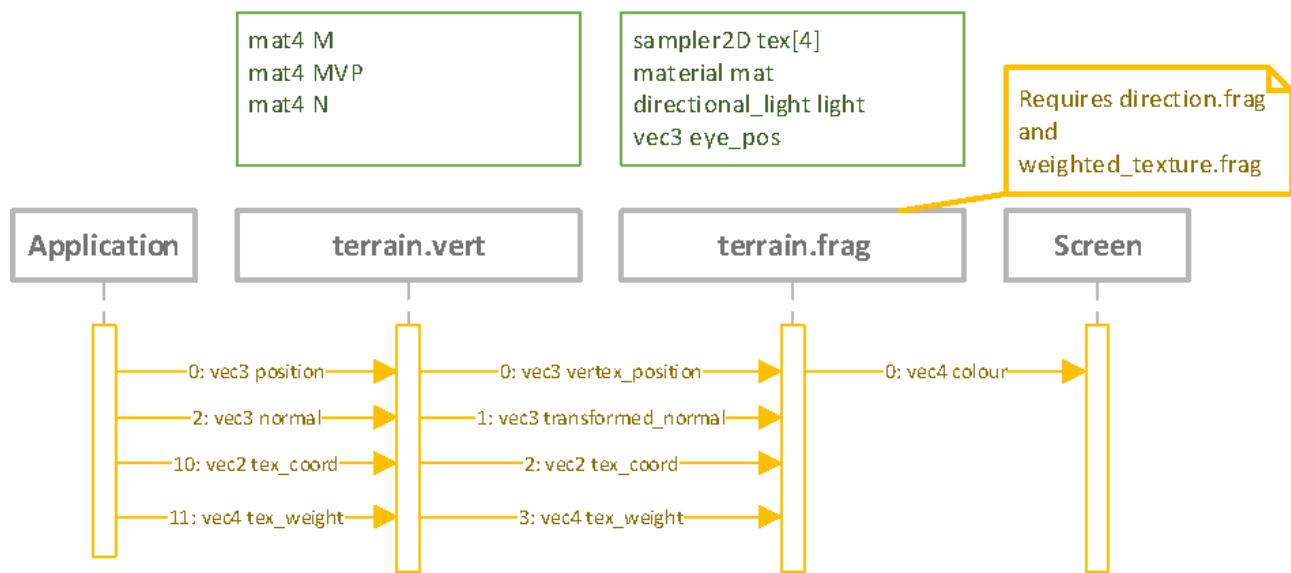


Figure 73.6: Terrain Shader Structure

Algorithm 27 Generating Texture Weights for Terrain

```

1: procedure CALCULATE_TEX_WEIGHTS(height_map_data, height_map_width,  

   height_map_height)
2:   for x = 0 → height_map_width do
3:     for z = 0 → height_map_height do
4:        $w_x \leftarrow 1.0 - |data[\text{height\_map\_width} * z + x].y - 0.0| / 0.25$ 
5:        $w_y \leftarrow 1.0 - |data[\text{height\_map\_width} * z + x].y - 0.15| / 0.25$ 
6:        $w_z \leftarrow 1.0 - |data[\text{height\_map\_width} * z + x].y - 0.5| / 0.25$ 
7:        $w_w \leftarrow 1.0 - |data[\text{height\_map\_width} * z + x].y - 0.9| / 0.25$ 
           ▷ use clamp to ensure all components of w are in range [0, 1]
8:       ...
           ▷ now sum the weights and divide by this sum to make weights total 1.0 (100%)
9:        $total \leftarrow w_x + w_y + w_z + w_w$ 
10:       $w \leftarrow \frac{w}{total}$ 
           ▷ add w to texture weight buffer

```

73.5.3 Completing the Lesson

As before, you will have to update the `generate_terrain` procedure to do the work we require. Then you will have to update the shaders and the set uniforms. For the textures, I used `sand.dds`, `grass.dds`, `rock.dds` and `snow.dds`. Figure 73.7 provides an example final render.

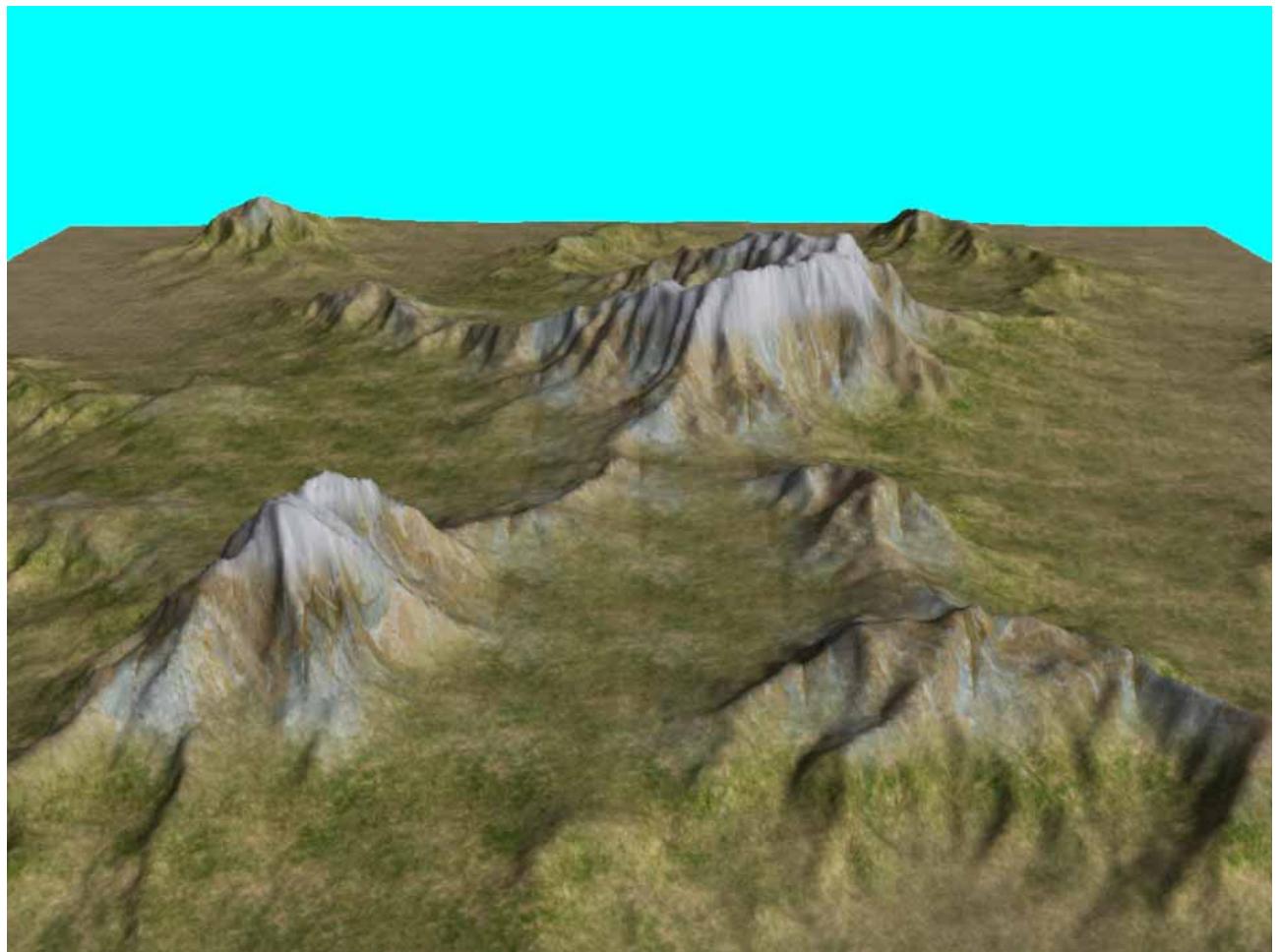


Figure 73.7: Final Terrain Render

Lesson 74

Fog

The final lighting effect we are going to visit is fog. Fog can be done in a number of ways, but the technique we are going to use is very cheap and based on the distance of the object from the camera. As such, we get reasonable looking fog. There are other techniques for developing fog, but these can be computationally expensive.

74.1 Getting Started

There are three different types of fog we will work through. In this lesson we will slowly build up the fragment shader part `fog.frag` (in `resources/shaders/parts`). You will find the starting main file contains the definitions for the three types of fog we will be using:

```
1 #define FOG_LINEAR 0
2 #define FOG_EXP 1
3 #define FOG_EXP2 2
```

The overall structure of our fog shader is provided in Figure 74.1. Again, we are working with a shader that looks similar to a standard lighting shader, but with the addition of a new value being passed from the vertex shader to the fragment shader - `CS_position`. This is the *Camera Space Position* of the vertex, which we calculate using the `MV` (*model-view*) transformation matrix:

$$CS_position = MV \times position$$

In the fragment shader we are only interested in the distance from the camera (the z component) which we calculate from the homogeneous `CS_coordinate`:

$$fog_coord = \left| \frac{CS_position_z}{CS_position_w} \right|$$

The different fog types require different values to operate. These values can be summarised as follows:

`fog_colour` - the colour of the fog

`fog_start` - the distance from the camera where the fog starts

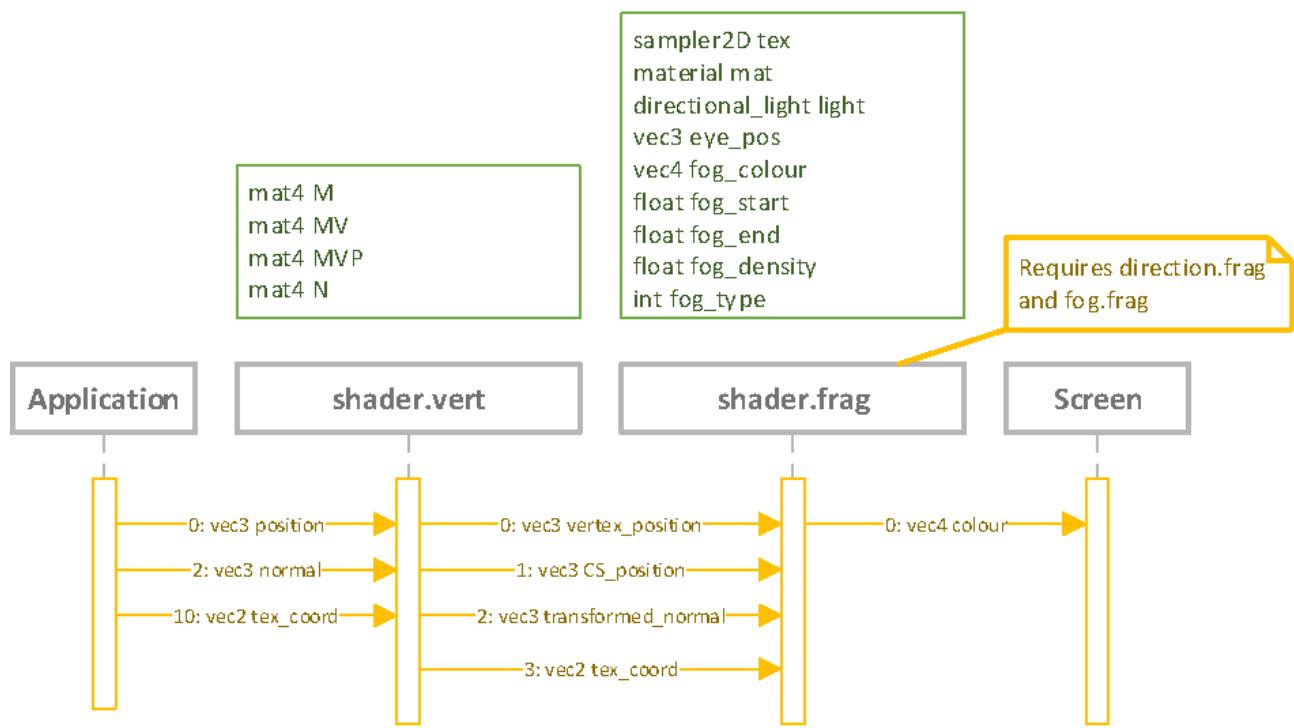


Figure 74.1: Fog Shader Structure

`fog_end` - the distance from the camera where the fog maximises

`fog_density` - the density of the fog

`fog_type` - the fog algorithm being used as defined above.

Let us now look at each of the fog approaches in turn.

74.2 Linear Fog

Linear fog is based purely on the distance of the viewer from the fragment scaled linearly based on the distance the fog extends. The calculation is as follows:

$$fog = \frac{fog_end - fog_coord}{fog_end - fog_start}$$

Your first task is to modify `fog.frag` to compute this. You will need to clamp the result as specified in the fog shader as well. You will also need to complete `shader.vert` and `shader.frag` as specified, and set the uniforms in the main application. You will need to experiment with the start, end and colour values to get the result you want. Figure 74.2 provides an example output. You can change the camera with keys 1 to 4.

74.3 Exponential Fog

Although linear fog looks OK, it doesn't really deal with the fact that fog gets denser the further from an object you are. To handle this we need to use an exponential function using Euler's constant (e). This requires us to use a density value. The function is below:

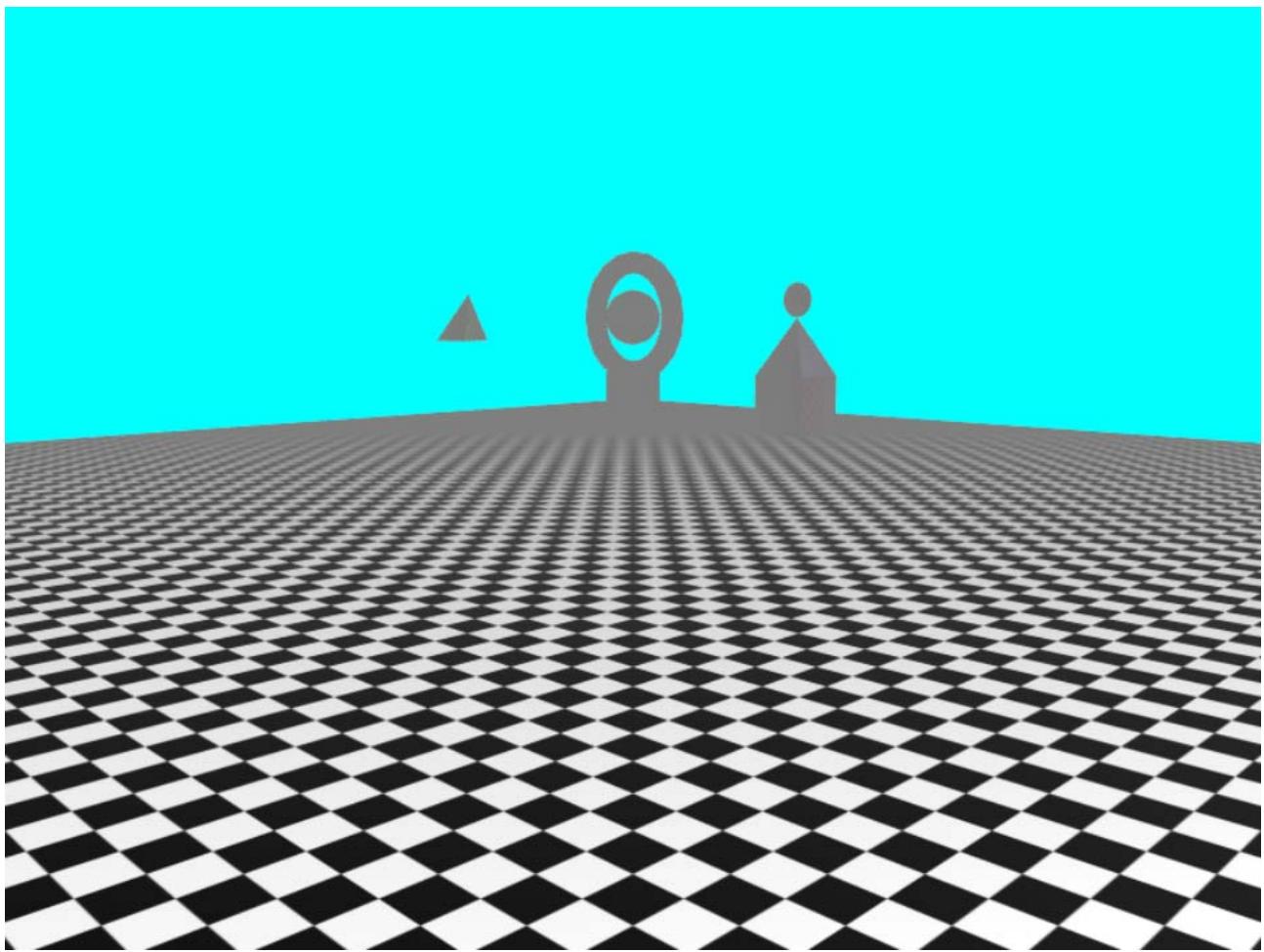


Figure 74.2: Linear Fog

$$fog = e^{-fog_density \times fog_coord}$$

e to a power can be calculated using the `exp` function in GLSL. Your task now is to update the `fog.frag` shader to use this approach, remembering to set the necessary uniforms in the main function. Figure 74.3 provides an example output taken from camera 2.

74.4 Exponential Squared Fog

The one flaw with exponential fog is that it looks too dense nearby. To overcome this we can use an exponential squared function as shown below.

$$fog = e^{-(fog_density \times fog_coord)^2}$$

Add this final fog calculation and update the main application so that you use it. An example output (taken from camera 3) is shown in Figure 74.4.

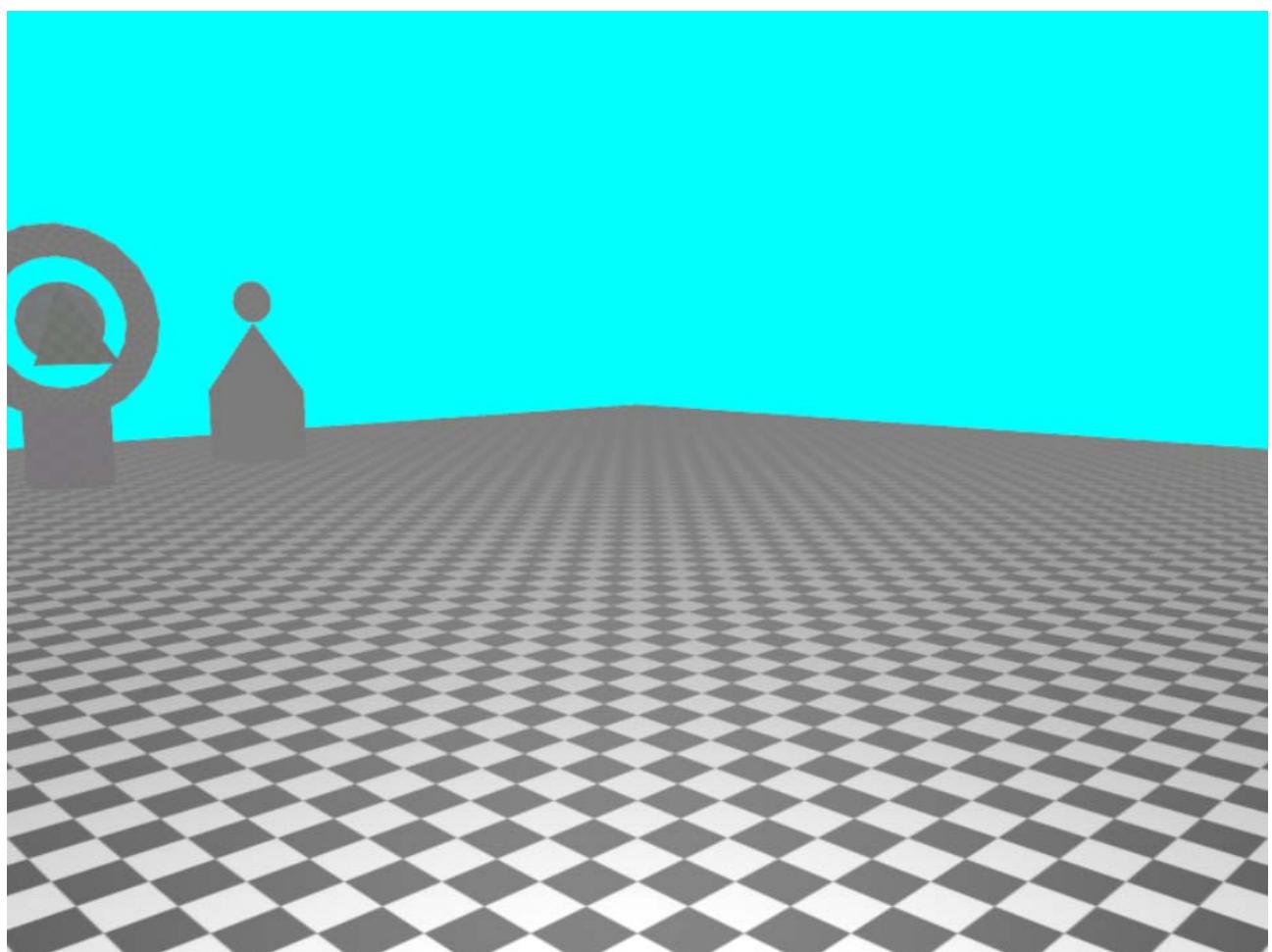


Figure 74.3: Exponential Fog

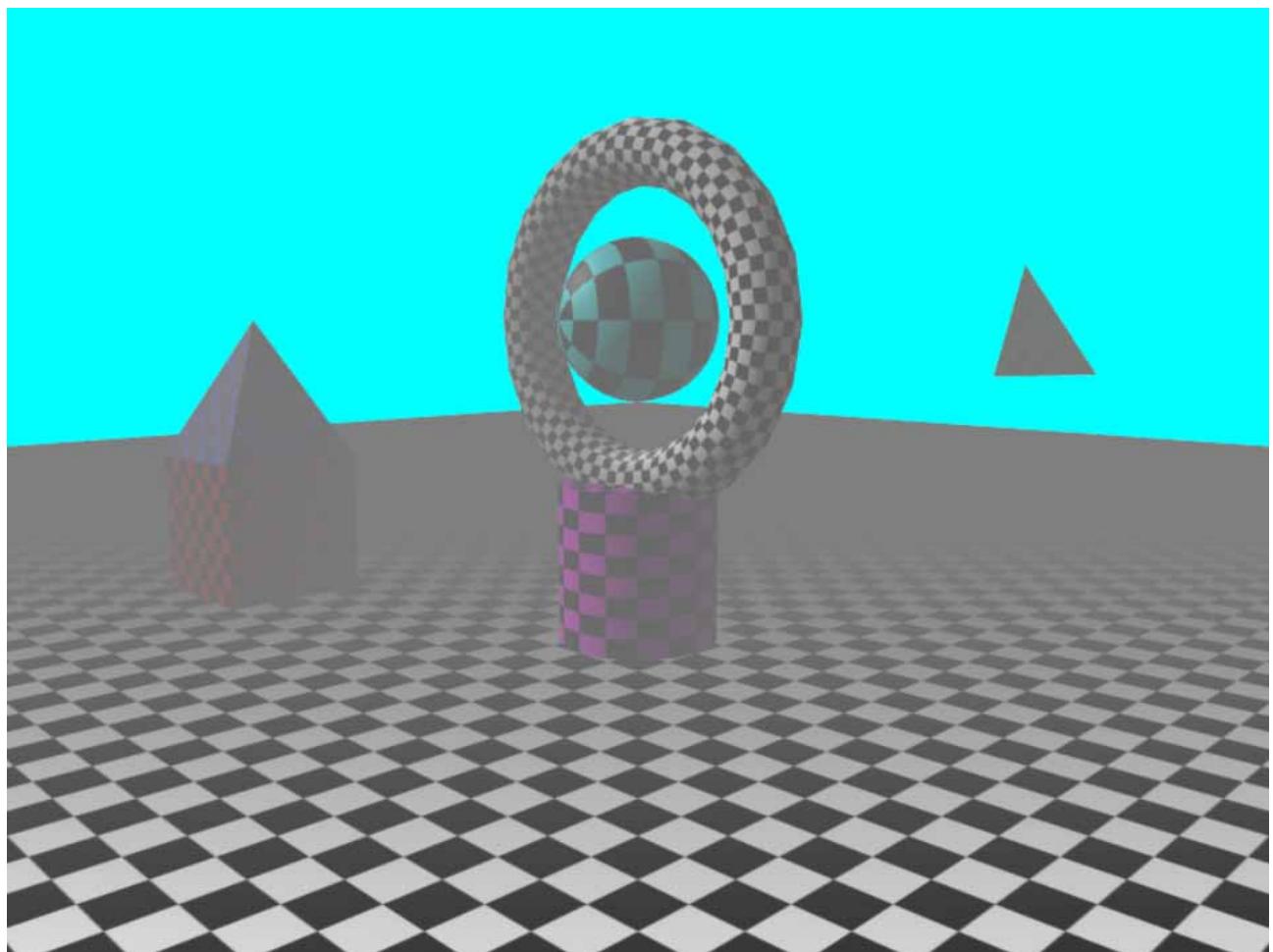


Figure 74.4: Exponential Squared Fog

Lesson 75

BRDF

Part VIII

Geometry Shader

Lesson 76

Geometry Shader

We are now going to introduce a new stage to our shader pipeline - *the geometry shader*. The geometry shader is a relatively new stage for shader programming (it wasn't supported in the PlayStation 3 and had a sort of workaround in the Xbox 360), and therefore there is less example shaders out there. However, it is a very powerful stage to the pipeline, allowing us to add and manipulate geometry in ways that the vertex shader does not.

76.1 Available Inputs to the Geometry Shader

To start off, let us look at the inputs that the geometry shader takes as these are a little different to the vertex and fragment shaders we have been working with. At present, we have worked on the idea that we define the incoming data and the outgoing data to all our stages in the pipeline. The only exception to this is in the vertex shader where we define the position of the vertex in screen space by setting `gl_Position`. There are other values we could set here as well, but have had no need.

The geometry shader works using these position values. The `gl_Position` value we set in the vertex shader is available in the geometry shader. How this is presented depends on the type of geometry we are rendering. Let us look at these types first.

76.1.1 Input Types

A GLSL geometry shader takes in geometry data based on the type of geometry primitive being rendered. It has five different geometry types it can handle. These are:

`points` - a single vertex

`lines` - two vertices making up a line

`lines_adjacency` - the two vertices making up a line, and the two vertices connected to these two

`triangles` - three vertices making up a triangle

`triangles_adjacency` - three vertices making up a triangle, and the three vertices connected to these vertices to create four triangles

The adjacency information is a little difficult to explain in words. However, if you look at Figure 76.1 it should give you an idea of what we mean.

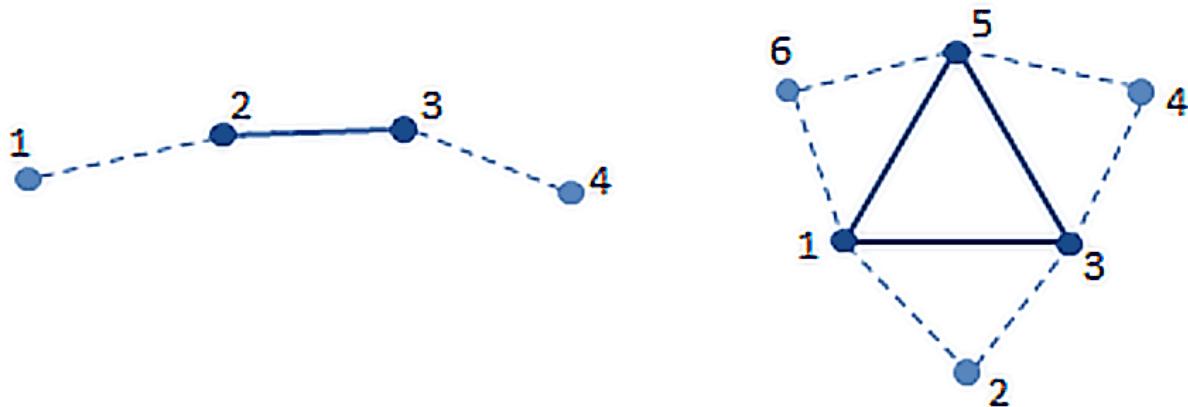


Figure 76.1: **Line and Triangle Adjacency**

Knowing what type of data is coming into the geometry shader is one thing but knowing how to access it is another. Let us look at what vertex data is available to us.

76.1.2 Available Variables

An OpenGL geometry shader has access to the following incoming variables:

```

1 in gl_PerVertex
2 {
3     vec4 gl_Position;
4     float gl_PointSize;
5     float gl_ClipDistance [];
6 } gl_in[];
7
8 in int glPrimitiveIDIn;
9 in int glInvocationID;
```

Notice that `gl_in` is an array of data. The number of vertices available on `gl_in` depends on the type of geometry being rendered. For example, for `points` only one vertex is available, two for `lines`, three for `triangles`, etc. We can then access individual vertices values using standard array access (e.g. `gl_in[2].gl_Position`).

76.1.3 Defining Incoming Values

So now we know how our incoming data is structured. The next step is stating this in our geometry shader. This is actually the easiest part. We use the `layout` keyword as follows:

```
1 layout (points) in;
```

`points` can be replaced with the type of geometry you are interested in.

76.1.4 Incoming Values from Previous Stage (Vertex Shader)

So what about the other values we output from the vertex shader? Can we access them? Yes, these are accessible in much the same way as we would expect. The only difference is that the incoming values are sent in as arrays to match the `gl_in` value. For example, to declare incoming normal data we would write the something similar to the following:

```
1 layout (location = 0) in vec3 normal[];
```

And that's it really for working with incoming data. Uniforms can be used as normal.

76.2 Available Outputs from the Geometry Shader

Let us now look at how we output data from the geometry shader. Again, we will do this in stages.

76.2.1 Output Types

A GLSL geometry shader can output only three different types of data. These are:

`points` - single vertices

`line_strip` - vertices defining a strip of lines

`triangle_strip` - vertices defining a strip of triangles

You can output different data than you have input to the geometry shader! For example, you can take in point data and output triangles. This is actually how we do particle effects on with the geometry shader, as we will see in an upcoming lesson.

The geometry shader itself isn't limited to how many values it outputs given input data. For example, we can take in one vertex and output as twenty. This allows us to duplicate geometry, as we shall see at the end of this lesson.

76.2.2 Output Variables

As with input, the geometry shader has a number of output values you can set. These are defined below:

```
1 out gl_PerVertex
2 {
3     vec4 gl_Position;
4     float gl_PointSize;
5     float gl_ClipDistance[];
6 };
7
8 out int gl_PrimitiveID;
9 out int gl_Layer;
10 out int gl_ViewportIndex;
```

Notice that we are not using an array of values this time - *we emit one piece of vertex information at a time!* The main thing we need to worry about is setting `gl_Position` as we did in the vertex shader. This value is the final screen space position of our vertex. We won't look at the other values in the module.

76.2.3 Defining Outgoing Values

To define outgoing data we follow a similar method to the one we used for incoming values:

```
1 layout (triangle_strip, max_vertices = 3) out;
```

We tell the geometry shader the type of output. We should also tell the geometry shader the number of vertices we expect to output using `max_vertices`.

76.2.4 Outgoing Values to the Next Stage (Fragment Shader)

We can also output values to the fragment shader in the standard manner - we do not use array syntax here. For example, to say we are outputting a normal we simply add:

```
1 layout (location = 0) out vec3 normal;
```

76.2.5 Outputting Vertex Data

The final piece of the puzzle is how we actually output a vertex from the geometry shader. This is done using the following function:

`EmitVertex()`

We have to set any values we require for the vertex (at least `gl_Position`) and then call `EmitVertex()` to output it. For example, we might have the following:

```
1 gl_Position = MVP * gl_in[0].gl_Position;
2 normal_out = N * normal_in;
3 tex_coord = vec2(1.0, 1.0);
4 EmitVertex();
```

Once we call `EmitVertex` the data is sent down the graphics pipeline and we can set the next vertex. The other call we need to perform is as follows:

`EndPrimitive()`

This states that we have output enough vertex information for an output primitive to be completed. This allows us to separate out our primitives into reasonable chunks.

76.3 Example Geometry Shader

OK, let us look at what a geometry shader looks like. This is below:

```

1 #version 440
2
3 // Define any uniforms required. For example:
4 uniform mat4 MVP;
5
6 // Layout of incoming data
7 layout (points) in;
8 // Layout of outgoing data
9 layout (triangle_strip, max_vertices = 3) out;
10
11 // Define any other incoming values from vertex shader. For example←
12 :←
12 layout (location = 0) in vec3 normal[];
13
14 layout (location = 0) out vec3 normal_out;
15
16 void main()
17 {
18     // .. perform operations
19 }
```

We have explained these parts individually, but we will build an example application very soon.

76.4 Loading a Geometry Shader

The final piece of the puzzle is how we load a geometry shader in the graphics framework. This is done in the same manner as the other shaders, except we use `GL_GEOMETRY_SHADER` as the type.

```
1 eff.add_shader("filename", GL_GEOMETRY_SHADER);
```

76.5 Example Shader - Copy Geometry

OK time for an example (and the opportunity to write part of a geometry shader). We are going to write a shader that copies our incoming geometry, placing the copies to the left and right of the original using an `offset` value. The structure of our copy shader is shown in Figure 76.2.

Notice that the vertex shader doesn't output anything - it doesn't even have the `MVP` transformation matrix. That is because the transformation will be undertaken in the geometry shader. This is so we can output the original (with no offset) and the other two pieces of geometry (with the offset) easily. The `shader.geom` code outlines everything you need - you just need to output the geometry two times more.

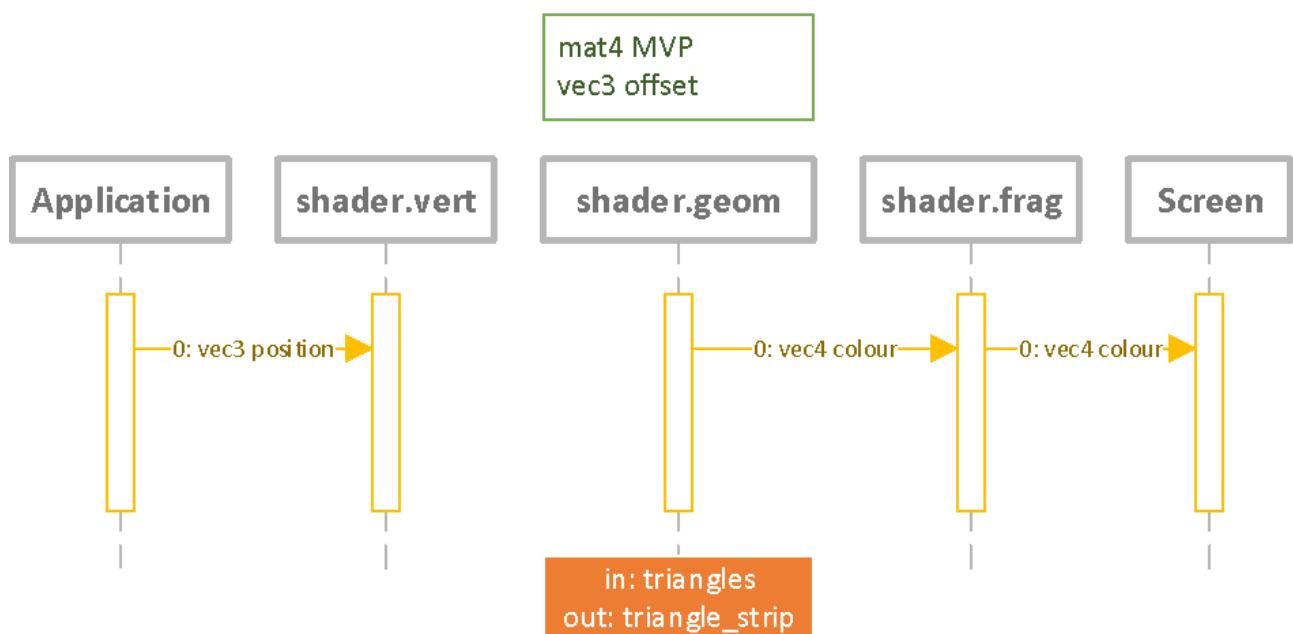


Figure 76.2: Copy Geometry Shader

The expected output from this application is shown in Figure 76.3.

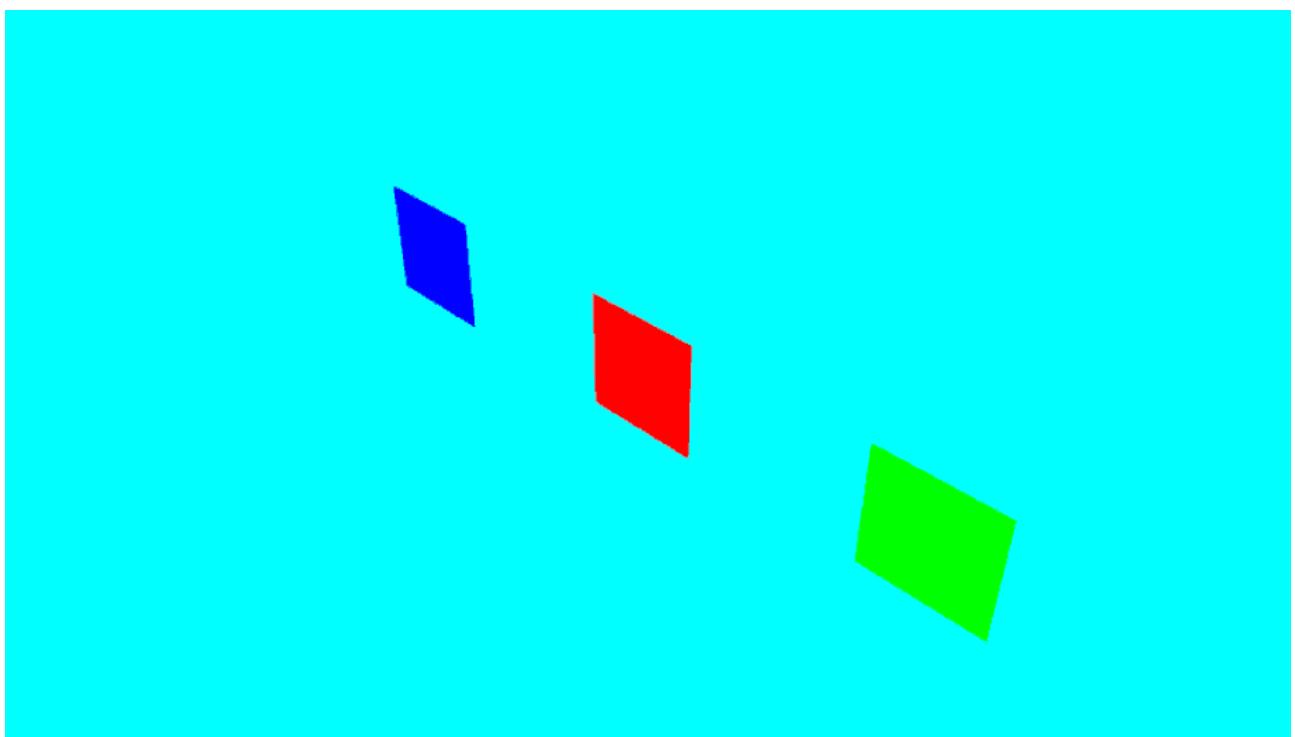


Figure 76.3: Copy Geometry Shader Output

Lesson 77

Exploding Shape

Our next geometry shader will manipulate our incoming geometry and move it outwards in an exploding fashion. This is actually quite easy to do. The basic equation is as follows:

$$p = p + (n * \alpha)$$

Where α is some form of explosion factor. We have to calculate the face normal (n) within the geometry shader - go back to the terrain lesson to remind yourself how we do this!

You will have to do perform the transformation for each of the incoming vertices. The structure of the shader is provided in Figure 77.1.

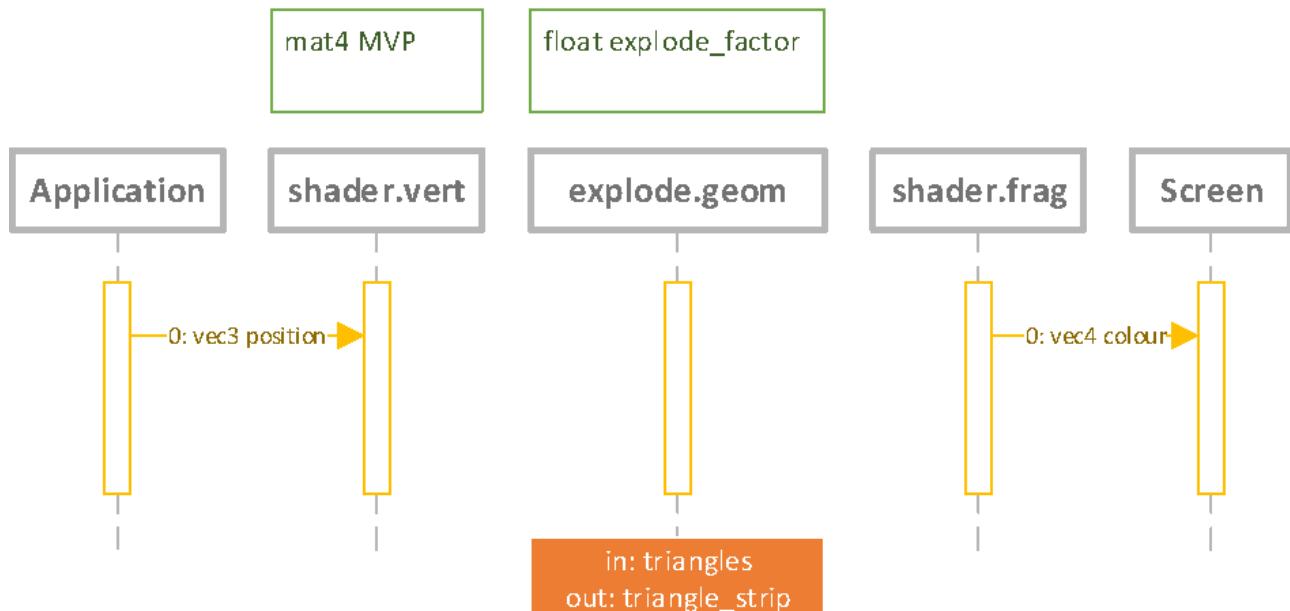


Figure 77.1: Explode Shader Structure

An example output from this lesson is shown in Figure 77.2.

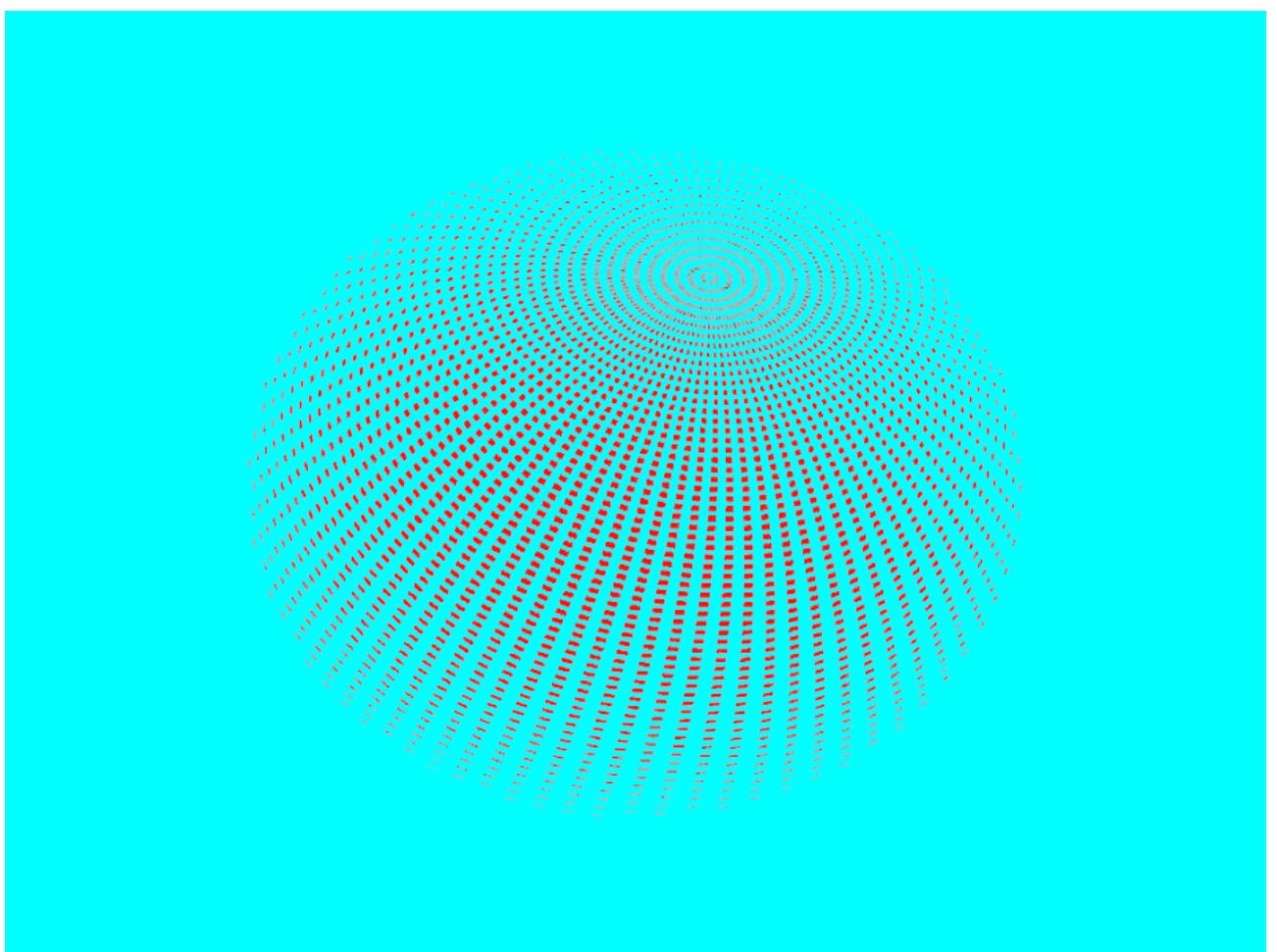


Figure 77.2: Output from Exploding Shade Shader

Lesson 78

Showing Normals for Debugging

So far our work with the geometry shader has looked at replicating geometry or manipulating existing geometry. But what about changing the type of geometry we are rendering. In this lesson we're going to use the geometry to render our normals for our shapes. This is actually fairly easy.

78.1 Normal Rendering Geometry Shader

For this geometry shader we will require normal data to be sent to our geometry shader. We will also need the position data (we get this from `gl_Position` as before). The work we will do will take place in the geometry shader only, and therefore we will require the MVP transformation matrix there as well. The shader structure is illustrated in Figure 78.1.

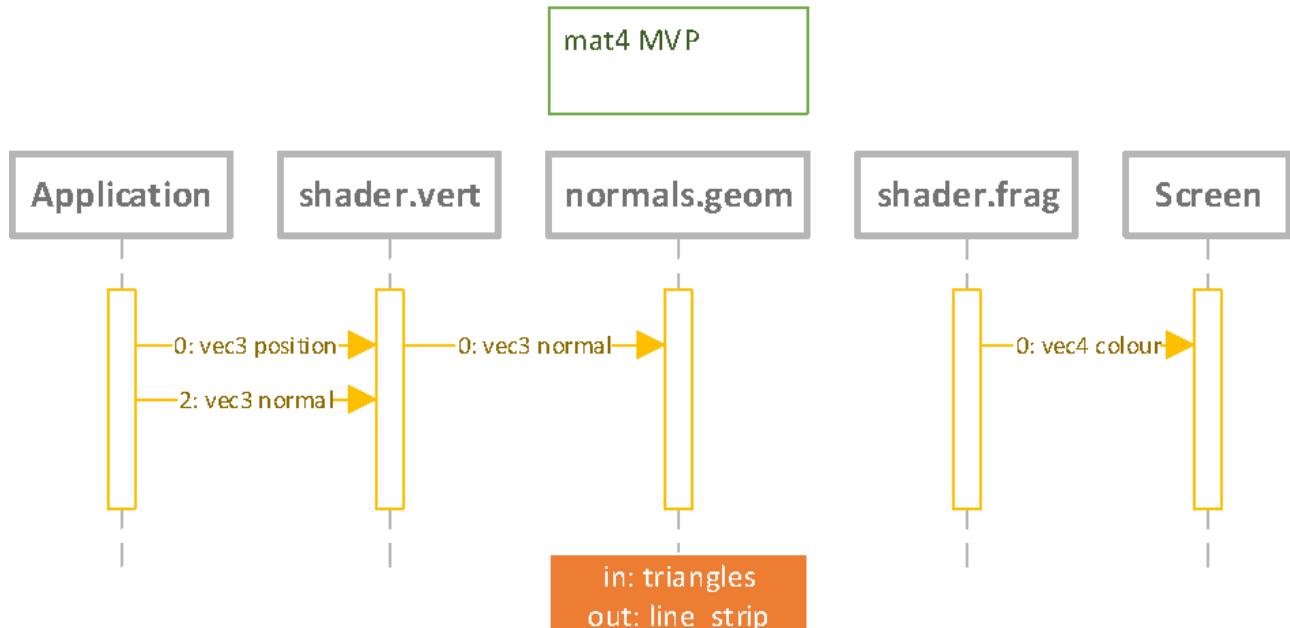


Figure 78.1: Normal Rendering Shader Structure

78.2 Completing the Lesson

Our approach takes in the vertex information as points and outputs lines. This makes sense. We have an incoming positional vertex, and from this we want to output a line representing the normal. The calculation we need to perform is:

$$P_0 = MVP \times P$$

$$P_1 = MVP \times (P + N)$$

Where P_0 is the start of the line and P_1 the end. This is how you have to complete the shader. You will also need to update the main application to load the shader and to render all the geometry using this shader before rendering it normally (yes we render the geometry twice). The output from this lesson is shown in Figure 78.2.

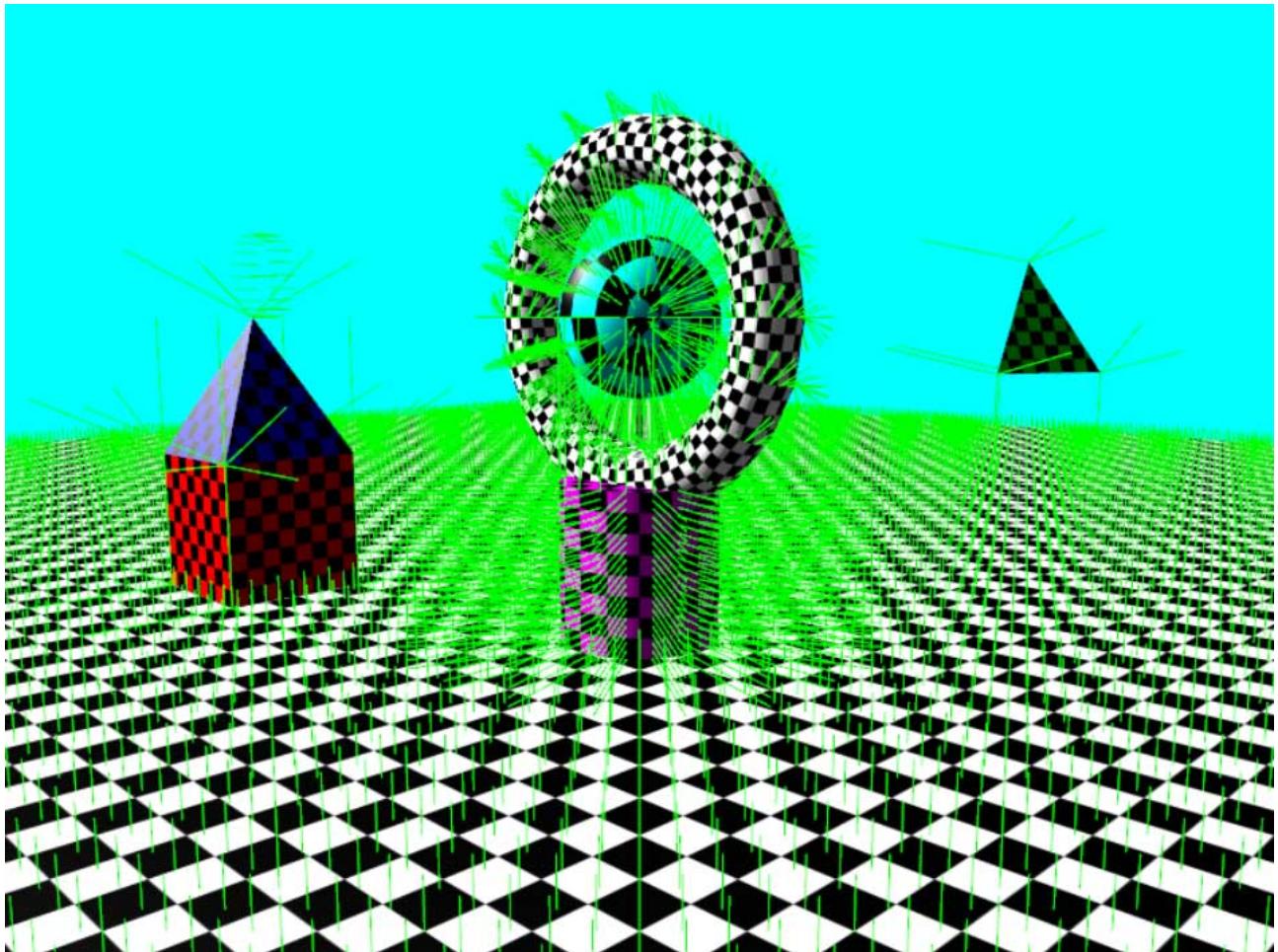


Figure 78.2: Output from Normals Application

Lesson 79

Billboarding

Now let us do something practical with our geometry shader - *billboarding*. Billboarding is a technique where we render a texture so that it always faces the viewer no matter which direction they face.

79.1 What is Billboarding?

Billboarding was a common technique back in the dawn of 3D games. Original first person shooters such as Doom (see Figure 79.1) used billboarding for the characters that moved through the world. This provided a cheap method for adding moving objects in a game without getting into 3D animation.

Billboarding is still used in modern games, but has been relegated to the rendering of items in the background. Typically grass and smoke effects (via particles) use billboarding for their effects. In this lesson we will look at simple billboarding and latter combine this with a particle effect to create smoke.

79.2 How Billboarding Works with the Geometry Shader

Our approach involves us taking in point data to the geometry shader and converting this into triangles (two to create a quad) to texture. This trick here is how we ensure that our billboard faces the screen. This is actually quite easy.

Consider a single vertex (a point) and how we transform the position:

$$\begin{aligned} \textit{position}_{\textit{world}} &= M \times \textit{position}_{\textit{model}} \\ \textit{position}_{\textit{camera}} &= V \times \textit{position}_{\textit{world}} \\ \textit{position}_{\textit{screen}} &= S \times \textit{position}_{\textit{camera}} \end{aligned}$$

Stage 3 of this process (converting the camera position to the screen position) is what we want to modify. At this stage, our points position is relative to the camera. Therefore, if we convert this point into a quad our quad will face the camera. In other words, our four vertices become the following:



Figure 79.1: Billboarding in Doom

$$\begin{aligned}
 v_0 &= position_{camera} + (0.5, 0.5) \\
 v_1 &= position_{camera} + (-0.5, 0.5) \\
 v_2 &= position_{camera} + (-0.5, -0.5) \\
 v_3 &= position_{camera} + (0.5, -0.5)
 \end{aligned}$$

We then transform these positions into screen space using the P matrix. The overall pipeline of how our billboarding works is shown in Figure 79.2.

79.3 Billboarding Shader Structure

Our billboarding shader is illustrated in Figure 79.3. Again notice that we are not sending anything from the vertex shader, but we are adding texture coordinates from the geometry shader.

79.4 Completing the Lesson

All the work for this lesson is in the geometry shader. You will have to complete this to get the expected output. This is shown in Figure 79.4.

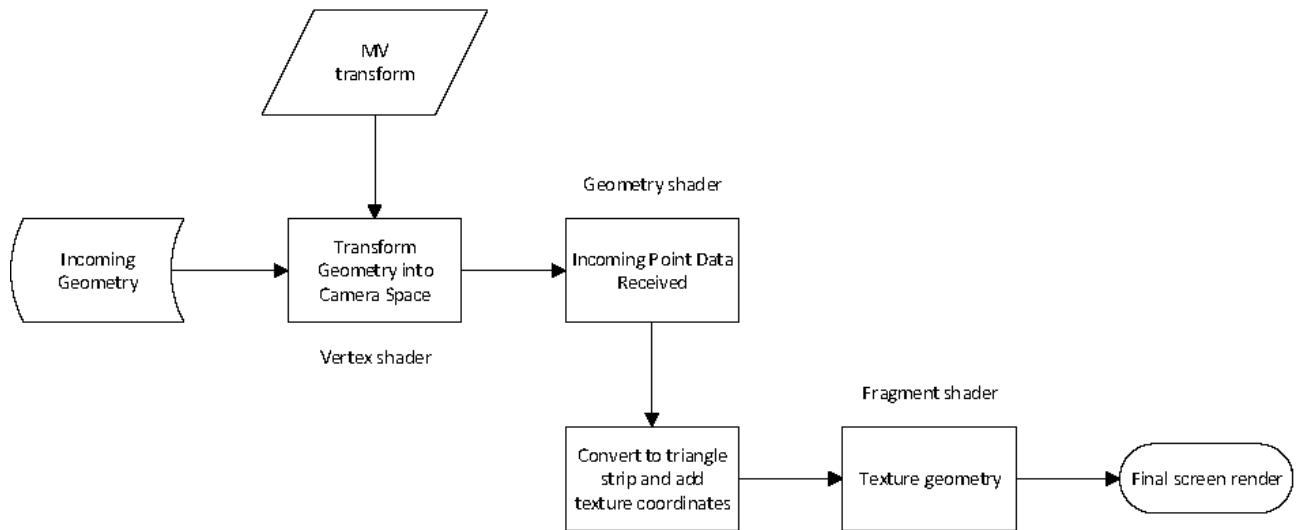


Figure 79.2: Billboarding Process

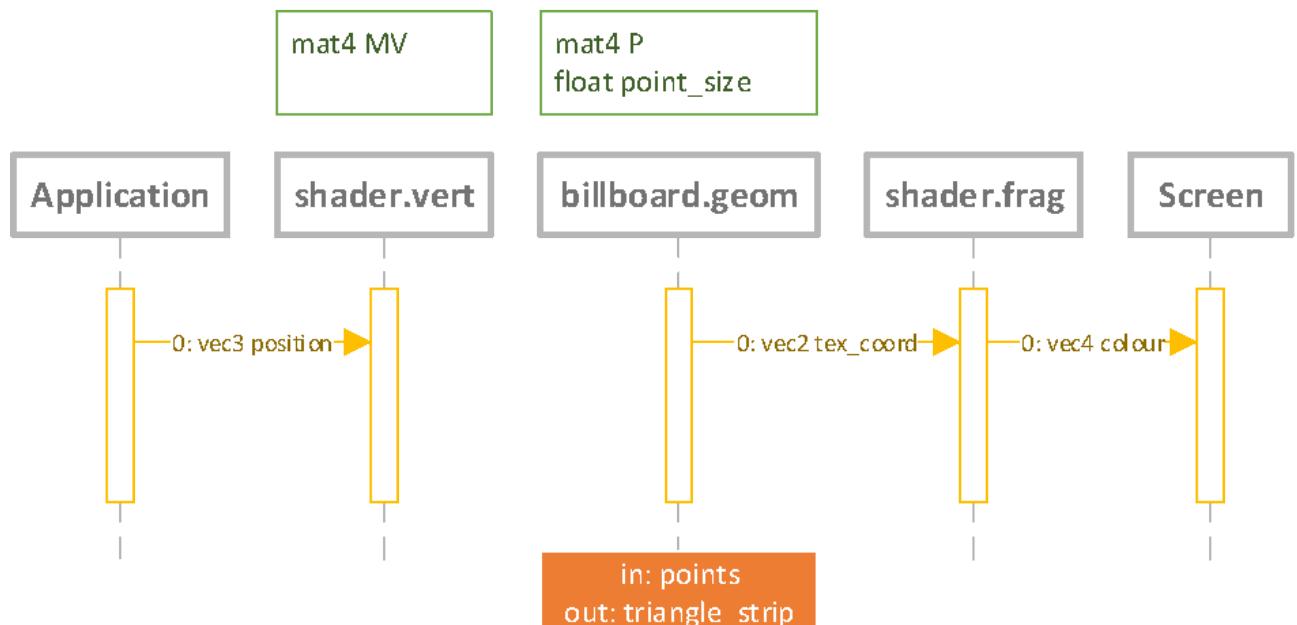


Figure 79.3: Billboarding Shader Structure



Figure 79.4: Billboarding Lesson Output

Lesson 80

Particle Effects

We will now do something that is outside the realm of just using the geometry shader for rendering effects. One of the other areas that the GPU is used for in games is for physics simulation. One particular technique that does relate to rendering is the use of particle effects. We will look at what a particle is at the end of this lesson. First, we will look at the capability of the GPU that allows us to work with particles in this way - *transform feedbacks*.

80.1 Transform Feedback

A transform feedback is a technique where we have capture our geometry data *before* it is rendered. This means that we can manipulate our incoming geometry, save the manipulated data to a buffer, then use this buffer in our render. Figure 80.1 illustrates a more detailed view of the graphics pipeline including the transform feedback.

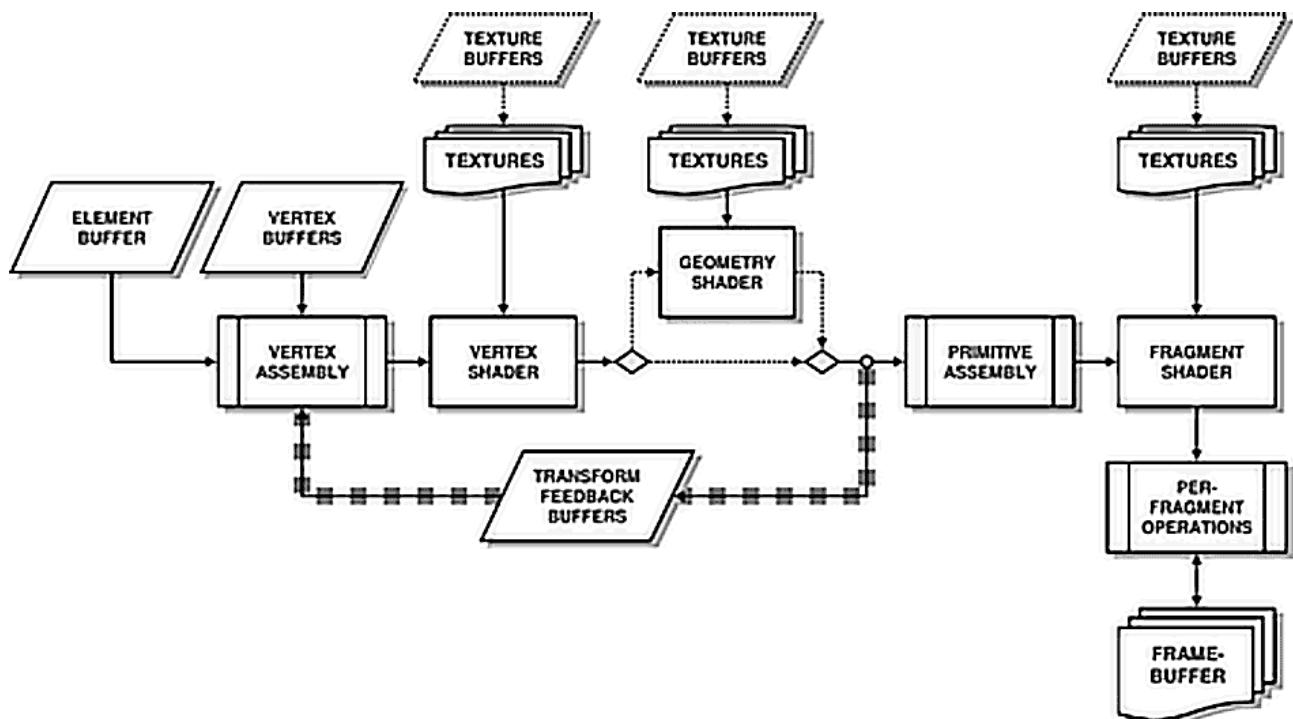


Figure 80.1: Transform Feedback Loop

80.2 The Shader

Our particle shader takes the structure shown in Figure 80.2. Notice that we are not outputting to the screen now but to a transform feedback.

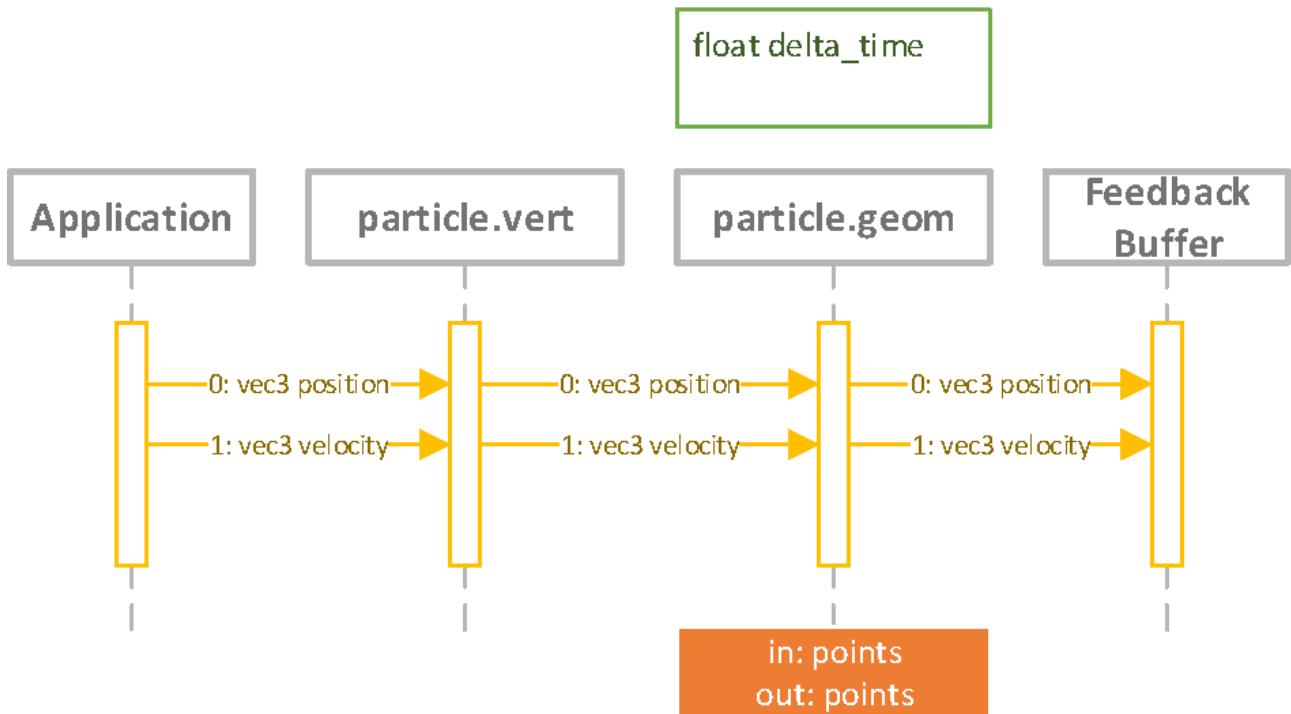


Figure 80.2: Particle Shader Structure

80.3 Particles

Our application will use the simplest form of physics we can have - *a particle simulation*. A particle is just a physical object that has a position and velocity. We could also add mass, but we will keep it very simple.

Our particle takes the following form in C++:

```

1 struct particle
2 {
3     vec3 position = vec3(0, 0, 0);
4     vec3 velocity = vec3(0, 0, 0);
5 };

```

To update the particle, we have to change the position. This is done using the following calculation:

$$p' = p + (v * t)$$

Where p' is the new position, p the starting position, v the velocity and t the time past. This is all the geometry shader really needs to do, but we will add a further restriction to our application. The position should never go higher than 5.

80.4 Creating Transform Buffers

In this lesson we are going to go far more into the internals of OpenGL and let you see some of the calls that the graphics framework takes care of for you. In particular, we are going to look at buffer creation and rendering. This is what happens in the `mesh` and `geometry` objects you have been using.

For our transform feedback we perform the following actions:

1. Create the buffers with OpenGL (allocated on the GPU)
2. Set the data for the buffers
3. Describe the data for the transform feedback in the shader

Let us look at each of these in turn.

80.4.1 Allocating the Buffers with OpenGL

OpenGL has a number of calls relating to the allocation of memory with the GPU. These calls typically work with texture and geometry data, but there are other types of buffers we can create.

For this lesson we need to create two transform feedbacks and the buffers to go along with them. We need two buffers as we need to be passing one through the GPU while storing to another. This is a typical double buffering approach that the GPU does for performing the standard renders.

The two calls we are interested in for the first stage are as follows:

```

1 // Generate transform feedbacks
2 glGenTransformFeedbacks(GLuint number, GLuint *ids);
3
4 // Generate buffers containing data for transform feedbacks
5 glGenBuffers(GLuint number, GLuint *ids);

```

`glGenTransformFeedbacks` generates a number of transform feedbacks with OpenGL. We need to tell OpenGL how many we want to create (in this case 2) and provide a pointer to some `unsigned int (GLuint)` values that OpenGL will store the IDs of these buffers to (the ID is how OpenGL recognises the buffer).

80.4.2 Setting the Buffer Data

The next stage setting the data in the buffers and linking the buffers to the relevant transform feedback. This requires us to perform the following four stages:

1. Bind the transform feedback we want to use in OpenGL. This tells OpenGL we want to use this transform feedback at the moment.
2. Bind the relevant buffer we want to use with OpenGL. This tells OpenGL we want to use this buffer at the moment.
3. Set the buffer data. This requires the size (in bytes) of the data we are using, and a pointer to the data we want to copy to the buffer on the GPU.

4. Link the buffer to the transform feedback using `glBindBufferBase`

These calls are illustrated below:

```

1 // Bind the transform feedback
2 glBindTransformFeedback(GL_TRANSFORM_FEEDBACK, GLuint id);
3 // Bind the buffer used with the transform feedback
4 glBindBuffer(GL_ARRAY_BUFFER, GLuint id);
5 // Set data of the buffer
6 glBufferData(GL_ARRAY_BUFFER, GLuint size_of_data, ptr_to_data, ←
    GL_DYNAMIC_DRAW);
7 // Bind buffer to transform feedback
8 glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 0, GLuint id);

```

80.4.3 Describing the Buffer

The next stage requires us to link the shader we are using to the transform feedback. This means telling OpenGL the data we are expecting out from the particular program. This is actually simple and a bit complicated at the same time. Therefore, the majority of the code required for this example is provided below.

```

1 // Define names of output data to capture from geometry shader
2 const GLchar* attrib_names[2] =
3 {
4     "position_out",
5     "velocity_out"
6 };
7 // Set these on the feedback
8 glTransformFeedbackVaryings(GLuint program_id, 2, attrib_names, ←
    GL_INTERLEAVED_ATTRIBUTES);
9
10 // Have to relink the program after this
11 glLinkProgram(GLuint program_id);

```

The two names (`position_out` and `velocity_out`) are the output values from our geometry shader stage. We use the call to `glTransformFeedbackVaryings` to tell OpenGL these names (the `attrib_names` value) and the number of names (2). We also have to tell OpenGL which shader program we are referring to (you can get this via `eff.get_program()`). Finally we have to re-link the program to set this information.

80.5 Performing the Update

OK, stage 1 down - we have created our data. Now what? Well next we need to perform the transform feedback loop, which we do in the `update` method (not the `render`).

Again, we have a number of stages to go through:

1. Disable rasterization (we are not outputting beyond the geometry shader)
2. Describe the data we are going to send with OpenGL
3. Perform the feedback

4. End the update

Let us look at these in turn.

80.5.1 Disabling Rendering

Disabling rendering in OpenGL is simply a case of enabling `GL_RASTERIZER_DISCARD` as shown below.

```
1 glEnable(GL_RASTERIZER_DISCARD);
```

80.5.2 Setting up the Data Streams

OK, now we get into more areas that the graphics framework hides from you. To describe our incoming data we need to enable a collection of data streams, and then tell OpenGL how the data will be sent to them. These streams are the incoming locations we have been using in our vertex shaders. The following code shows how we go about this.

```
1 // Bind the buffers for use
2 glBindBuffer(GL_ARRAY_BUFFER, GLuint id);
3 glBindTransformFeedback(GL_TRANSFORM_FEEDBACK, GLuint id);
4
5 // Define how our data looks like to the shader
6 glEnableVertexAttribArray(GLuint idx);
7 // ... other data values
8 glVertexAttribPointer(GLuint idx, GLuint num_values, type, GL_FALSE, ←
    GLuint size_of_total_data_type, (const GLvoid*)offset);
```

`glEnableVertexAttribArray` enables one of the streams. We provide it with the stream we want to enable (0, 1, 2, etc.). These are the locations we use in our vertex shader.

`glVertexAttribPointer` tells OpenGL what data to send down this stream. We use the same `idx` value, then tell OpenGL the number of values (`num_values`) and the `type` of values (e.g. `GL_FLOAT` for floating point values). We also need to tell OpenGL the total size of a set of data we are using in the stream, and an offset (in bytes) for the value we are streaming.

OK, that was complicated. Let us look at how we set this up for our `position` data in our particles:

```
1 glEnableVertexAttribArray(0); // Position location
2 glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(particle), (←
    const GLvoid*)0); // Position description
```

We are using location 0. The type for `position` is `vec3` so we have 3 values of type `GL_FLOAT`. We need the size of the values in the data (these are of type `particle` so we need `sizeof(particle)`). The offset (in bytes) from the start of the particle is 0.

You will also need to define the `velocity` value. This is in location 0 and has an offset of 12 bytes (the size of the `vec3` representing the `position` value).

80.5.3 Performing the Feedback

OK now are ready to perform the feedback loop. This looks as follows:

```

1 // Perform the transform feedback
2 glBeginTransformFeedback(GL_POINTS);
3 // Check if first frame
4 if (first_frame)
5 {
6     glDrawArrays(GL_POINTS, 0, GLuint num_primitives);
7     first_frame = false;
8 }
9 // Otherwise perform the draw
10 else
11     glDrawTransformFeedback(GL_POINTS, GLuint id);
12
13 // End the transform feedback
14 glEndTransformFeedback();

```

We start the feedback (`glBeginTransformFeedback`) and then determine if we are performing the first frame. If we are, we need to perform a single render once to output the data into the first transform feedback. This is done using `glDrawArrays` with the number of primitives we have (*hint - how many particles do we have?*).

If not in the first frame we use `glDrawTransformFeedback` to send the particles through the geometry shader from one buffer to the other.

At the end we call `glEndTransformFeedback`.

80.5.4 Ending the Update

At the end of the update we have to unwind the settings we did at the start. This means disabling the vertex information and disabling the discarding of rasterization. This is as follows:

```

1 // Disable used locations
2 glDisableVertexAttribArray(GLuint idx);
3 // ... others
4
5 // Switch on rendering again
6 glEnable(GL_RASTERIZER_DISCARD);

```

80.6 Rendering the Transform Feedback with OpenGL

Finally, we can look at how we render the data. Actually, we did this already in the update - we are now just going to perform the render properly. The outline of the stages is below.

```

1 // Bind the buffer for rendering
2 glBindBuffer(GL_ARRAY_BUFFER, GLuint id);
3
4 // Describe the data

```

```
5 glEnableVertexAttribArray(GLuint idx);
6 glVertexAttribPointer(GLuint idx, GLuint num_values, type, GL_FALSE, ←
   GLuint size_of_data_struct, 0);
7 // Perform the render by drawing the transform feedback
8 glDrawTransformFeedback(GL_POINTS, GLuint id);
9 // Disable vertex attribute array
10 glDisableVertexAttribArray(GLuint);
```

80.7 Completing the Lesson

You have the necessary information to complete the main application file. You will also need to complete the shader. An example output is given in Figure 80.3.

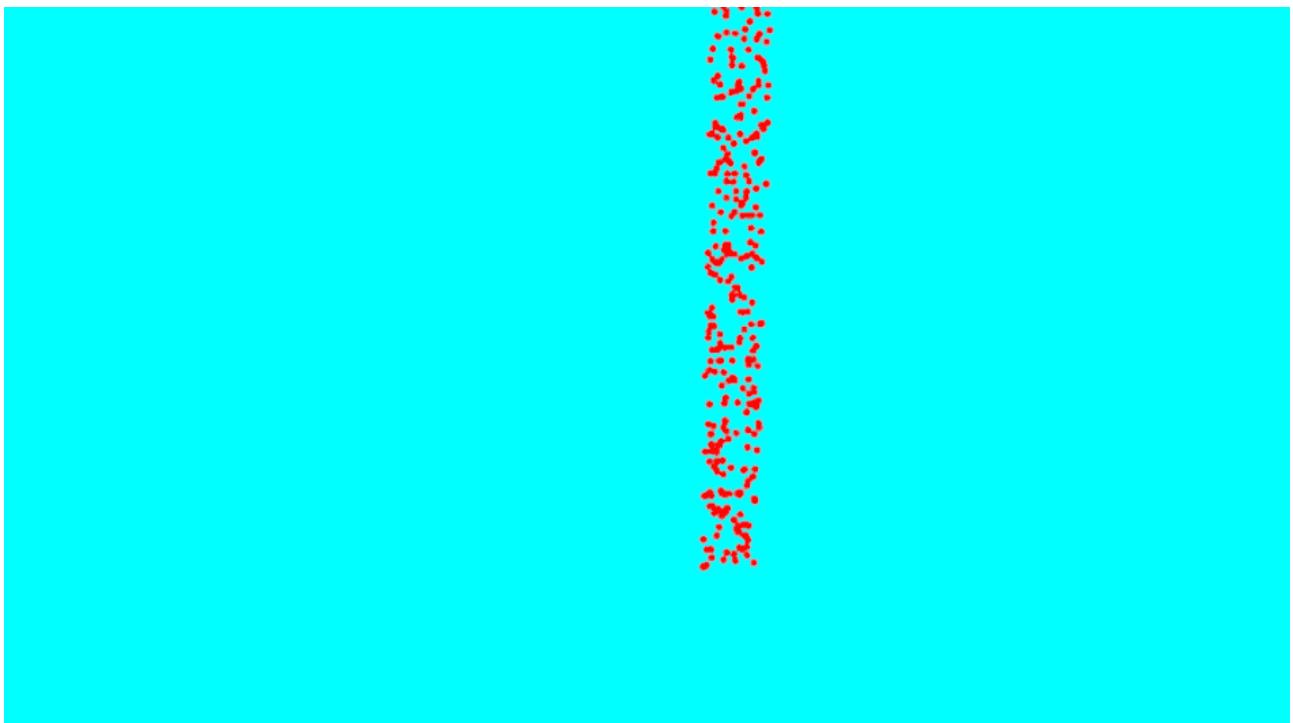


Figure 80.3: Output from Particles Lesson

Lesson 81

Smoke

Our final work with the geometry shader requires you to combine the ideas from the billboarding lesson with those from the particle effect lesson. You have all the information you need for this. The example output is given in Figure 81.1.

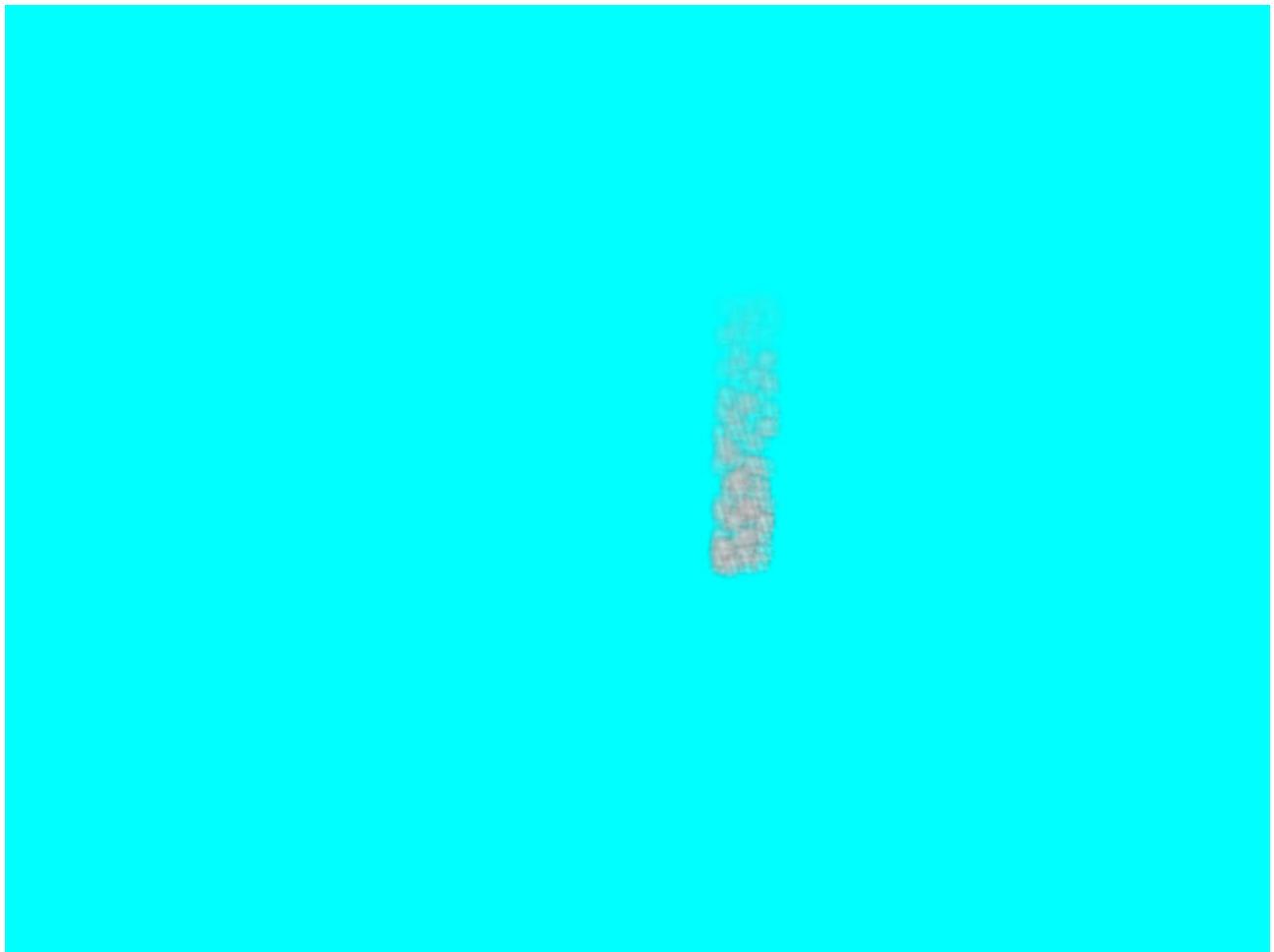


Figure 81.1: Output form Smoke Application

Part IX

Post-processing

Lesson 82

Post-processing

We have now taken ourselves to the point where we can build, texture and light our scenes in a number of manners. There is little more to learn from the point of view of basic rendering techniques. We are now going to move onto a technique called post-processing. Post-processing is a technique where we capture a render in a texture and then use the captured texture in a future part of our render. We call each capture a render pass.

We have actually already been doing this using the shadowing technique previously. We are just going to expand the depth buffer capture to also capture render textures.

Lesson 83

Frame Buffers

We are going to start our examination of post-processing by using the fundamental element - the ability to capture frames. We do this using a frame buffer. The graphics framework actually has quite a bit of the underlying functionality built in. We will look at how you can use it.

83.1 One Last Look at the Pipeline

We previously re-examined the graphics pipeline when discussing particle effects and how we can capture geometry data using a transform feedback. A frame buffer is another method of capturing render information, but this time as a texture that stores the frame information. If we look at the graphics pipeline again (Figure 83.1) we can see that the frame buffer comes at the end.

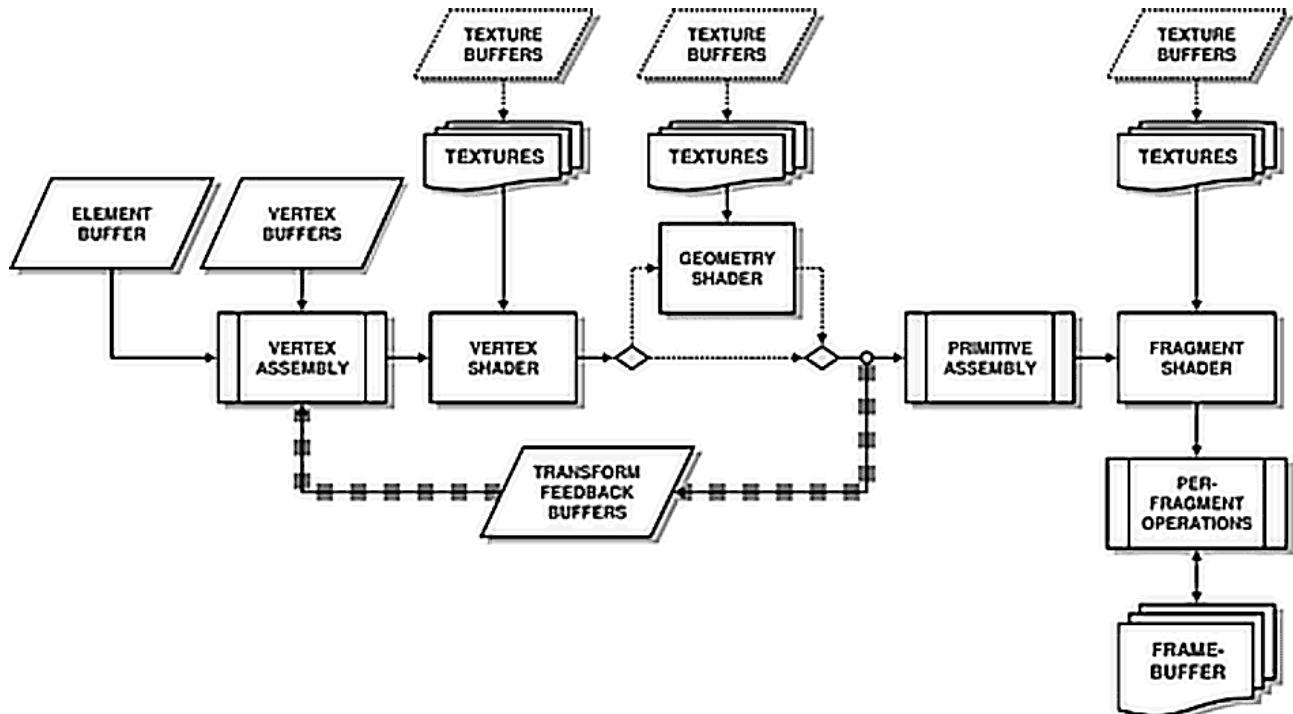


Figure 83.1: Graphics Pipeline

83.2 Creating a frame_buffer

The graphics framework comes with a `frame_buffer` object. We create it as we did the `shadow_map`:

```
frame_buffer(width, height)
```

In practice, `width` and `height` should be the screen width and height.

The use of the `frame_buffer` is the same as the `shadow_map`. You should revisit that lesson to remind yourself of the general idea.

83.3 Working with Frame Buffers

When working with a frame buffer we follow the following stages:

1. Bind the frame buffer
2. Render the scene
3. Bind back to the screen
4. Render the captured texture through another shader effect

We can in fact do this a number of times (rendering to a frame buffer is often called a render pass), reusing the resulting information in various ways.

83.4 Completing the Lesson

We are simply going to capture our render and use it to texture a cube. This uses just the basic texturing shader we made a long time ago. All you need to do is use the `frame_buffer` correctly.

Figure 83.2 illustrates the output. You can see the scene rendered to each face of the box. It is difficult to make out but visible enough for illustration.

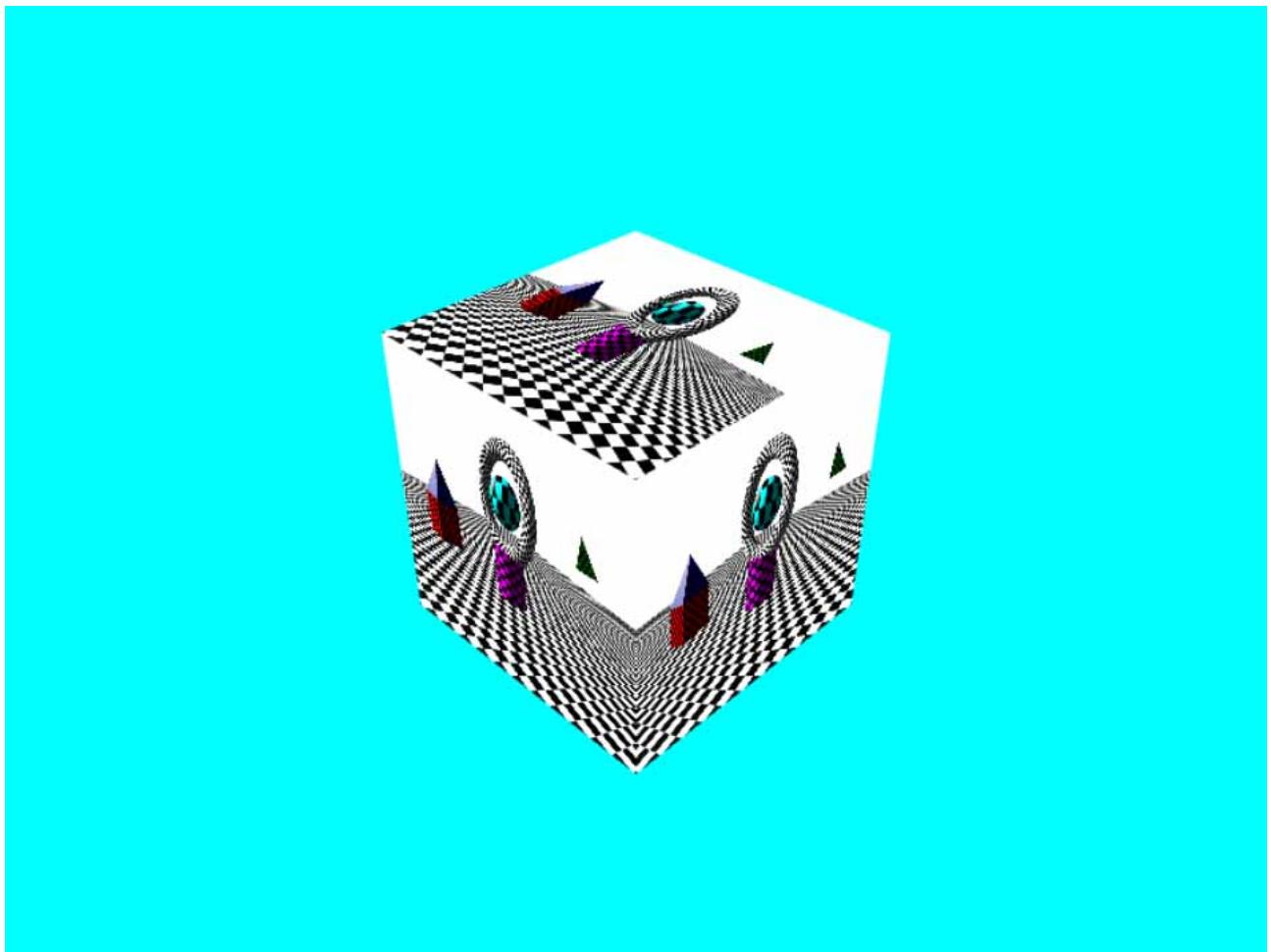


Figure 83.2: Using a Frame Buffer to Capture a Render

Lesson 84

Displaying to Screen

The true goal of post processing is to capture a render pass in a texture, and then redraw that texture so that we can apply a particular texture effect to it. This requires us to redraw the texture to the screen in a manner so that it fills the screen. To do this, we will need to apply a flat piece of geometry that will fill the screen (a screen space quad). We can do this trivially.

84.1 Things you Should Remember - Screen Dimensions

Remember that the screen has coordinates in the range $(-1, -1, -1)$ to $(1, 1, 1)$. We can therefore define a quad with these dimensions, the z component being 0 for all vertices. We will also need the relevant texture coordinates.

We can still use the basic texturing shader. However, now we don't need to transform the position data into screen space - they are already there. Therefore our *MVP* matrix is just the *identity* matrix.

With these simple ideas you should be able to complete the next lesson. It's output is shown in Figure 84.1.

Yes this seems like a lot of work to just render the same information, but now we have access to the frame and can do some post-process effects on it.

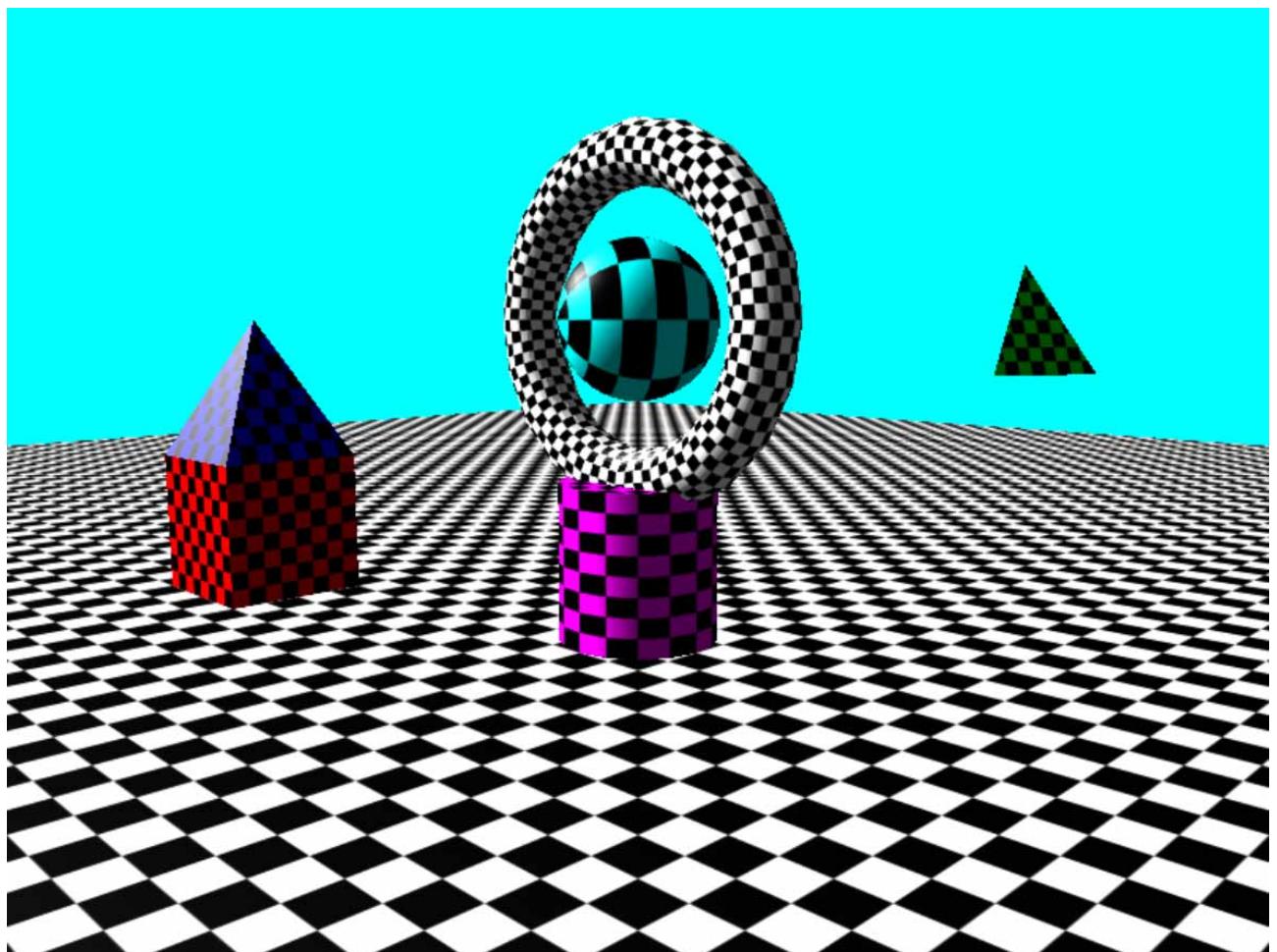


Figure 84.1: Displaying a Frame Buffer to the Screen

Lesson 85

Greyscale Post-process

Our first post-process will convert the render to greyscale. This is easy - we just need to work out the intensity of an individual pixel. To do this, we need use the following equation:

$$i = 0.299c_r + 0.587c_g + 0.184c_b$$

Where c is the vector representing the colour value. We can simplify this down to a single calculation:

$$i = f \cdot c$$

where

$$f = (0.299, 0.587, 0.184)$$

This is all the information you need to complete this post-process. Figure 85.1 provides an example output.

85.1 Exercises

1. Can you make a general purpose solution that allows us to use a particular colour filter provided by the main application rather than a fixed colour?
2. Now simplify further, and use a matrix multiplication to so that we can just multiply the incoming colour directly.
3. Sepia colour can be achieved by adding (0.314, 0.169, -0.090) to the outgoing greyscale colour. Can you implement a sepia shader and also integrate this into your generic solution?

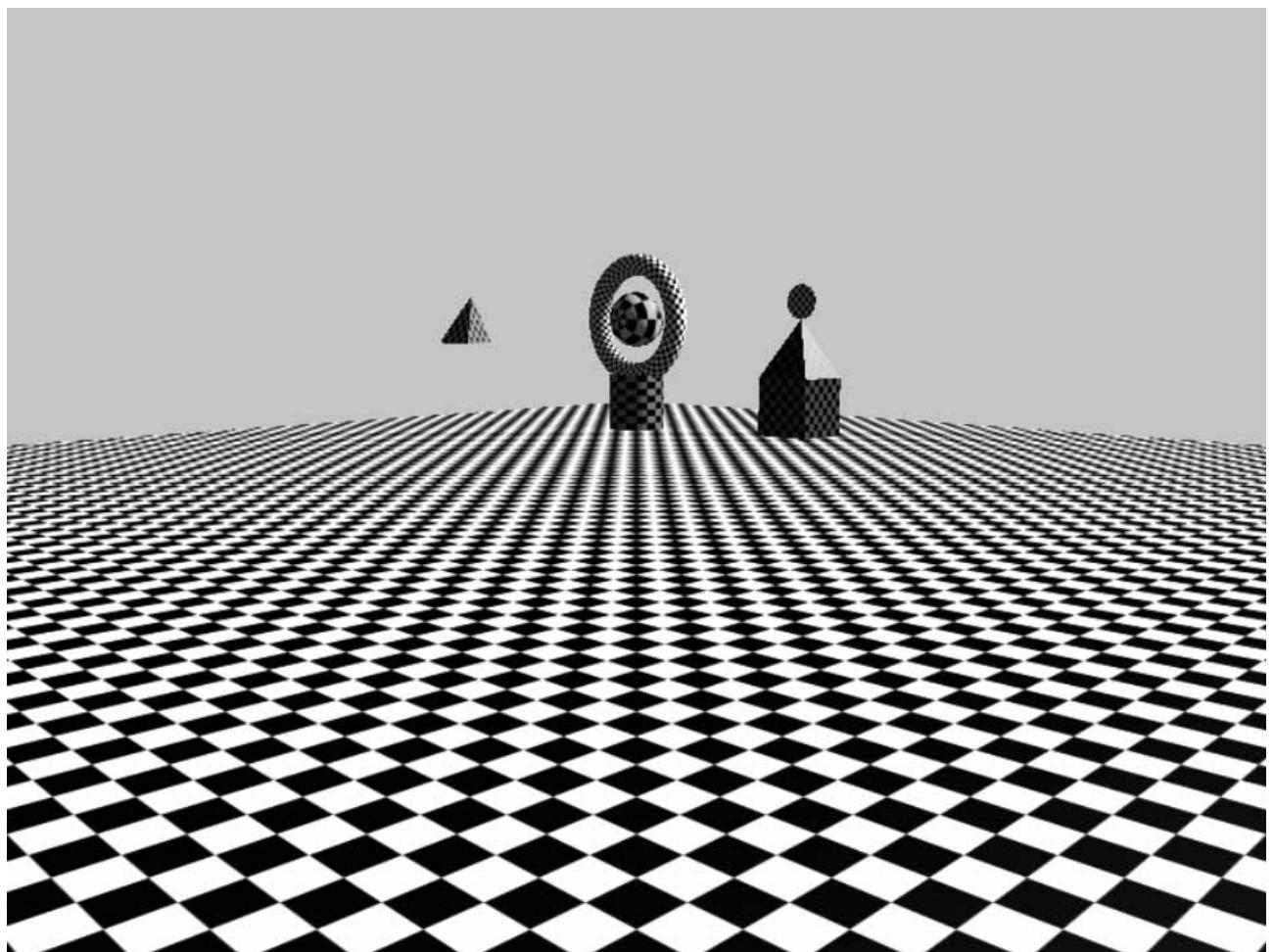


Figure 85.1: Greyscale Post-Process

Lesson 86

Blur Post-process

Blurring requires us to use a technique called box filtering. Essentially, we determine the final pixel colour based on the surrounding pixel colours. Consider it as follows:

	S1	
S2	col	S3
	S4	

$$col = \frac{S1 + S2 + S3 + S4}{4}$$

We can write this in matrix form as:

$$\frac{\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}}{4}$$

What we need to do is work out the texture coordinates to sample from for each of these four values. This requires us to know the inverse of the screen dimensions:

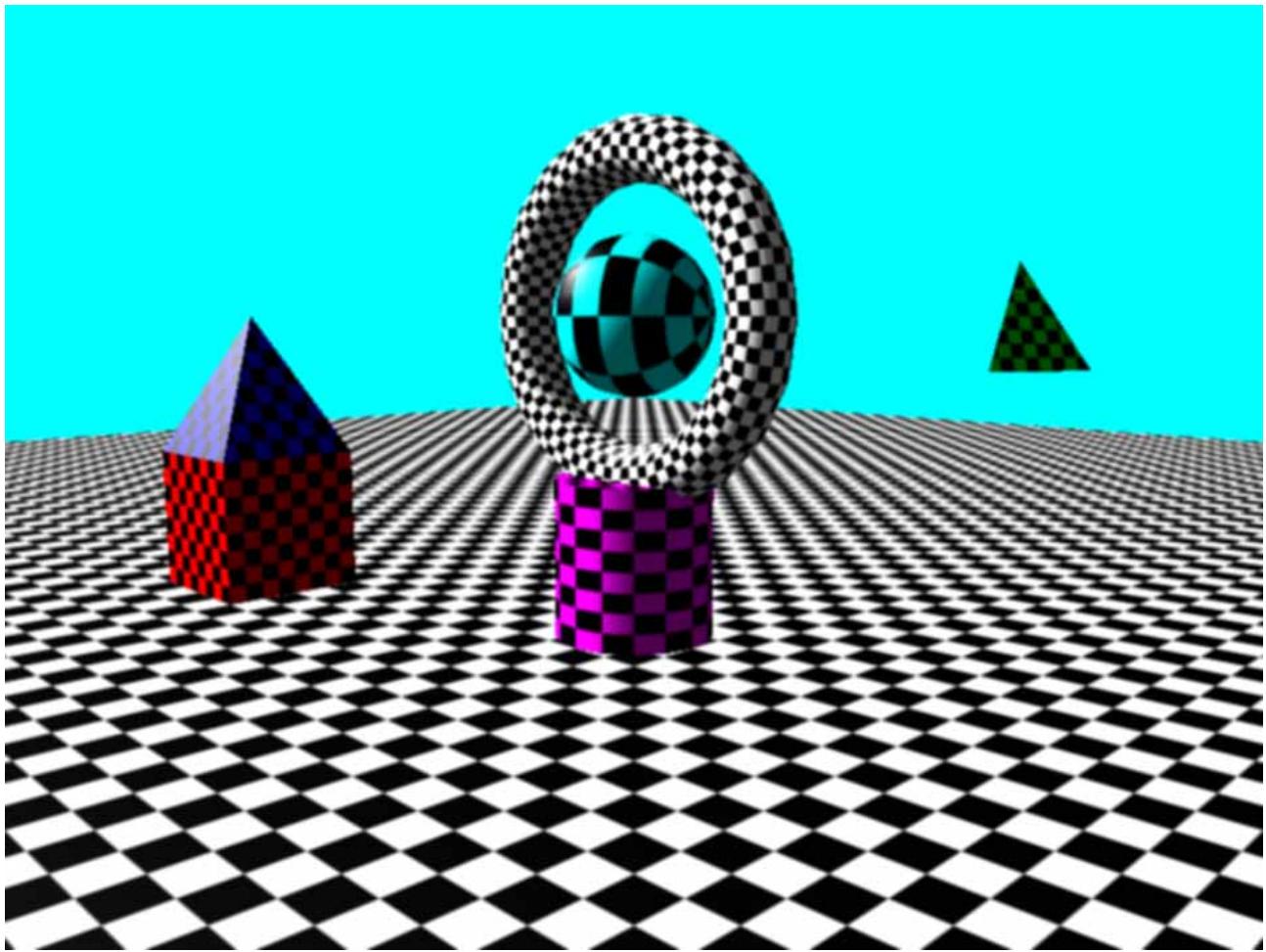
$$inv_width = \frac{1}{width}$$
$$inv_height = \frac{1}{height}$$

We use these values to calculate the texture coordinates of the individual pixels we need to sample from. That is, for each line in the matrix we perform the following calculation:

```
1 vec2 uv = tex_coord + vec2(samples[i].x * inverse_width, samples[i].y * inverse_height);
```

The new *uv* value is the texture coordinate to sample from. We then divide this value by 4 and add to the summed colour. We store this value in the *w* component in the shader.

An example output (although possibly hard to see here) is shown in Figure 86.1.

Figure 86.1: **utput from Blur Shader**

86.1 Exercises

Implement the following filters:

1. An edge detection filter can be given as follows:

$$\begin{array}{r} \left[\begin{array}{ccc} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{array} \right] \\ \hline 1 \end{array}$$

2. A sharpening filter can be given as follows:

$$\begin{array}{r} \left[\begin{array}{ccc} 0 & -2 & 0 \\ -2 & 11 & -2 \\ 0 & -2 & 0 \end{array} \right] \\ \hline 3 \end{array}$$

3. Gaussian blur can be calculated in a two phase process as follows:

$$\begin{array}{r} \left[\begin{array}{ccccccc} 1 & 6 & 15 & 20 & 15 & 6 & 1 \end{array} \right] \\ \hline 64 \end{array}$$

$$\begin{array}{r} \left[\begin{array}{c} 1 \\ 6 \\ 15 \\ 20 \\ 15 \\ 6 \\ 1 \end{array} \right] \\ \hline 64 \end{array}$$

Lesson 87

Motion Blur Post-process

Motion blur uses multiple render passes to generate the required effect. Essentially, we are undertaking the chain of passes shown in Figure 87.1.

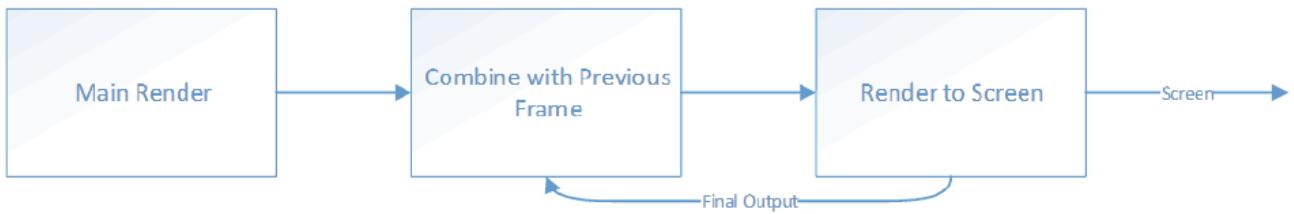


Figure 87.1: Motion Blur Post-process Pipeline

The motion blur is controlled by blending the current render with the previous one. This is just our standard multi-texturing approach. You should be able to complete the lesson with this information. Figure 87.2 provides an example output.

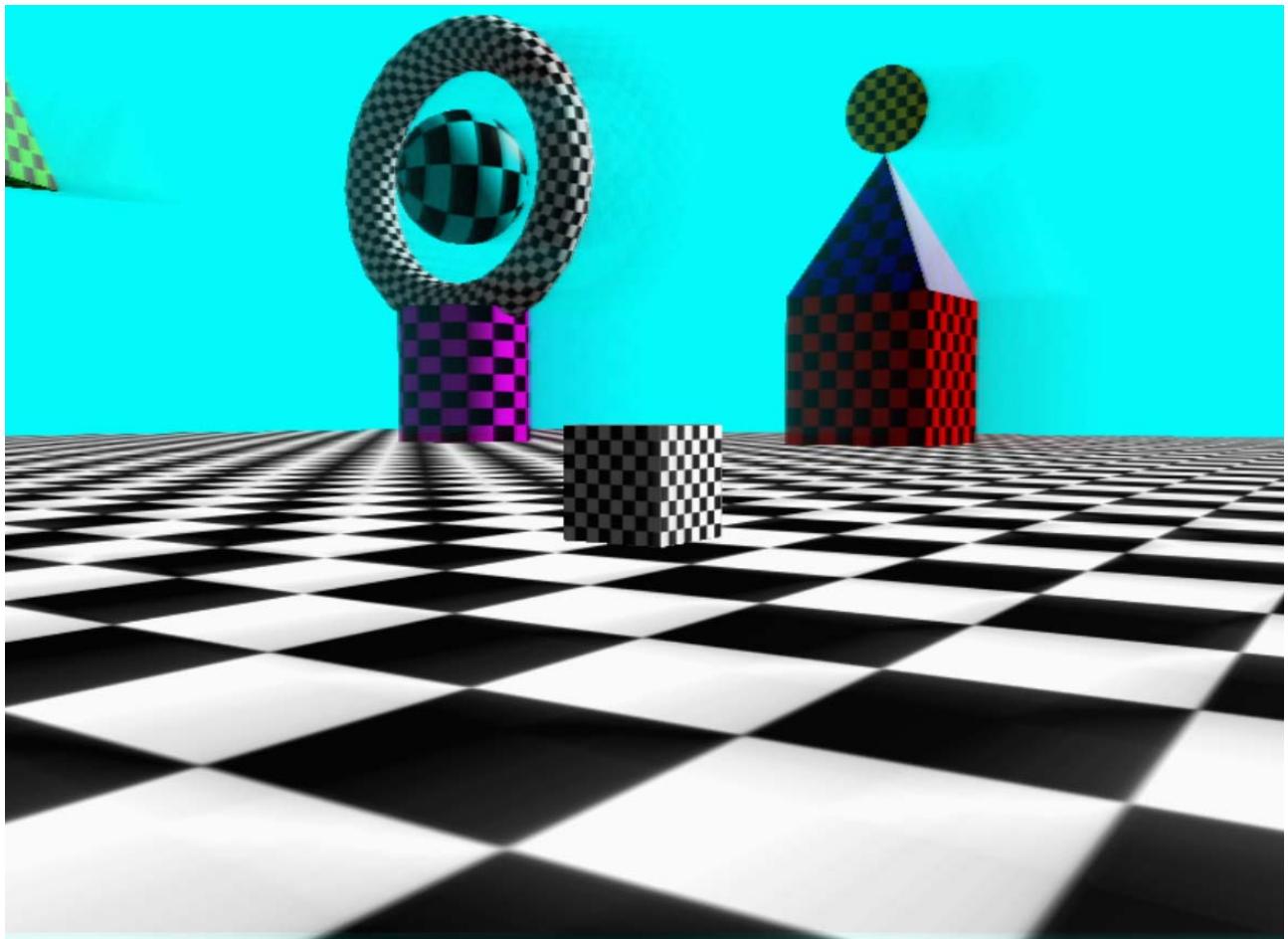


Figure 87.2: Motion Blur Post Process

Lesson 88

Depth-of-Field

Depth-of-field is another multi-pass technique that allows us to create the illusion of focus on our objects. Figure 88.1 shows the post-process pipeline involved.

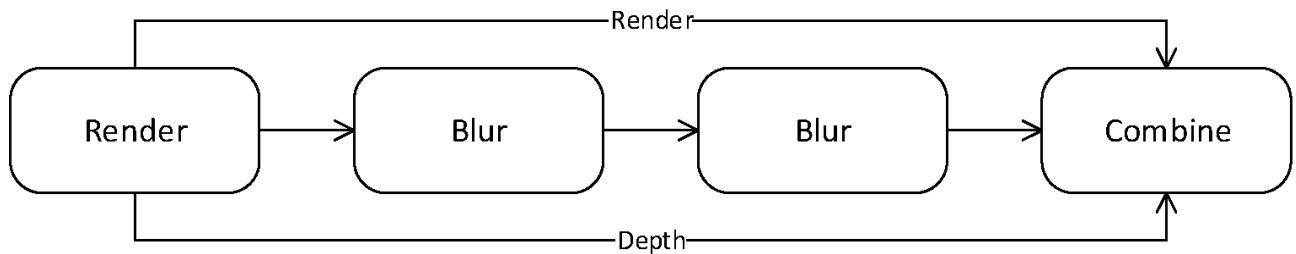


Figure 88.1: **Depth of Field Post-process Pipeline**

Again, you should be able to work out everything else you need. An example output is provided in Figure 88.2.

88.1 Exercise

The depth of field effect can be enhanced by adding in the Gaussian blur post process before combining. Try this.

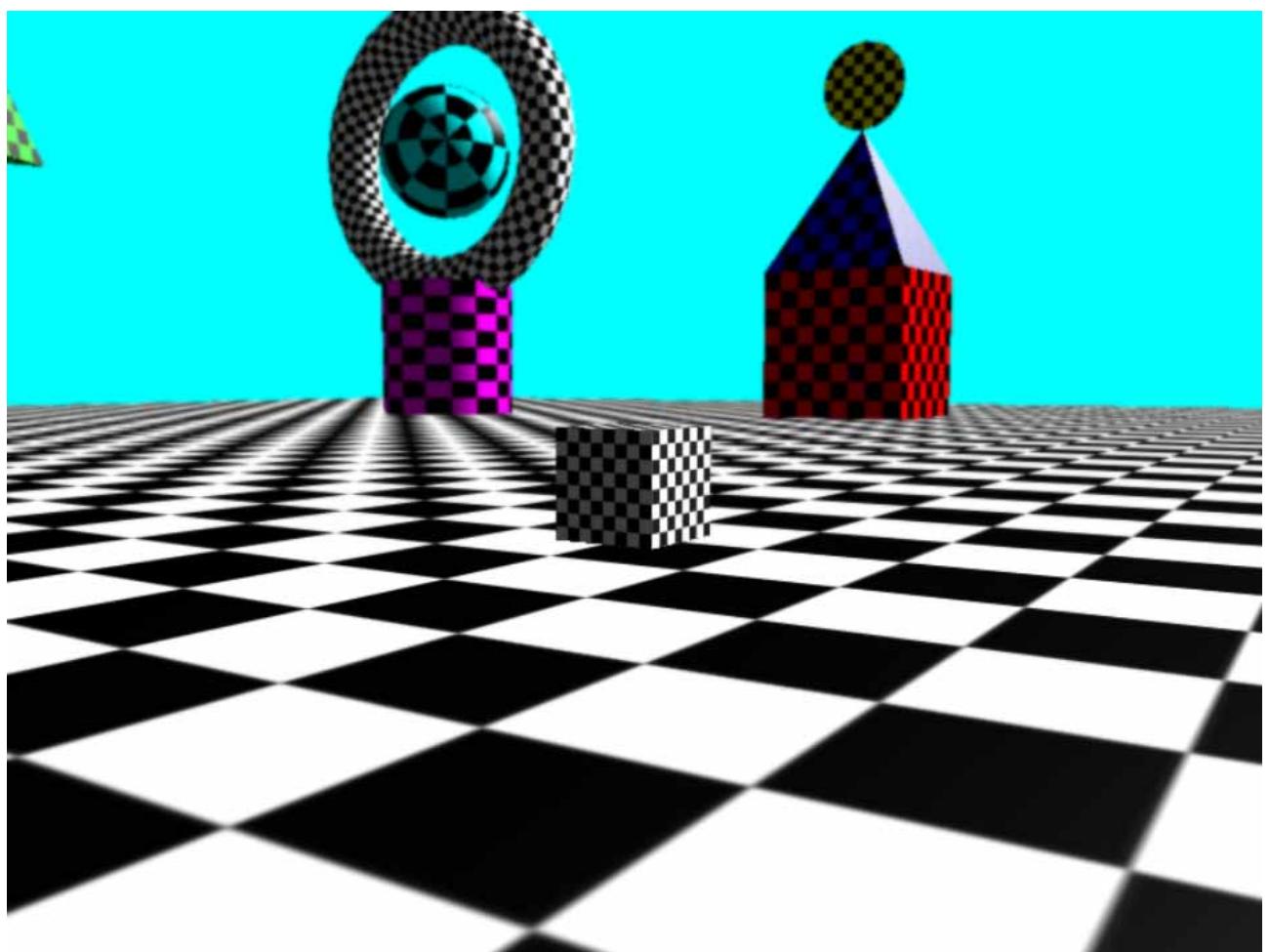


Figure 88.2: Output from Depth-of-Field Post Process

Lesson 89

Masking Post-process

Our next post-process is a simple single pass one - masking. This just involves us using some multi-texturing techniques as we have done before. An example output is given in Figure 89.1.

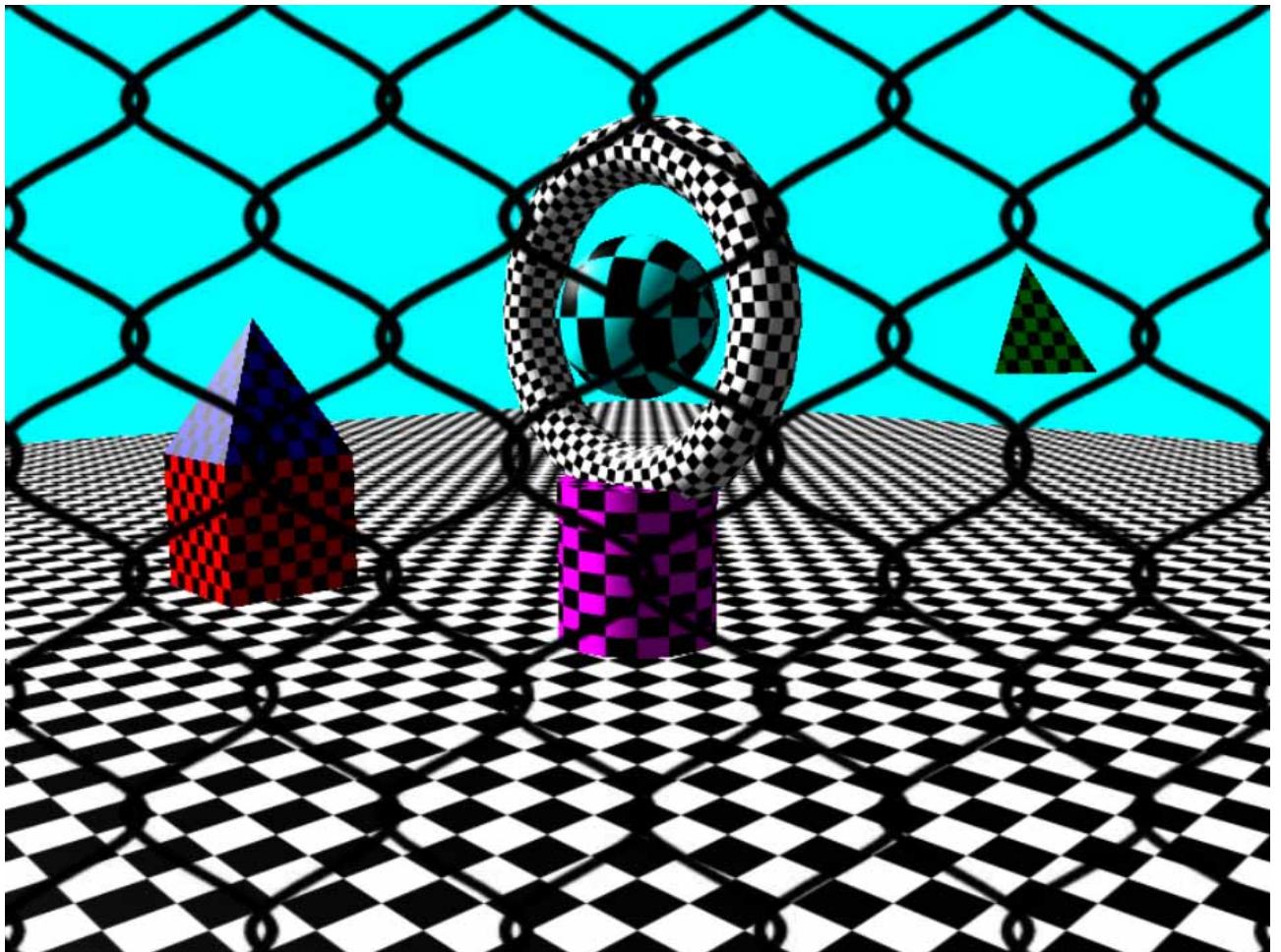


Figure 89.1: Output from Masking Post-Process

Lesson 90

Ambient Occlusion

Lesson 91

Deferred Shading

GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc.

<<http://fsf.org/>>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document “free” in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The “**Document**”, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as “**you**”. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A “**Modified Version**” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A “**Secondary Section**” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “**Invariant Sections**” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The “**Cover Texts**” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A “**Transparent**” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not “Transparent” is called “**Opaque**”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The “**Title Page**” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

The “**publisher**” means any person or entity that distributes copies of the Document to the public.

A section “**Entitled XYZ**” means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as “**Acknowledgements**”, “**Dedications**”, “**Endorsements**”, or “**History**”.) To “**Preserve the Title**” of such a section when you modify the Document means that it remains a section “Entitled XYZ” according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers

may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text,

to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrighted works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC")

contained in the site means any set of copyrightable works thus published on the MMC site.

“CC-BY-SA” means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

“Incorporate” means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is “eligible for relicensing” if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

Bibliography

- [1] Tomas Akenine-Möller, Eric Haines, and Natty Hoffman. *Real-Time Rendering*. A. K. Peters, Ltd., Natick, MA, USA, 3rd edition, 2008.
- [2] Eric Lengyel. *Mathematics for 3D Game Programming and Computer Graphics*. Cengage Learning PTR, third edition, 2012.
- [3] Dave Shreiner and Edward Angel. *Interactive Computer Graphics: A Top-Down Approach with Shader-Based OpenGL*. Pearson Education, 6th edition, May 2011.