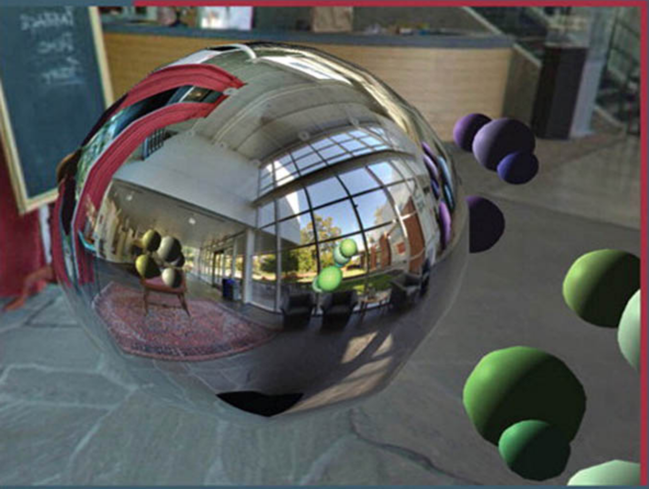
WebGL® Programming Guide

Interactive 3D Graphics Programming with WebGL



Kouichi Matsuda ▪ Rodger Lea

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Praise for
WebGL Programming Guide

"WebGL provides one of the final features for creating applications that deliver 'the desktop application experience' in a web browser, and the *WebGL Programming Guide* leads the way in creating those applications. Its coverage of all aspects of using WebGL—JavaScript, OpenGL ES, and fundamental graphics techniques—delivers a thorough education on everything you need to get going. Web-based applications are the wave of the future, and this book will get you ahead of the curve!"

Dave Shreiner, Coauthor of *The OpenGL Programming Guide, Eighth Edition*; Series Editor, *OpenGL Library* (Addison Wesley)

"HTML5 is evolving the Web into a highly capable application platform supporting beautiful, engaging, and fully interactive applications that run portably across many diverse systems. WebGL is a vital part of HTML5, as it enables web programmers to access the full power and functionality of state-of-the-art 3D graphics acceleration. WebGL has been designed to run securely on any web-capable system and will unleash a new wave of developer innovation in connected 3D web-content, applications, and user interfaces. This book will enable web developers to fully understand this new wave of web functionality and leverage the exciting opportunities it creates."

Neil Trevett, Vice President Mobile Content, NVIDIA; President, The Khronos Group

"With clear explanations supported by beautiful 3D renderings, this book does wonders in transforming a complex topic into something approachable and appealing. Even without denying the sophistication of WebGL, it is an accessible resource that beginners should consider picking up before anything else."

Evan Burchard, Author, *Web Game Developer's Cookbook* (Addison Wesley)

"Both authors have a strong OpenGL background and transfer this knowledge nicely over to WebGL, resulting in an excellent guide for beginners as well as advanced readers."

Daniel Haehn, Research Software Developer, Boston Children's Hospital

"*WebGL Programming Guide* provides a straightforward and easy-to-follow look at the mechanics of building 3D applications for the Web without relying on bulky libraries or wrappers. A great resource for developers seeking an introduction to 3D development concepts mixed with cutting-edge web technology."

Brandon Jones, Software Engineer, Google

“This is more great work from a brilliant researcher. Kouichi Matsuda shows clear and concise steps to bring the novice along the path of understanding WebGL. This is a complex topic, but he makes it possible for anyone to start using this exciting new web technology. And he includes basic 3D concepts to lay the foundation for further learning. This will be a great addition to any web designer’s library.”

Chris Marrin, WebGL Spec. Editor

“*WebGL Programming Guide* is a great way to go from a WebGL newbie to a WebGL expert. WebGL, though simple in concept, requires a lot of 3D math knowledge, and *WebGL Programming Guide* helps you build this knowledge so you’ll be able to understand and apply it to your programs. Even if you end up using some other WebGL 3D library, the knowledge learned in *WebGL Programming Guide* will help you understand what those libraries are doing and therefore allow you to tame them to your application’s specific needs. Heck, even if you eventually want to program desktop OpenGL and/or DirectX, *WebGL Programming Guide* is a great start as most 3D books are outdated relative to current 3D technology. *WebGL Programming Guide* will give you the foundation for fully understanding modern 3D graphics.”

Gregg Tavares, An Implementer of WebGL in Chrome

WebGL Programming Guide

WebGL Programming Guide:

Interactive 3D Graphics Programming with WebGL

Kouichi Matsuda
Rodger Lea

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales
international@pearsoned.com

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2013936083

ISBN-13: 978-0-321-90292-4
ISBN-10: 0-321-90292-0

Text printed in the United States on recycled paper at Edwards Brothers Malloy in Ann Arbor, Michigan

First printing: June 2013

Editor-in-Chief
Mark Taub

Executive Editor
Laura Lewin

Development Editor
Sheri Cain

Managing Editor
Krista Hansing

Senior Project Editor
Lori Lyons

Copy Editor
Gill Editorial Services

Senior Indexer
Cheryl Lenser

Proofreader
Paula Lowell

Technical Reviewers
Jeff Gilbert
Daniel Haehn
Rick Rafey

Editorial Assistant
Olivia Basegio

Interior Designer
Mark Shirar

Cover Designer
Chuti Prasertsith

Senior Compositor
Gloria Schurick

Graphics
Laura Robbins

Manufacturing Buyer
Dan Uhrig

Thoughts are filled along with time, the distant days will not return,
and time passed is like a spiral of semiprecious stones...
—Kouichi Matsuda

To my wife, family, and friends—for making life fun.
—Rodger Lea

Contents

Preface xvii

1. Overview of WebGL 1

Advantages of WebGL 3
- You Can Start Developing 3D Graphics Applications Using Only a Text Editor 3
- Publishing Your 3D Graphics Applications Is Easy 4
- You Can Leverage the Full Functionality of the Browser 5
- Learning and Using WebGL Is Easy 5

Origins of WebGL 5
Structure of WebGL Applications 6
Summary 7

2. Your First Step with WebGL 9

What Is a Canvas? 9
- Using the <canvas> Tag 11
- DrawRectangle.js 13

The World's Shortest WebGL Program: Clear Drawing Area 16
- The HTML File (HelloCanvas.html) 17
- JavaScript Program (HelloCanvas.js) 18
- Experimenting with the Sample Program 23

Draw a Point (Version 1) 23
- HelloPoint1.html 25
- HelloPoint1.js 25
- What Is a Shader? 27
- The Structure of a WebGL Program that Uses Shaders 28
- Initializing Shaders 30
- Vertex Shader 33
- Fragment Shader 35
- The Draw Operation 36
- The WebGL Coordinate System 38
- Experimenting with the Sample Program 40

Draw a Point (Version 2) 41
- Using Attribute Variables 41
- Sample Program (HelloPoint2.js) 42
- Getting the Storage Location of an Attribute Variable 44
- Assigning a Value to an Attribute Variable 45
- Family Methods of gl.vertexAttrib3f() 47
- Experimenting with the Sample Program 49

Draw a Point with a Mouse Click....50
Sample Program (ClickedPoints.js)....50
Register Event Handlers....52
Handling Mouse Click Events....53
Experimenting with the Sample Program....57
Change the Point Color....58
Sample Program (ColoredPoints.js)....59
Uniform Variables....61
Retrieving the Storage Location of a Uniform Variable....62
Assigning a Value to a Uniform Variable....63
Family Methods of gl.uniform4f()....65
Summary....66

3. Drawing and Transforming Triangles 67

Drawing Multiple Points....68
Sample Program (MultiPoint.js)....70
Using Buffer Objects....72
Create a Buffer Object (gl.createBuffer())....74
Bind a Buffer Object to a Target (gl.bindBuffer())....75
Write Data into a Buffer Object (gl.bufferData())....76
Typed Arrays....78
Assign the Buffer Object to an Attribute Variable (gl.vertexAttribPointer())....79
Enable the Assignment to an Attribute Variable (gl.enableVertexAttribArray())....81
The Second and Third Parameters of gl.drawArrays()....82
Experimenting with the Sample Program....84
Hello Triangle....85
Sample Program (HelloTriangle.js)....85
Basic Shapes....87
Experimenting with the Sample Program....89
Hello Rectangle (HelloQuad)....89
Experimenting with the Sample Program....91
Moving, Rotating, and Scaling....91
Translation....92
Sample Program (TranslatedTriangle.js)....93
Rotation....96
Sample Program (RotatedTriangle.js)....99
Transformation Matrix: Rotation....102
Transformation Matrix: Translation....105
Rotation Matrix, Again....106
Sample Program (RotatedTriangle_Matrix.js)....107

Reusing the Same Approach for Translation 111
Transformation Matrix: Scaling 111
Summary 113

4. More Transformations and Basic Animation 115

Translate and Then Rotate 115
Transformation Matrix Library: cuon-matrix.js 116
Sample Program (RotatedTriangle_Matrix4.js) 117
Combining Multiple Transformation 119
Sample Program (RotatedTranslatedTriangle.js) 121
Experimenting with the Sample Program 123
Animation 124
The Basics of Animation 125
Sample Program (RotatingTriangle.js) 126
Repeatedly Call the Drawing Function (tick()) 129
Draw a Triangle with the Specified Rotation Angle (draw()) 130
Request to Be Called Again (requestAnimationFrame()) 131
Update the Rotation Angle (animate()) 133
Experimenting with the Sample Program 135
Summary 136

5. Using Colors and Texture Images 137

Passing Other Types of Information to Vertex Shaders 137
Sample Program (MultiAttributeSize.js) 139
Create Multiple Buffer Objects 140
The gl.vertexAttribPointer() Stride and Offset Parameters 141
Sample Program (MultiAttributeSize_Interleaved.js) 142
Modifying the Color (Varying Variable) 146
Sample Program (MultiAttributeColor.js) 147
Experimenting with the Sample Program 150
Color Triangle (ColoredTriangle.js) 151
Geometric Shape Assembly and Rasterization 151
Fragment Shader Invocations 155
Experimenting with the Sample Program 156
Functionality of Varying Variables and the Interpolation Process 157
Pasting an Image onto a Rectangle 160
Texture Coordinates 162
Pasting Texture Images onto the Geometric Shape 162
Sample Program (TexturedQuad.js) 163
Using Texture Coordinates (initVertexBuffers()) 166
Setting Up and Loading Images (initTextures()) 166
Make the Texture Ready to Use in the WebGL System (loadTexture()) 170

Flip an Image's Y-Axis 170
Making a Texture Unit Active (gl.activeTexture()) 171
Binding a Texture Object to a Target (gl.bindTexture()) 173
Set the Texture Parameters of a Texture Object (gl.texParameteri()) 174
Assigning a Texture Image to a Texture Object (gl.texImage2D()) 177
Pass the Texture Unit to the Fragment Shader (gl.uniform1i()) 179
Passing Texture Coordinates from the Vertex Shader to the Fragment Shader ... 180
Retrieve the Texel Color in a Fragment Shader (texture2D()) 181
Experimenting with the Sample Program 182
Pasting Multiple Textures to a Shape 183
Sample Program (MultiTexture.js) 184
Summary 189

6. The OpenGL ES Shading Language (GLSL ES) 191

Recap of Basic Shader Programs 191
Overview of GLSL ES 192
Hello Shader! 193
Basics 193
Order of Execution 193
Comments 193
Data (Numerical and Boolean Values) 194
Variables 194
GLSL ES Is a Type Sensitive Language 195
Basic Types 195
Assignment and Type Conversion 196
Operations 197
Vector Types and Matrix Types 198
Assignments and Constructors 199
Access to Components 201
Operations 204
Structures 207
Assignments and Constructors 207
Access to Members 207
Operations 208
Arrays 208
Samplers 209
Precedence of Operators 210
Conditional Control Flow and Iteration 211
if Statement and if-else Statement 211
for Statement 211
continue, break, discard Statements 212

Functions ... 213
Prototype Declarations ... 214
Parameter Qualifiers ... 214
Built-In Functions ... 215
Global Variables and Local Variables ... 216
Storage Qualifiers ... 217
const Variables ... 217
Attribute Variables ... 218
Uniform Variables ... 218
Varying Variables ... 219
Precision Qualifiers ... 219
Preprocessor Directives ... 221
Summary ... 223

7. Toward the 3D World **225**

What's Good for Triangles Is Good for Cubes ... 225
Specifying the Viewing Direction ... 226
Eye Point, Look-At Point, and Up Direction ... 227
Sample Program (LookAtTriangles.js) ... 229
Comparing LookAtTriangles.js with RotatedTriangle_Matrix4.js ... 232
Looking at Rotated Triangles from a Specified Position ... 234
Sample Program (LookAtRotatedTriangles.js) ... 235
Experimenting with the Sample Program ... 236
Changing the Eye Point Using the Keyboard ... 238
Sample Program (LookAtTrianglesWithKeys.js) ... 238
Missing Parts ... 241
Specifying the Visible Range (Box Type) ... 241
Specify the Viewing Volume ... 242
Defining a Box-Shaped Viewing Volume ... 243
Sample Program (OrthoView.html) ... 245
Sample Program (OrthoView.js) ... 246
Modifying an HTML Element Using JavaScript ... 247
The Processing Flow of the Vertex Shader ... 248
Changing Near or Far ... 250
Restoring the Clipped Parts of the Triangles (LookAtTrianglesWithKeys_ViewVolume.js) ... 251
Experimenting with the Sample Program ... 253
Specifying the Visible Range Using a Quadrangular Pyramid ... 254
Setting the Quadrangular Pyramid Viewing Volume ... 256
Sample Program (PerspectiveView.js) ... 258
The Role of the Projection Matrix ... 260
Using All the Matrices (Model Matrix, View Matrix, and Projection Matrix) ... 262

Sample Program (PerspectiveView_mvp.js) 263
Experimenting with the Sample Program 266
Correctly Handling Foreground and Background Objects 267
Hidden Surface Removal 270
Sample Program (DepthBuffer.js) 272
Z Fighting 273
Hello Cube 275
Drawing the Object with Indices and Vertices Coordinates 277
Sample Program (HelloCube.js) 278
Writing Vertex Coordinates, Colors, and Indices to the Buffer Object 281
Adding Color to Each Face of a Cube 284
Sample Program (ColoredCube.js) 285
Experimenting with the Sample Program 287
Summary 289

8. Lighting Objects **291**

Lighting 3D Objects 291
Types of Light Source 293
Types of Reflected Light 294
Shading Due to Directional Light and Its Diffuse Reflection 296
Calculating Diffuse Reflection Using the Light Direction and the Orientation of a Surface 297
The Orientation of a Surface: What Is the Normal? 299
Sample Program (LightedCube.js) 302
Add Shading Due to Ambient Light 307
Sample Program (LightedCube_ambient.js) 308
Lighting the Translated-Rotated Object 310
The Magic Matrix: Inverse Transpose Matrix 311
Sample Program (LightedTranslatedRotatedCube.js) 312
Using a Point Light Object 314
Sample Program (PointLightedCube.js) 315
More Realistic Shading: Calculating the Color per Fragment 319
Sample Program (PointLightedCube_perFragment.js) 319
Summary 321

9. Hierarchical Objects **323**

Drawing and Manipulating Objects Composed of Other Objects 324
Hierarchical Structure 325
Single Joint Model 326
Sample Program (JointModel.js) 328
Draw the Hierarchical Structure (draw()) 332
A Multijoint Model 334

Sample Program (MultiJointModel.js) 335
Draw Segments (drawBox()) 339
Draw Segments (drawSegment()) 340
Shader and Program Objects: The Role of initShaders() 344
Create Shader Objects (gl.createShader()) 345
Store the Shader Source Code in the Shader Objects (g.shaderSource()) 346
Compile Shader Objects (gl.compileShader()) 347
Create a Program Object (gl.createProgram()) 349
Attach the Shader Objects to the Program Object (gl.attachShader()) 350
Link the Program Object (gl.linkProgram()) 351
Tell the WebGL System Which Program Object to Use (gl.useProgram()) 353
The Program Flow of initShaders() 353
Summary 356

10. Advanced Techniques 357

Rotate an Object with the Mouse 357
How to Implement Object Rotation 358
Sample Program (RotateObject.js) 358
Select an Object 360
How to Implement Object Selection 361
Sample Program (PickObject.js) 362
Select the Face of the Object 365
Sample Program (PickFace.js) 366
HUD (Head Up Display) 368
How to Implement a HUD 369
Sample Program (HUD.html) 369
Sample Program (HUD.js) 370
Display a 3D Object on a Web Page (3DoverWeb) 372
Fog (Atmospheric Effect) 372
How to Implement Fog 373
Sample Program (Fog.js) 374
Use the w Value (Fog_w.js) 376
Make a Rounded Point 377
How to Implement a Rounded Point 377
Sample Program (RoundedPoints.js) 378
Alpha Blending 380
How to Implement Alpha Blending 380
Sample Program (LookAtBlendedTriangles.js) 381
Blending Function 382
Alpha Blend 3D Objects (BlendedCube.js) 384
How to Draw When Alpha Values Coexist 385

Switching Shaders 386
How to Implement Switching Shaders 387
Sample Program (ProgramObject.js) 387
Use What You've Drawn as a Texture Image 392
Framebuffer Object and Renderbuffer Object 392
How to Implement Using a Drawn Object as a Texture 394
Sample Program (FramebufferObjectj.js) 395
Create Frame Buffer Object (gl.createFramebuffer()) 397
Create Texture Object and Set Its Size and Parameters 397
Create Renderbuffer Object (gl.createRenderbuffer()) 398
Bind Renderbuffer Object to Target and Set Size (gl.bindRenderbuffer(), gl.renderbufferStorage()) 399
Set Texture Object to Framebuffer Object (gl.bindFramebuffer(), gl.framebufferTexture2D()) 400
Set Renderbuffer Object to Framebuffer Object (gl.framebufferRenderbuffer()) 401
Check Configuration of Framebuffer Object (gl.checkFramebufferStatus()) 402
Draw Using the Framebuffer Object 403
Display Shadows 405
How to Implement Shadows 405
Sample Program (Shadow.js) 406
Increasing Precision 412
Sample Program (Shadow_highp.js) 413
Load and Display 3D Models 414
The OBJ File Format 417
The MTL File Format 418
Sample Program (OBJViewer.js) 419
User-Defined Object 422
Sample Program (Parser Code in OBJViewer.js) 423
Handling Lost Context 430
How to Implement Handling Lost Context 431
Sample Program (RotatingTriangle_contextLost.js) 432
Summary 434

A. No Need to Swap Buffers in WebGL 437

B. Built-in Functions of GLSL ES 1.0 441

Angle and Trigonometry Functions 441
Exponential Functions 443
Common Functions 444
Geometric Functions 447

Matrix Functions..448
Vector Functions..449
Texture Lookup Functions..451

C. Projection Matrices 453

Orthogonal Projection Matrix..453
Perspective Projection Matrix..453

D. WebGL/OpenGL: Left or Right Handed? 455

Sample Program CoordinateSystem.js...456
Hidden Surface Removal and the Clip Coordinate System.......................................459
The Clip Coordinate System and the Viewing Volume...460
What Is Correct?..462
Summary 464

E. The Inverse Transpose Matrix 465

F. Load Shader Programs from Files 471

G. World Coordinate System Versus Local Coordinate System 473

The Local Coordinate System..474
The World Coordinate System...475
Transformations and the Coordinate Systems..477

H. Web Browser Settings for WebGL 479

Glossary 481

References 485

Index 487

Preface

WebGL is a technology that enables drawing, displaying, and interacting with sophisticated interactive three-dimensional computer graphics ("3D graphics") from within web browsers. Traditionally, 3D graphics has been restricted to high-end computers or dedicated game consoles and required complex programming. However, as both personal computers and, more importantly, web browsers have become more sophisticated, it has become possible to create and display 3D graphics using accessible and well-known web technologies. This book provides a comprehensive overview of WebGL and takes the reader, step by step, through the basics of creating WebGL applications. Unlike other 3D graphics technologies such as OpenGL and Direct3D, WebGL applications can be constructed as web pages so they can be directly executed in the browsers without installing any special plug-ins or libraries. Therefore, you can quickly develop and try out a sample program with a standard PC environment; because everything is web based, you can easily publish the programs you have constructed on the web. One of the promises of WebGL is that, because WebGL applications are constructed as web pages, the same program can be run across a range of devices, such as smart phones, tablets, and game consoles, through the browser. This powerful model means that WebGL will have a significant impact on the developer community and will become one of the preferred tools for graphics programming.

Who the Book Is For

We had two main audiences in mind when we wrote this book: web developers looking to add 3D graphics to their web pages and applications, and 3D graphics programmers wishing to understand how to apply their knowledge to the web environment. For web developers who are familiar with standard web technologies such as HTML and JavaScript and who are looking to incorporate 3D graphics into their web pages or web applications, WebGL offers a simple yet powerful solution. It can be used to add 3D graphics to enhance web pages, to improve the user interface (UI) for a web application by using a 3D interface, and even to develop more complex 3D applications and games that run in web browsers.

The second target audience is programmers who have worked with one of the main 3D application programming interfaces (APIs), such as Direct3D or OpenGL, and who are interested in understanding how to apply their knowledge to the web environment. We would expect these programmers to be interested in the more complex 3D applications that can be developed in modern web browsers.

However, the book has been designed to be accessible to a wide audience using a step-by-step approach to introduce features of WebGL, and it assumes no background in 2D or 3D graphics. As such, we expect it also to be of interest to the following:

- General programmers seeking an understanding of how web technologies are evolving in the graphics area
- Students studying 2D and 3D graphics because it offers a simple way to begin to experiment with graphics via a web browser rather than setting up a full programming environment
- Web developers exploring the "bleeding edge" of what is possible on mobile devices such as Android or iPhone using the latest mobile web browsers

What the Book Covers

This book covers the WebGL 1.0 API along with all related JavaScript functions. You will learn how HTML, JavaScript, and WebGL are related, how to set up and run WebGL applications, and how to incorporate sophisticated 3D program "shaders" under the control of JavaScript. The book details how to write vertex and fragment shaders, how to implement advanced rendering techniques such as per-pixel lighting and shadowing, and basic interaction techniques such as selecting 3D objects. Each chapter develops a number of working, fully functional WebGL applications and explains key WebGL features through these examples. After finishing the book, you will be ready to write WebGL applications that fully harness the programmable power of web browsers and the underlying graphics hardware.

How the Book Is Structured

This book is organized to cover the API and related web APIs in a step-by-step fashion, building up your knowledge of WebGL as you go.

Chapter 1—Overview of WebGL

This chapter briefly introduces you to WebGL, outlines some of the key features and advantages of WebGL, and discusses its origins. It finishes by explaining the relationship of WebGL to HTML5 and JavaScript and which web browsers you can use to get started with your exploration of WebGL.

Chapter 2—Your First Step with WebGL

This chapter explains the `<canvas>` element and the core functions of WebGL by taking you, step-by-step, through the construction of several example programs. Each example is written in JavaScript and uses WebGL to display and interact with a simple shape on a web page. The example WebGL programs will highlight some key points, including: (1) how WebGL uses the `<canvas>` element object and how to draw on it; (2) the linkage between HTML and WebGL using JavaScript; (3) simple WebGL drawing functions; and (4) the role of shader programs within WebGL.

Chapter 3—Drawing and Transforming Triangles

This chapter builds on those basics by exploring how to draw more complex shapes and how to manipulate those shapes in 3D space. This chapter looks at: (1) the critical role of triangles in 3D graphics and WebGL's support for drawing triangles; (2) using multiple triangles to draw other basic shapes; (3) basic transformations that move, rotate, and scale triangles using simple equations; and (4) how matrix operations make transformations simple.

Chapter 4—More Transformations and Basic Animation

In this chapter, you explore further transformations and begin to combine transformations into animations. You: (1) are introduced to a matrix transformation library that hides the mathematical details of matrix operations; (2) use the library to quickly and easily combine multiple transformations; and (3) explore animation and how the library helps you animate simple shapes. These techniques provide the basics to construct quite complex WebGL programs and will be used in the sample programs in the following chapters.

Chapter 5—Using Colors and Texture Images

Building on the basics described in previous chapters, you now delve a little further into WebGL by exploring the following three subjects: (1) besides passing vertex coordinates, how to pass other data such as color information to the vertex shader; (2) the conversion from a shape to fragments that takes place between the vertex shader and the fragment shader, which is known as the rasterization process; and (3) how to map images (or textures) onto the surfaces of a shape or object. This chapter is the final chapter focusing on the key functionalities of WebGL.

Chapter 6—The OpenGL ES Shading Language (GLSL ES)

This chapter takes a break from examining WebGL sample programs and explains the core features of the OpenGL ES Shading Language (GLSL ES) in detail. You will cover: (1) data, variables, and variable types; (2) vector, matrix, structure, array, and sampler; (3) operators, control flow, and functions; (4) attributes, uniforms, and varyings; (5) precision qualifier; and (6) preprocessor and directives. By the end of this chapter you will have a good understanding of GLSL ES and how it can be used to write a variety of shaders.

Chapter 7—Toward the 3D World

This chapter takes the first step into the 3D world and explores the implications of moving from 2D to 3D. In particular, you will explore: (1) representing the user's view into the 3D world; (2) how to control the volume of 3D space that is viewed; (3) clipping; (4) foreground and background objects; and (5) drawing a 3D object—a cube. All these issues have a significant impact on how the 3D scene is drawn and presented to viewers. A mastery of them is critical to building compelling 3D scenes.

Chapter 8—Lighting Objects

This chapter focuses on lighting objects, looking at different light sources and their effects on the 3D scene. Lighting is essential if you want to create realistic 3D scenes because it helps to give the scene a sense of depth.

The following key points are discussed in this chapter: (1) shading, shadows, and different types of light sources including point, directional, and ambient; (2) reflection of light in the 3D scene and the two main types: diffuse and ambient reflection; and (3) the details of shading and how to implement the effect of light to make objects look three-dimensional.

Chapter 9—Hierarchical Objects

This chapter is the final chapter describing the core features and how to program with WebGL. Once completed, you will have mastered the basics of WebGL and will have enough knowledge to be able to create realistic and interactive 3D scenes. This chapter focuses on hierarchical objects, which are important because they allow you to progress beyond single objects like cubes or blocks to more complex objects that you can use for game characters, robots, and even modeling humans.

Chapter 10—Advanced Techniques

This chapter touches on a variety of important techniques that use what you have learned so far and provide you with an essential toolkit for building interactive, compelling 3D graphics. Each technique is introduced through a complete example, which you can reuse when building your own WebGL applications.

Appendix A—No Need to Swap Buffers in WebGL

This appendix explains why WebGL programs don't need to swap buffers.

Appendix B—Built-In Functions of GLSL ES 1.0

This appendix provides a reference for all the built-in functions available in the OpenGL ES Shading Language.

Appendix C—Projection Matrices

This appendix provides the projection matrices generated by `Matrix4.setOrtho()` and `Matrix4.setPerspective()`.

Appendix D—WebGL/OpenGL: Left or Right Handed?

This appendix explains how WebGL and OpenGL deal internally with the coordinate system and clarify that technically, both WebGL and OpenGL are agnostic as to handedness.

Appendix E—The Inverse Transpose Matrix

This appendix explains how the inverse transpose matrix of the model matrix can deal with the transformation of normal vectors.

Appendix F—Loading Shader Programs from Files

This appendix explains how to load the shader programs from files.

Appendix G—World Coordinate System Versus Local Coordinate System

This appendix explains the different coordinate systems and how they are used in 3D graphics.

Appendix H—Web Browser Settings for WebGL

This appendix explains how to use advanced web browser settings to ensure that WebGL is displayed correctly, and what to do if it isn't.

WebGL-Enabled Browsers

At the time of writing, WebGL is supported by Chrome, Firefox, Safari, and Opera. Sadly, some browsers, such as IE9 (Microsoft Internet Explorer), don't yet support WebGL. In this book, we use the Chrome browser released by Google, which, in addition to WebGL supports a number of useful features such as a console function for debugging. We have checked the sample programs in this book using the following environment (Table P.1) but would expect them to work with any browser supporting WebGL.

Table P.1 PC Environment

Browser	Chrome (25.0.1364.152 m)
OS	Windows 7 and 8
Graphics boards	NVIDIA Quadro FX 380, NVIDIA GT X 580, NVIDIA GeForce GTS 450, Mobile Intel 4 Series Express Chipset Family, AMD Radeon HD 6970

Refer to the www.khronos.org/webgl/wiki/BlacklistsAndWhitelists for an updated list of which hardware cards are known to cause problems.

To confirm that you are up and running, download Chrome (or use your preferred browser) and point it to the companion website for this book at https://sites.google.com/site/webglbook/

Navigate to Chapter 3 and click the link to the sample file `HelloTriangle.html`. If you can see a red triangle as shown in Figure P.1 in the browser, WebGL is working.

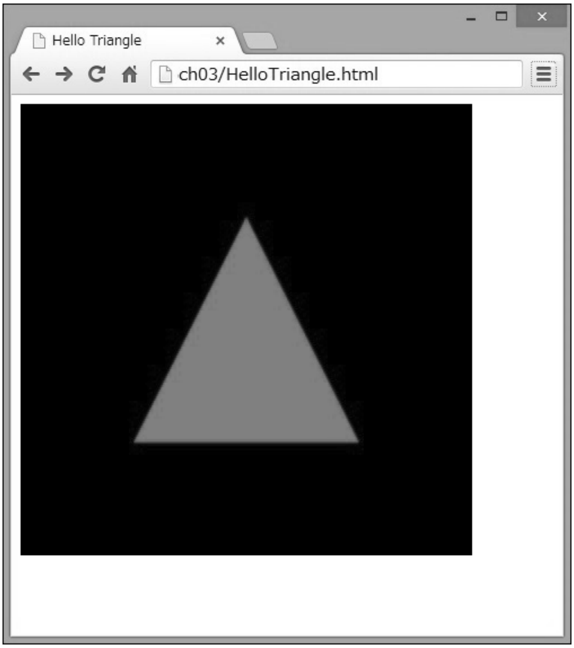


Figure P.1 Loading `HelloTriangle` results in a red triangle

If you don't see the red triangle shown in the figure, take a look at Appendix H, which explains how to change your browser settings to load WebGL.

Sample Programs and Related Links

All sample programs in this book and related links are available on the companion websites. The official site hosted by the publisher is www.informit.com/title/9780321902924 and the author site is hosted at https://sites.google.com/site/webglbook/.

The latter site contains the links to each sample program in this book. You can run each one directly by clicking the links.

If you want to modify the sample programs, you can download the zip file of all the samples, available on both sites, to your local disk. In this case, you should note that the sample program consists of both the HTML file and the associated JavaScript file in the same folder. For example, for the sample program `HelloTriangle`, you need both `HelloTriangle.html` and `HelloTriangle.js`. To run `HelloTriangle`, double-click `HelloTriangle.html`.

Style Conventions

These style conventions are used in this book:

- **Bold**—First occurrences of key terms and important words
- *Italic*—Parameter names and names of references
- `Monospace`—Code examples, methods, functions, variables, command options, JavaScript object names, filenames, and HTML tags

Acknowledgments

We have been fortunate to receive help and guidance from many talented individuals during the process of creating this book, both with the initial Japanese version and the subsequent English one.

Takafumi Kanda helped by providing numerous code samples for our support libraries and sample programs; without him, this book could not have been realized. Yasuko Kikuchi, Chie Onuma, and Yuichi Nishizawa provided valuable feedback on early versions of the book. Of particular note, one insightful comment by Ms. Kikuchi literally stopped the writing, causing a reevaluation of several sections and leading to a much stronger book. Hiroyuki Tanaka and Kazsuhira Oonishi (iLinx) gave excellent support with the sample programs, and Teruhisa Kamachi and Tetsuo Yoshitani supported the writing of sections on HTML5 and JavaScript. The WebGL working group, especially Ken Russell (Google), Chris Marin (Apple), and Dan Ginsburg (AMD), have answered many technical questions. We have been privileged to receive an endorsement from the president of the Khronos Group, Neil Trevett, and appreciate the help of Hitoshi Kasai (Principal, MIACIS Associates) who provided the connection to Mr. Trevett and the WebGL working group. In addition, thank you to Xavier Michel and Makoto Sato (Sophia University), who greatly helped with the translation of the original text and issues that arose during the translation. For the English version, Jeff Gilbert, Rick Rafey, and Daniel Haehn reviewed this book carefully and gave us excellent technical comments and feedback that greatly improved the book. Our thanks also to Laura Lewin and Olivia Basegio from Pearson, who have helped with organizing the publication and ensuring the whole process has been as smooth and as painless as possible.

We both owe a debt of gratitude to the authors of the “Red Book” (OpenGL Programming Guide) and the “Gold Book” (OpenGL ES 2.0 Programming Guide) both published by Pearson, without which this book would not have been possible. We hope, in some small way, that this book repays some of that debt.

About the Authors

Dr. Kouichi Matsuda has a broad background in user interface and user experience design and its application to novel multimedia products. His work has taken him from product development, through research, and back to development, having spent time at NEC, Sony Corporate Research, and Sony Computer Science Laboratories. He is currently a chief distinguished researcher focused on user experience and human computer interaction across a range of consumer electronics. He was the designer of the social 3D virtual world called "PAW" (personal agent-oriented virtual world), was involved in the development of the VRML97 (ISO/IEC 14772-1:1997) standard from the start, and has remained active in both VRML and X3D communities (precursors to WebGL). He has written 15 books on computer technologies and translated a further 25 into Japanese. His expertise covers user experiences, user interface, human computer interaction, natural language understanding, entertainment-oriented network services, and interface agent systems. Always on the lookout for new and exciting possibilities in the technology space, he combines his professional life with a love of hot springs, sea in summer, wines, and MANGA (at which he dabbles in drawing and illustrations). He received his Ph.D. (Engineering) from the Graduate School of Engineering, University of Tokyo, and can be reached via WebGL.prog.guide@gmail.com.

Dr. Rodger Lea is an adjunct professor with the Media and Graphics Interdisciplinary Centre at the University of British Columbia, with an interest in systems aspects of multimedia and distributed computing. With more than 20 years of experience leading research groups in both academic and industrial settings, he has worked on early versions of shared 3D worlds, helped define VRML97, developed multimedia operating systems, prototyped interactive digital TV, and led developments on multimedia home networking standards. He has published more than 60 research papers and three books, and he holds 12 patents. His current research explores the growing "Internet of Things," but he retains a passion for all things media and graphics.

This page intentionally left blank

Chapter 3

Drawing and Transforming Triangles

Chapter 2, "Your First Step with WebGL," explained the basic approach to drawing WebGL graphics. You saw how to retrieve the WebGL context and clear a `<canvas>` in preparation for drawing your 2D/3DCG. You then explored the roles and features of the vertex and fragment shaders and how to actually draw graphics with them. With this basic structure in mind, you then constructed several sample programs that drew simple shapes composed of points on the screen.

This chapter builds on those basics by exploring how to draw more complex shapes and how to manipulate those shapes in 3D space. In particular, this chapter looks at

- The critical role of triangles in 3DCG and WebGL's support for drawing triangles
- Using multiple triangles to draw other basic shapes
- Basic transformations that move, rotate, and scale triangles using simple equations
- How matrix operations make transformations simple

By the end of this chapter, you will have a comprehensive understanding of WebGL's support for drawing basic shapes and how to use matrix operations to manipulate those shapes. Chapter 4, "More Transformations and Basic Animation," then builds on this knowledge to explore simple animations.

Drawing Multiple Points

As you are probably aware, 3D models are actually made from a simple building block: the humble triangle. For example, looking at the frog in Figure 3.1, the figure on the right side shows the triangles used to make up the shape, and in particular the three vertices that make up one triangle of the head. So, although this game character has a complex shape, its basic components are the same as a simple one, except of course for many more triangles and their associated vertices. By using smaller and smaller triangles, and therefore more and more vertices, you can create more complex or smoother objects. Typically, a complex shape or game character will consist of tens of thousands of triangles and their associated vertices. Thus, multiple vertices used to make up triangles are pivotal for drawing 3D objects.

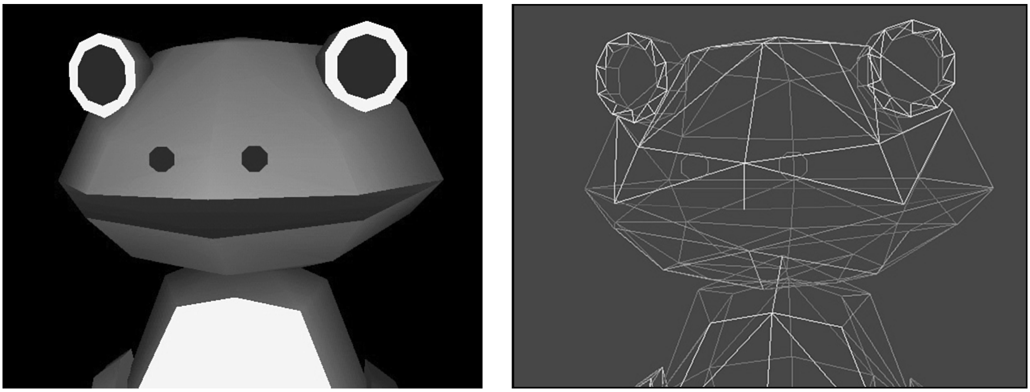


Figure 3.1 Complex characters are also constructed from multiple triangles

In this section, you explore the process of drawing shapes using multiple vertices. However, to keep things simple, you'll continue to use 2D shapes, because the technique to deal with multiple vertices for a 2D shape is the same as dealing with them for a 3D object. Essentially, if you can master these techniques for 2D shapes, you can easily understand the examples in the rest of this book that use the same techniques for 3D objects.

As an example of handling multiple vertices, let's create a program, `MultiPoint`, that draws three red points on the screen; remember, three points or vertices make up the triangle. Figure 3.2 shows a screenshot from `Multipoint`.

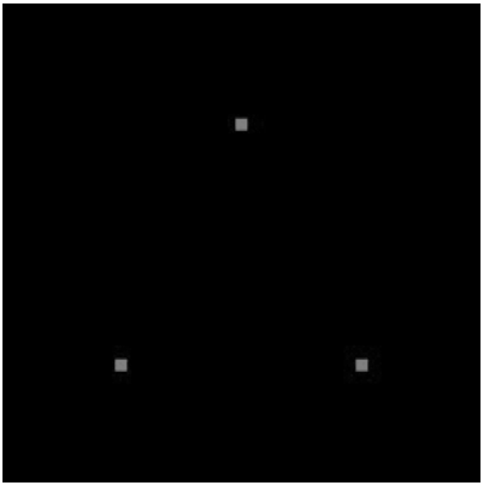


Figure 3.2 `MultiPoint`

In the previous chapter, you created a sample program, `ClickedPoints`, that drew multiple points based on mouse clicks. `ClickedPoints` stored the position of the points in a JavaScript array (`g_points[]`) and used the `gl.drawArrays()` method to draw each point (Listing 3.1). To draw multiple points, you used a loop that iterated through the array, drawing each point in turn by passing one vertex at a time to the shader.

Listing 3.1 Drawing Multiple Points as Shown in ClickedPoints.js (Chapter 2)

```
for(var i = 0; i<len; i+=2) {
  // Pass the position of a point to a_Position variable
  gl.vertexAttrib3f(a_Position, g_points[i], g_points[i+1], 0.0);

  // Draw a point
  gl.drawArrays(gl.POINTS, 0, 1);
}
```

Obviously, this method is useful only for single points. For shapes that use multiple vertices, you need a way to simultaneously pass multiple vertices to the vertex shader so that you can draw shapes constructed from multiple vertices, such as triangles, rectangles, and cubes.

WebGL provides a convenient way to pass multiple vertices and uses something called a **buffer object** to do so. A buffer object is a memory area that can store multiple vertices in the WebGL system. It is used both as a staging area for the vertex data and a way to simultaneously pass the vertices to a vertex shader.

Let's examine a sample program before explaining the buffer object so you can get a feel for the processing flow.

Sample Program (MultiPoint.js)

The processing flowchart for `MultiPoint.js` (see Figure 3.3) is basically the same as for `ClickedPoints.js` (Listing 2.7) and `ColoredPoints.js` (Listing 2.8), which you saw in Chapter 2. The only difference is a new step, setting up the positions of vertices, which is added to the previous flow.

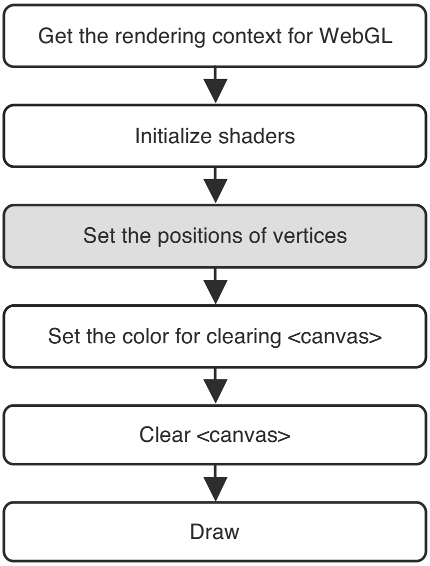


Figure 3.3 Processing flowchart for MultiPoints.js

This step is implemented at line 34, the function `initVertexBuffers()`, in Listing 3.2.

Listing 3.2 MultiPoint.js

```
1 // MultiPoint.js
2 // Vertex shader program
3 var VSHADER_SOURCE =
4   'attribute vec4 a_Position;\n' +
5   'void main() {\n' +
6   '  gl_Position = a_Position;\n' +
7   '  gl_PointSize = 10.0;\n' +
8   '}\n';
9
10 // Fragment shader program
    ...
15
16 function main() {
    ...
```

```
  // Get the rendering context for WebGL
  var gl = getWebGLContext(canvas);
     ...
  // Initialize shaders
  if (!initShaders(gl, VSHADER_SOURCE, FSHADER_SOURCE)) {
     ...
  }

  // Set the positions of vertices
  var n = initVertexBuffers(gl);
  if (n < 0) {
    console.log('Failed to set the positions of the vertices');
    return;
  }

  // Set the color for clearing <canvas>
     ...
  // Clear <canvas>
...
  // Draw three points
  gl.drawArrays(gl.POINTS, 0, n); // n is 3
}

function initVertexBuffers(gl) {
  var vertices = new Float32Array([
    0.0, 0.5,   -0.5, -0.5,   0.5, -0.5
  ]);
  var n = 3; // The number of vertices

  // Create a buffer object
  var vertexBuffer = gl.createBuffer();
  if (!vertexBuffer) {
    console.log('Failed to create the buffer object ');
    return -1;
  }

  // Bind the buffer object to target
  gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
  // Write date into the buffer object
  gl.bufferData(gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW);

  var a_Position = gl.getAttribLocation(gl.program, 'a_Position');
 ...
  // Assign the buffer object to a_Position variable
  gl.vertexAttribPointer(a_Position, 2, gl.FLOAT, false, 0, 0);
```

```
75
76   // Enable the assignment to a_Position variable
77   gl.enableVertexAttribArray(a_Position);
78
79   return n;
80 }
```

The new function `initVertexBuffers()` is defined at line 50 and used at line 34 to set up the vertex buffer object. The function stores multiple vertices in the buffer object and then completes the preparations for passing it to a vertex shader:

```
33 // Set the positions of vertices
34  var n = initVertexBuffers(gl);
```

The return value of this function is the number of vertices being drawn, stored in the variable `n`. Note that in case of error, `n` is negative.

As in the previous examples, the drawing operation is carried out using a single call to `gl.drawArrays()` at Line 48. This is similar to `ClickedPoints.js` except that `n` is passed as the third argument of `gl.drawArrays()` rather than the value 1:

```
46   // Draw three points
47   gl.drawArrays(gl.POINTS, 0, n); // n is 3
```

Because you are using a buffer object to pass multiple vertices to a vertex shader in `initVertexBuffers()`, you need to specify the number of vertices in the object as the third parameter of `gl.drawArrays()` so that WebGL then knows to draw a shape using all the vertices in the buffer object.

Using Buffer Objects

As indicated earlier, a buffer object is a mechanism provided by the WebGL system that provides a memory area allocated in the system (see Figure 3.4) that holds the vertices you want to draw. By creating a buffer object and then writing the vertices to the object, you can pass multiple vertices to a vertex shader through one of its attribute variables.

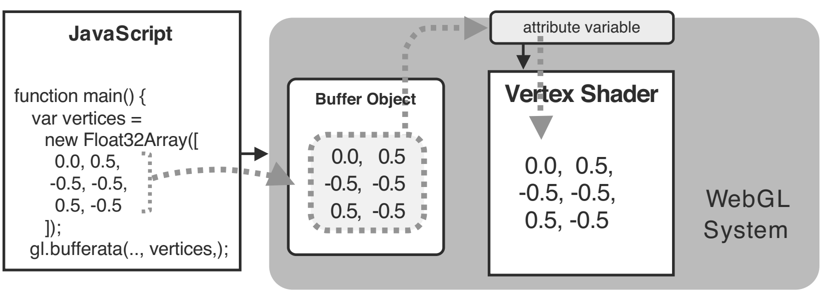


Figure 3.4 Passing multiple vertices to a vertex shader by using a buffer object

In the sample program, the data (vertex coordinates) written into a buffer object is defined as a special JavaScript array (`Float32Array`) as follows. We will explain this special array in detail later, but for now you can think of it as a normal array:

```
var vertices = new Float32Array([
  0.0, 0.5,   -0.5, -0.5,   0.5, -0.5
]);
```

There are five steps needed to pass multiple data values to a vertex shader through a buffer object. Because WebGL uses a similar approach when dealing with other objects such as texture objects (Chapter 4) and framebuffer objects (Chapter 8, "Lighting Objects"), let's explore these in detail so you will be able to apply the knowledge later:

1. Create a buffer object (`gl.createBuffer()`).
2. Bind the buffer object to a target (`gl.bindBuffer()`).
3. Write data into the buffer object (`gl.bufferData()`).
4. Assign the buffer object to an attribute variable (`gl.vertexAttribPointer()`).
5. Enable assignment (`gl.enableVertexAttribArray()`).

Figure 3.5 illustrates the five steps.

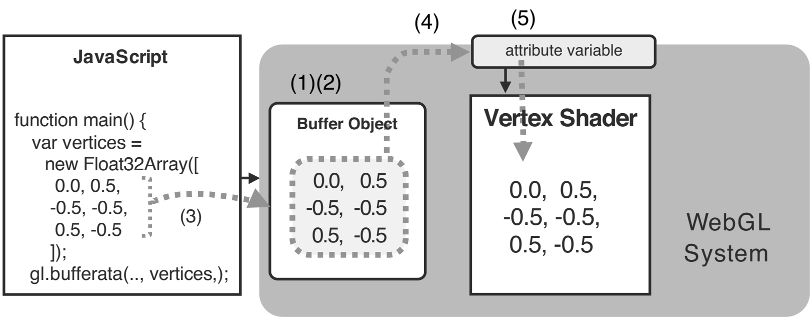


Figure 3.5 The five steps to pass multiple data values to a vertex shader using a buffer object

The code performing the steps in the sample program in Listing 3.2 is as follows:

```
// Create a buffer object                                   <- (1)
var vertexBuffer = gl.createBuffer();
if (!vertexBuffer) {
  console.log('Failed to create a buffer object');
  return -1;
}

// Bind the buffer object to a target                       <- (2)
```

```
64   gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
65   // Write date into the buffer object                          <- (3)
66   gl.bufferData(gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW);
67
68   var a_Position = gl.getAttribLocation(gl.program, 'a_Position');
    ...
73   // Assign the buffer object to a_Position variable             <- (4)
74   gl.vertexAttribPointer(a_Position, 2, gl.FLOAT, false, 0, 0);
75
76   // Enable the assignment to a_Position variable                <- (5)
77   gl.enableVertexAttribArray(a_Position);
```

Let's start with the first three steps (1–3), from creating a buffer object to writing data (vertex coordinates in this example) to the buffer, explaining the methods used within each step.

Create a Buffer Object (gl.createBuffer())

Before you can use a buffer object, you obviously need to create the buffer object. This is the first step, and it's carried out at line 57:

```
57   var vertexBuffer = gl.createBuffer();
```

You use the `gl.createBuffer()` method to create a buffer object within the WebGL system. Figure 3.6 shows the internal state of the WebGL system. The upper part of the figure shows the state before executing the method, and the lower part is after execution. As you can see, when the method is executed, it results in a single buffer object being created in the WebGL system. The keywords `gl.ARRAY_BUFFER` and `gl.ELEMENT_ARRAY_BUFFER` in the figure will be explained in the next section, so you can ignore them for now.

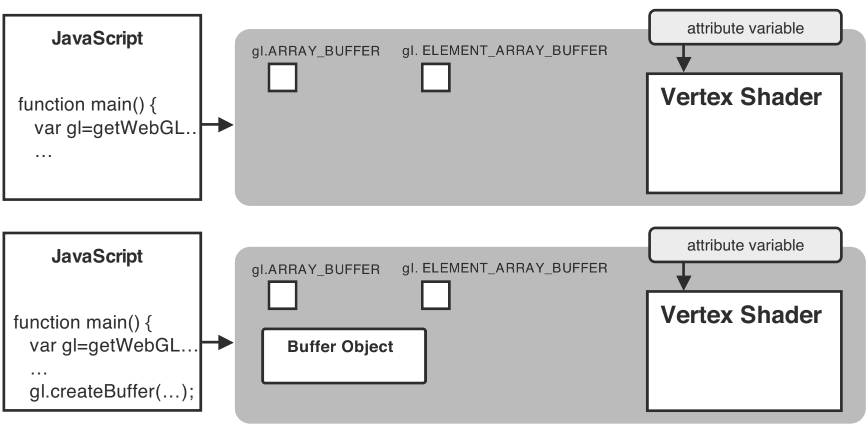


Figure 3.6 Create a buffer object

The following shows the specification of `gl.createBuffer()`.

`gl.createBuffer()`		
Create a buffer object.		
Return value	non-null	The newly created buffer object.
	null	Failed to create a buffer object.
Errors	None	

The corresponding method `gl.deleteBuffer()` deletes the buffer object created by `gl.createBuffer()`.

`gl.deleteBuffer``(buffer)`		
Delete the buffer object specified by *buffer*.		
Parameters	buffer	Specifies the buffer object to be deleted.
Return Value	None	
Errors	None	

Bind a Buffer Object to a Target (gl.bindBuffer())

After creating a buffer object, the second step is to bind it to a "target." The target tells WebGL what type of data the buffer object contains, allowing it to deal with the contents correctly. This binding process is carried out at line 64 as follows:

```
64 gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
```

The specification of `gl.bindBuffer()` is as follows.

`gl.bindBuffer(target, buffer)`		
Enable the buffer object specified by *buffer* and bind it to the *target*.		
Parameters	Target can be one of the following:	
	`gl.ARRAY_BUFFER`	Specifies that the buffer object contains vertex data.

	`gl.ELEMENT_ARRAY_BUFFER`	Specifies that the buffer object contains index values pointing to vertex data. (See Chapter 6, "The OpenGL ES Shading Language [GLSL ES].)
	buffer	Specifies the buffer object created by a previous call to `gl.createBuffer()`. When `null` is specified, binding to the *target* is disabled.
Return Value	None	
Errors	INVALID_ENUM	*target* is none of the above values. In this case, the current binding is maintained.

In the sample program in this section, `gl.ARRAY_BUFFER` is specified as the *target* to store vertex data (positions) in the buffer object. After executing line 64, the internal state in the WebGL system changes, as shown in Figure 3.7.

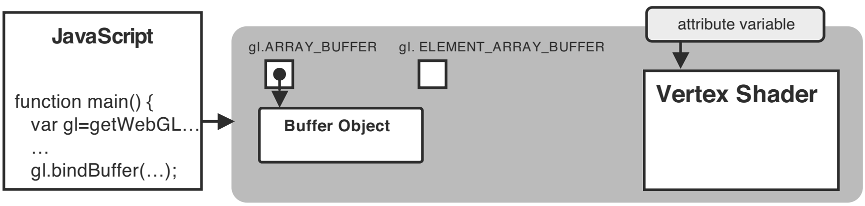


Figure 3.7 Bind a buffer object to a target

The next step is to write data into the buffer object. Note that because you won't be using the `gl.ELEMENT_ARRAY_BUFFER` until Chapter 6, it'll be removed from the following figures for clarity.

Write Data into a Buffer Object (gl.bufferData())

Step 3 allocates storage and writes data to the buffer. You use `gl.bufferData()` to do this, as shown at line 66:

```
66   gl.bufferData(gl.ARRAY_BUFFER, vertices, gl.STATIC_DRAW);
```

This method writes the data specified by the second parameter (`vertices`) into the buffer object bound to the first parameter (`gl.ARRAY_BUFFER`). After executing line 66, the internal state of the WebGL system changes, as shown in Figure 3.8.

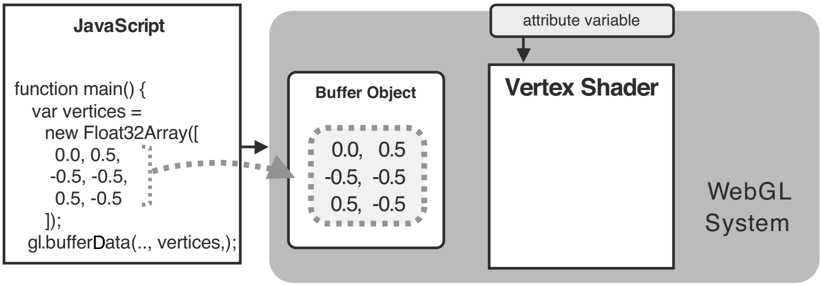


Figure 3.8 Allocate storage and write data into a buffer object

You can see in this figure that the vertex data defined in your JavaScript program is written to the buffer object bound to `gl.ARRAY_BUFFER`. The following table shows the specification of `gl.bufferData()`.

`gl.bufferData(target, data, usage)`

Allocate storage and write the data specified by *data* to the buffer object bound to *target*.

Parameters	target	Specifies `gl.ARRAY_BUFFER` or `gl.ELEMENT_ARRAY_BUFFER`.	
	data	Specifies the data to be written to the buffer object (typed array; see the next section).	
	usage	Specifies a hint about how the program is going to use the data stored in the buffer object. This hint helps WebGL optimize performance but will not stop your program from working if you get it wrong.	
		`gl.STATIC_DRAW`	The buffer object data will be specified once and used many times to draw shapes.
		`gl.STREAM_DRAW`	The buffer object data will be specified once and used a few times to draw shapes.
		`gl.DYNAMIC_DRAW`	The buffer object data will be specified repeatedly and used many times to draw shapes.
Return value	None		
Errors	INVALID_ENUM	*target* is none of the preceding constants	

Now, let us examine what data is passed to the buffer object using `gl.bufferData()`. This method uses the special array `vertices` mentioned earlier to pass data to the vertex shader. The array is created at line 51 using the `new` operator with the data arranged as <x coordinate and y coordinate of the first vertex>, <x coordinate and y coordinate of the second vertex>, and so on:

```
51   var vertices = new Float32Array([
52     0.0, 0.5,   -0.5, -0.5,   0.5, -0.5
53   ]);
54   var n = 3; // The number of vertices
```

As you can see in the preceding code snippet, you are using the `Float32Array` object instead of the more usual JavaScript `Array` object to store the data. This is because the standard array in JavaScript is a general-purpose data structure able to hold both numeric data and strings but isn't optimized for large quantities of data of the same type, such as `vertices`. To address this issue, the typed array, of which one example is `Float32Array`, has been introduced.

Typed Arrays

WebGL often deals with large quantities of data of the same type, such as vertex coordinates and colors, for drawing 3D objects. For optimization purposes, a special type of array (**typed array**) has been introduced for each data type. Because the type of data in the array is known in advance, it can be handled efficiently.

`Float32Array` at line 51 is an example of a typed array and is generally used to store vertex coordinates or colors. It's important to remember that a typed array is expected by WebGL and is needed for many operations, such as the second parameter *data* of `gl.bufferData()`.

Table 3.1 shows the different typed arrays available. The third column shows the corresponding data type in C as a reference for those of you familiar with the C language.

Table 3.1 Typed Arrays Used in WebGL

Typed Array	Number of Bytes per Element	Description (C Types)
`Int8Array`	1	8-bit signed integer (signed char)
`Uint8Array`	1	8-bit unsigned integer (unsigned char)
`Int16Array`	2	16-bit signed integer (signed short)
`Uint16Array`	2	16-bit unsigned integer (unsigned short)
`Int32Array`	4	32-bit signed integer (signed int)
`Uint32Array`	4	32-bit unsigned integer (unsigned int)
`Float32Array`	4	32-bit floating point number (float)
`Float64Array`	8	64-bit floating point number (double)

Like JavaScript, these typed arrays have a set of methods, a property, and a constant available that are shown in Table 3.2. Note that, unlike the standard `Array` object in JavaScript, the methods `push()` and `pop()` are not supported.

Table 3.2 Methods, Property, Constant of Typed Arrays

Methods, Properties, and Constants	Description
`get(index)`	Get the *index*-th element
`set(index, value)`	Set *value* to the *index*-th element
`set(array, offset)`	Set the elements of *array* from *offset*-th element
`length`	The length of the array
`BYTES_PER_ELEMENT`	The number of bytes per element in the array

Just like standard arrays, the `new` operator creates a typed array and is passed the array data. For example, to create `Float32Array` vertices, you could pass the array `[0.0, 0.5, -0.5, -0.5, 0.5, -0.5]`, which represents a set of vertices. Note that the only way to create a typed array is by using the `new` operator. Unlike the `Array` object, the `[]` operator is not supported:

```
51   var vertices = new Float32Array([
52     0.0, 0.5,   -0.5, -0.5,   0.5, -0.5
53   ]);
```

In addition, just like a normal JavaScript array, an empty typed array can be created by specifying the number of elements of the array as an argument. For example:

```
var vertices = new Float32Array(4);
```

With that, you've completed the first three steps of the process to set up and use a buffer (that is, creating a buffer object in the WebGL system, binding the buffer object to a target, and then writing data into the buffer object). Let's now look at how to actually use the buffer, which takes place in steps 4 and 5 of the process.

Assign the Buffer Object to an Attribute Variable (gl.vertexAttribPointer())

As explained in Chapter 2, you can use `gl.vertexAttrib[1234]f()` to assign data to an attribute variable. However, these methods can only be used to assign a single data value to an attribute variable. What you need here is a way to assign an array of values—the vertices in this case—to an attribute variable.

`gl.vertexAttribPointer()` solves this problem and can be used to assign a buffer object (actually a reference or handle to the buffer object) to an attribute variable. This can be seen at line 74 when you assign a buffer object to the attribute variable `a_Position`:

```
74   gl.vertexAttribPointer(a_Position, 2, gl.FLOAT, false, 0, 0);
```

The specification of `gl.vertexAttribPointer()` is as follows.

`gl.vertexAttribPointer(location, size, type, normalized, stride, offset)`

Assign the buffer object bound to `gl.ARRAY_BUFFER` to the attribute variable specified by *location*.

Parameters	location	Specifies the storage location of an attribute variable.
	size	Specifies the number of components per vertex in the buffer object (valid values are 1 to 4). If *size* is less than the number of components required by the attribute variable, the missing components are automatically supplied just like `gl.vertexAttrib[1234]f()`. For example, if *size* is 1, the second and third components will be set to 0, and the fourth component will be set to 1.
	type	Specifies the data format using one of the following:

`gl.UNSIGNED_BYTE`	unsigned byte	for `Uint8Array`
`gl.SHORT`	signed short integer	for `Int16Array`
`gl.UNSIGNED_SHORT`	unsigned short integer	for `Uint16Array`
`gl.INT`	signed integer	for `Int32Array`
`gl.UNSIGNED_INT`	unsigned integer	for `Uint32Array`
`gl.FLOAT`	floating point number	for `Float32Array`

	normalized	Either `true` or `false` to indicate whether nonfloating data should be normalized to [0, 1] or [–1, 1].
	stride	Specifies the number of bytes between different vertex data elements, or zero for default stride (see Chapter 4).
	offset	Specifies the offset (in bytes) in a buffer object to indicate what number-th byte the vertex data is stored from. If the data is stored from the beginning, *offset* is 0.
Return value	None	

Errors	INVALID_OPERATION	There is no current program object.
	INVALID_VALUE	*location* is greater than or equal to the maximum number of attribute variables (8, by default). *stride* or *offset* is a negative value.

So, after executing this fourth step, the preparations are nearly completed in the WebGL system for using the buffer object at the attribute variable specified by *location*. As you can see in Figure 3.9, although the buffer object has been assigned to the attribute variable, WebGL requires a final step to "enable" the assignment and make the final connection.

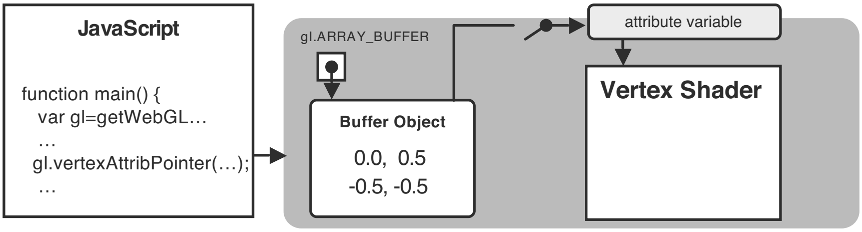


Figure 3.9 Assign a buffer object to an attribute variable

The fifth and final step is to enable the assignment of the buffer object to the attribute variable.

Enable the Assignment to an Attribute Variable (gl.enableVertexAttribArray())

To make it possible to access a buffer object in a vertex shader, we need to enable the assignment of the buffer object to an attribute variable by using `gl.enableVertexAttribArray()` as shown in line 77:

```
77   gl.enableVertexAttribArray(a_Position);
```

The following shows the specification of `gl.enableVertexAttribArray()`. Note that we are using the method to handle a buffer even though the method name suggests it's only for use with "vertex arrays." This is not a problem and is simply a legacy from OpenGL.

`gl.enableVertexAttribArray(location)`

Enable the assignment of a buffer object to the attribute variable specified by *location*.

Parameters	location	Specifies the storage location of an attribute variable.
Return value	None	
Errors	INVALID_VALUE	*location* is greater than or equal to the maximum number of attribute variables (8 by default).

When you execute `gl.enableVertexAttribArray()` specifying an attribute variable that has been assigned a buffer object, the assignment is enabled, and the unconnected line is connected as shown in Figure 3.10.

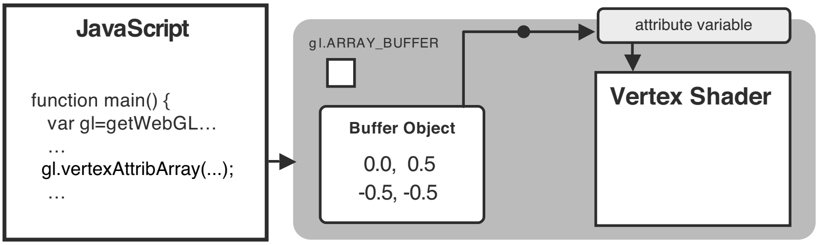


Figure 3.10 Enable the assignment of a buffer object to an attribute variable

You can also break this assignment (disable it) using the method `gl.disableVertexAttribArray()`.

`gl.disableVertexAttribArray(location)`

Disable the assignment of a buffer object to the attribute variable specified by *location*.

Parameters	location	Specifies the storage location of an attribute variable.
Return Value	None	
Errors	INVALID_VALUE	*location* is greater than or equal to the maximum number of attribute variables (8 by default).

Now, everything is set! All you need to do is run the vertex shader, which draws the points using the vertex coordinates specified in the buffer object. As in Chapter 2, you will use the method `gl.drawArrays`, but because you are drawing multiple points, you will actually use the second and third parameters of `gl.drawArrays()`.

Note that after enabling the assignment, you can no longer use `gl.vertexAttrib[1234]f()` to assign data to the attribute variable. You have to explicitly disable the assignment of a buffer object. You can't use both methods simultaneously.

The Second and Third Parameters of gl.drawArrays()

Before entering into a detailed explanation of these parameters, let's take a look at the specification of `gl.drawArrays()` that was introduced in Chapter 2. Following is a recap of the method with only the relevant parts of the specification shown.

gl.drawArrays(mode, first, count)

Execute a vertex shader to draw shapes specified by the *mode* parameter.

Parameters	mode	Specifies the type of shape to be drawn. The following symbolic constants are accepted: `gl.POINTS`, `gl.LINES`, `gl.LINE_STRIP`, `gl.LINE_LOOP`, `gl.TRIANGLES`, `gl.TRIANGLE_STRIP`, and `gl.TRIANGLE_FAN`.
	first	Specifies what number-th vertex is used to draw from (integer).
	count	Specifies the number of vertices to be used (integer).

In the sample program this method is used as follows:

```
47   gl.drawArrays(gl.POINTS, 0, n); // n is 3
```

As in the previous examples, because you are simply drawing three points, the first parameter is still `gl.POINTS`. The second parameter *first* is set to 0 because you want to draw from the first coordinate in the buffer. The third parameter *count* is set to 3 because you want to draw three points (in line 47, `n` is 3).

When your program runs line 47, it actually causes the vertex shader to be executed *count* (three) times, sequentially passing the vertex coordinates stored in the buffer object via the attribute variable into the shader (Figure 3.11).

Note that for each execution of the vertex shader, 0.0 and 1.0 are automatically supplied to the z and w components of `a_Position` because `a_Position` requires four components (`vec4`) and you are supplying only two.

Remember that at line 74, the second parameter *size* of `gl.vertexAttribPointer()` is set to 2. As just discussed, the second parameter indicates how many coordinates per vertex are specified in the buffer object and, because you are only specifying the x and y coordinates in the buffer, you set the size value to 2:

```
74   gl.vertexAttribPointer(a_Position, 2, gl.FLOAT, false, 0, 0);
```

After drawing all points, the content of the color buffer is automatically displayed in the browser (bottom of Figure 3.11), resulting in our three red points, as shown in Figure 3.2.

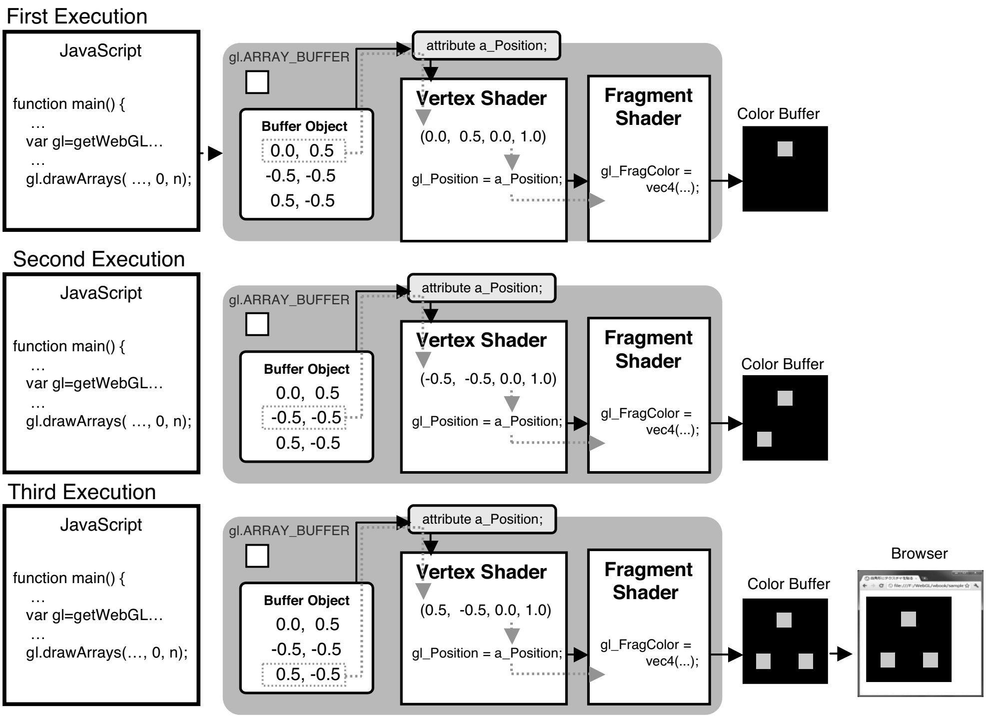


Figure 3.11 How the data in a buffer object is passed to a vertex shader during execution

Experimenting with the Sample Program

Let's experiment with the sample program to better understand how `gl.drawArrays()` works by modifying the second and third parameters. First, let's specify 1 as the third argument for *count* at line 47 instead of our variable `n` (set to 3) as follows:

```
47  gl.drawArrays(gl.POINTS, 0, 1);
```

In this case, the vertex shader is executed only once, and a single point is drawn using the first vertex in the buffer object.

If you now specify 1 as the second argument, only the second vertex is used to draw a point. This is because you are telling WebGL that you want to start drawing from the second vertex and you only want to draw one vertex. So again, you will see only a single point, although this time it is the second vertex coordinates that are shown in the browser:

```
47  gl.drawArrays(gl.POINTS, 1, 1);
```

This gives you a quick feel for the role of the parameters *first* and *count*. However, what will be happen if you change the first parameter *mode*? The next section explores the first parameter in more detail.

Hello Triangle

Now that you've learned the basic techniques to pass multiple vertex coordinates to a vertex shader, let's try to draw other shapes using multiple vertex coordinates. This section uses a sample program `HelloTriangle`, which draws a single 2D triangle. Figure 3.12 shows a screenshot of `HelloTriangle`.

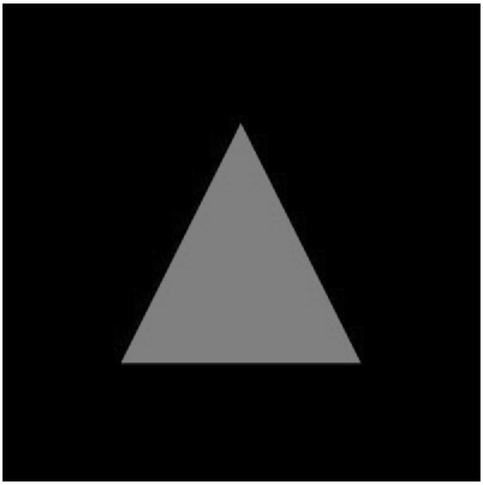


Figure 3.12 HelloTriangle

Sample Program (HelloTriangle.js)

Listing 3.3 shows `HelloTriangle.js`, which is almost identical to `MultiPoint.js` used in the previous section with two critical differences.

Listing 3.3 HelloTriangle.js

```
// HelloTriangle.js
// Vertex shader program
var VSHADER_SOURCE =
  'attribute vec4 a_Position;\n' +
  'void main() {\n' +
  '  gl_Position = a_Position;\n' +
  '}\n';

```

```
 9 // Fragment shader program
10 var FSHADER_SOURCE =
11   'void main() {\n' +
12   '  gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);\n' +
13   '}\n';
14
15 function main() {
    ...
19   // Get the rendering context for WebGL
20   var gl = getWebGLContext(canvas);
     ...
26   // Initialize shaders
27   if (!initShaders(gl, VSHADER_SOURCE, FSHADER_SOURCE)) {
     ...
30   }
31
32   // Set the positions of vertices
33   var n = initVertexBuffers(gl);
...
39   // Set the color for clearing <canvas>
     ...
45   // Draw a triangle
46   gl.drawArrays(gl.TRIANGLES, 0, n);
47 }
48
49 function initVertexBuffers(gl) {
50   var vertices = new Float32Array([
51     0.0, 0.5,   -0.5, -0.5,   0.5, -0.5
52   ]);
53   var n = 3; // The number of vertices
     ...
78   return n;
79 }
```

The two differences from `MultiPoint.js` are

- The line to specify the size of a point `gl_PointSize = 10.0;` has been removed from the vertex shader. This line only has an effect when you are drawing a point.
- The first parameter of `gl.drawArrays()` has been changed from `gl.POINTS` to `gl.TRIANGLES` at line 46.

The first parameter, *mode,* of `gl.drawArrays()` is powerful and provides the ability to draw various shapes. Let's take a look.

Basic Shapes

By changing the argument we use for the first parameter, *mode,* of `gl.drawArrays()`, we can change the meaning of line 46 into "execute the vertex shader three times (`n` is 3), and draw a triangle using the three vertices in the buffer, starting from the first vertex coordinate":

```
46   gl.drawArrays(gl.TRIANGLES, 0, n);
```

In this case, the three vertices in the buffer object are no longer individual points, but become three vertices of a triangle.

The WebGL method `gl.drawArrays()` is both powerful and flexible, allowing you to specify seven different types of basic shapes as the first argument. These are explained in more detail in Table 3.3. Note that v0, v1, v2 ... indicates the vertices specified in a buffer object. The order of vertices affects the drawing of the shape.

The shapes in the table are the only ones that WebGL can draw directly, but they are the basics needed to construct complex 3D graphics. (Remember the frog at the start of this chapter.)

Table 3.3 Basic Shapes Available in WebGL

Basic Shape	Mode	Description
Points	`gl.POINTS`	A series of points. The points are drawn at v0, v1, v2 ...
Line segments	`gl.LINES`	A series of unconnected line segments. The individual lines are drawn between vertices given by (v0, v1), (v2, v3), (v4, v5)... If the number of vertices is odd, the last one is ignored.
Line strips	`gl.LINE_STRIP`	A series of connected line segments. The line segments are drawn between vertices given by (v0, v1), (v1, v2), (v2, v3), ... The first vertex becomes the start point of the first line, the second vertex becomes the end point of the first line and the start point of the second line, and so on. The *i*-th (*i*> 1) vertex becomes the start point of the *i*-th line and the end point of the *i-1*-th line. (The last vertex becomes the end point of the last line.)
Line loops	`gl.LINE_LOOP`	A series of connected line segments. In addition to the lines drawn by `gl.LINE_STRIP`, the line between the last vertex and the first vertex is drawn. The line segments drawn are (v0, v1), (v1, v2), ..., and (v*n*, v0). v*n* is the last vertex.
Triangles	`gl.TRIANGLES`	A series of separate triangles. The triangles given by vertices (v0, v1, v2), (v3, v4, v5), ... are drawn. If the number of vertices is not a multiple of 3, the remaining vertices are ignored.

Basic Shape	Mode	Description
Triangle strips	`gl.TRIANGLE_STRIP`	A series of connected triangles in strip fashion. The first three vertices form the first triangle and the second triangle is formed from the next vertex and one of the sides of the first triangle. The triangles are drawn given by (v0, v1, v2), (v2, v1, v3), (v2, v3, v4) ... (Pay attention to the order of vertices.)
Triangle fans	`gl.TRIANGLE_FAN`	A series of connected triangles sharing the first vertex in fan-like fashion. The first three vertices form the first triangle and the second triangle is formed from the next vertex, one of the sides of the first triangle, and the first vertex. The triangles are drawn given by (v0, v1, v2), (v0, v2, v3), (v0, v3, v4), ...

Figure 3.13 shows these basic shapes.

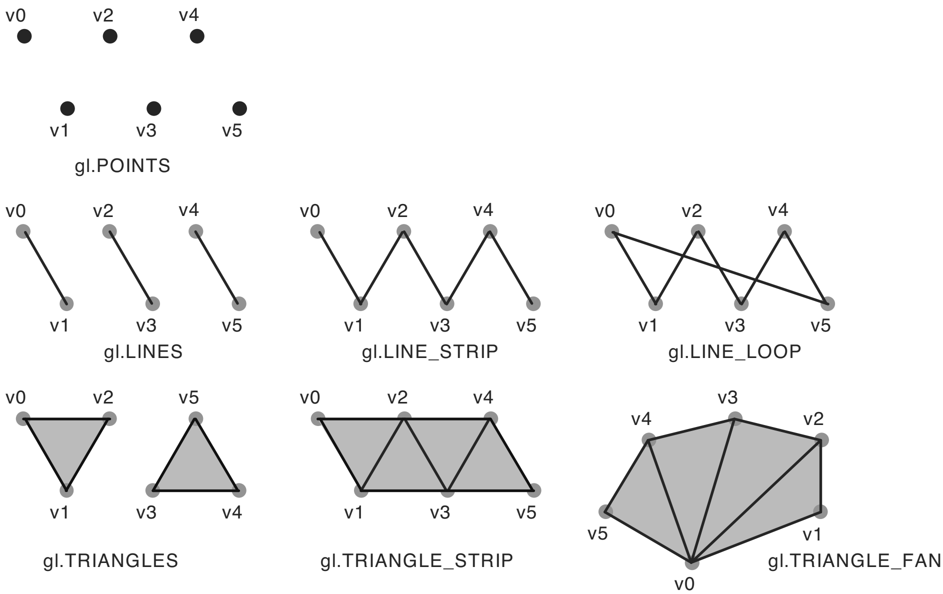


Figure 3.13 Basic shapes available in WebGL

As you can see from the figure, WebGL can draw only three types of shapes: a point, a line, and a triangle. However, as explained at the beginning of this chapter, spheres to cubes to 3D monsters to humanoid characters in a game can be constructed from small triangles. Therefore, you can use these basic shapes to draw anything.

Experimenting with the Sample Program

To examine what will happen when using `gl.LINES`, `gl.LINE_STRIP`, and `gl.LINE_LOOP`, let's change the first argument of `gl.drawArrays()` as shown next. The name of each sample program is `HelloTriangle_LINES`, `HelloTriangle_LINE_STRIP`, and `HelloTriangle_LINE_LOOP`, respectively:

```
gl.drawArrays(gl.LINES, 0, n);
gl.drawArrays(gl.LINE_STRIP, 0, n);
gl.drawArrays(gl.LINE_LOOP, 0, n);
```

Figure 3.14 shows a screenshot of each program.

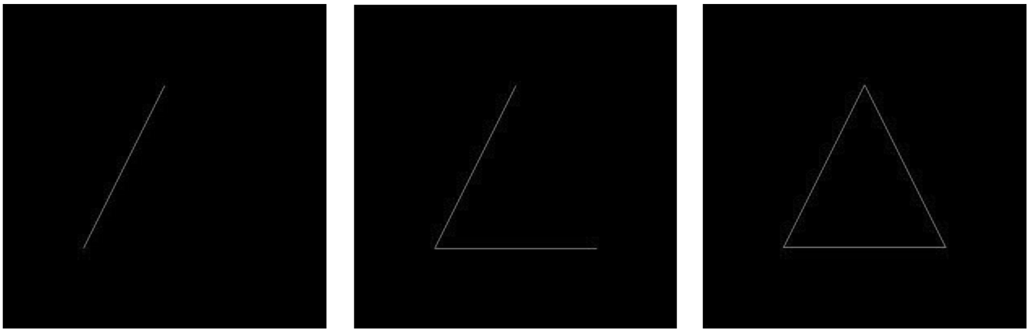


Figure 3.14 gl.LINES, gl.LINE_STRIP, and gl.LINE_LOOP

As you can see, `gl.LINES` draws a line using the first two vertices and does not use the last vertex, whereas `gl.LINE_STRIP` draws two lines using the first three vertices. Finally, `gl.LINE_LOOP` draws the lines in the same manner as `gl.LINE_STRIP` but then "loops" between the last vertex and the first vertex and makes a triangle.

Hello Rectangle (HelloQuad)

Let's use this basic way of drawing triangles to draw a rectangle. The name of the sample program is `HelloQuad`, and Figure 3.15 shows a screenshot when it's loaded into your browser.

Figure 3.16 shows the vertices of the rectangle. Of course, the number of vertices is four because it is a rectangle. As explained in the previous section, WebGL cannot draw a rectangle directly, so you need to divide the rectangle into two triangles (v0, v1, v2) and (v2, v1, v3) and then draw each one using `gl.TRIANGLES`, `gl.TRIANGLE_STRIP`, or `gl.TRIANGLE_FAN`. In this example, you'll use `gl.TRIANGLE_STRIP` because it only requires you to specify four vertices. If you were to use `gl.TRIANGLES`, you would need to specify a total of six.

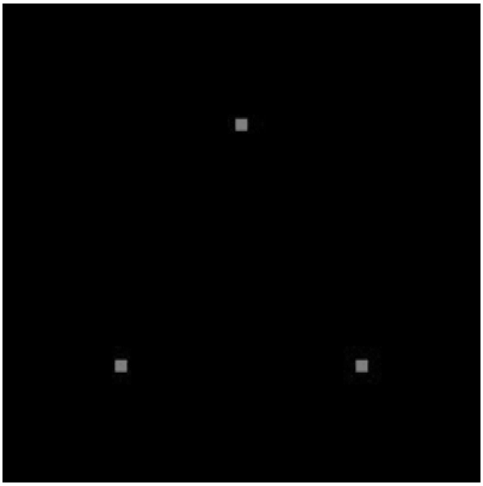


Figure 3.15 HelloQuad

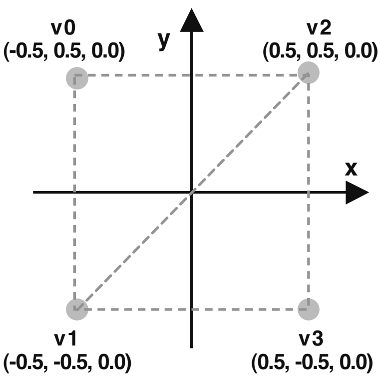


Figure 3.16 The four vertex coordinates of the rectangle

Basing the example on `HelloTriangle.js`, you need to add an extra vertex coordinate at line 50. Pay attention to the order of vertices; otherwise, the draw command will not execute correctly:

```
50   var vertices = new Float32Array([
51     -0.5, 0.5,   -0.5, -0.5,   0.5, 0.5,   0.5, -0.5
52   ]);
```

Because you've added a fourth vertex, you need to change the number of vertices from 3 to 4 at line 53:

```
53   var n = 4; // The number of vertices
```

Then, by modifying line 46 as follows, your program will draw a rectangle in the browser:

```
46   gl.drawArrays(gl.TRIANGLE_STRIP, 0, n);
```

Experimenting with the Sample Program

Now that you have a feel for how to use `gl.TRIANGLE_STRIP`, let's change the first parameter of `gl.drawArrays()` to `gl.TRIANGLE_FAN`. The name of the sample program is `HelloQuad_FAN`:

```
46   gl.drawArrays(gl.TRIANGLE_FAN, 0, n);
```

Figure 3.17 show a screenshot of `HelloQuad_FAN`. In this case, we can see the ribbon-like shape on the screen.

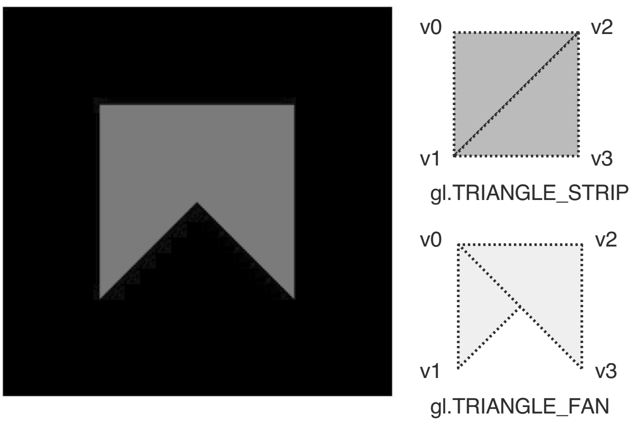


Figure 3.17 HelloQuad_FAN

Looking at the order of vertices and the triangles drawn by `gl.TRIANGLE_FAN` shown on the right side of Figure 3.17, you can see why the result became a ribbon-like shape. Essentially, `gl.TRIANGLE_FAN` causes WebGL to draw a second triangle that shares the first vertex (v0), and this second triangle overlaps the first, creating the ribbon-like effect.

Moving, Rotating, and Scaling

Now that you understand the basics of drawing shapes like triangles and rectangles, let's take another step and try to move (translate), rotate, and scale the triangle and display the results on the screen. These operations are called **transformations (affine transformations)**. This section introduces some math to explain each transformation and help you to understand how each operation can be realized. However, when you write your own programs, you don't need the math; instead, you can use one of several convenient libraries, explained in the next section, that handle the math for you.

If you find reading this section and in particular the math too much on first read, it's okay to skip it and return later. Or, if you already know that transformations can be written using a matrix, you can skip this section as well.

First, let's write a sample program, `TranslatedTriangle`, that moves a triangle 0.5 units to the right and 0.5 units up. You can use the triangle you drew in the previous section. The right direction means the positive direction of the x-axis, and the up direction means the positive direction of the y-axis. (See the coordinate system in Chapter 2.) Figure 3.18 shows `TranslatedTriangle`.

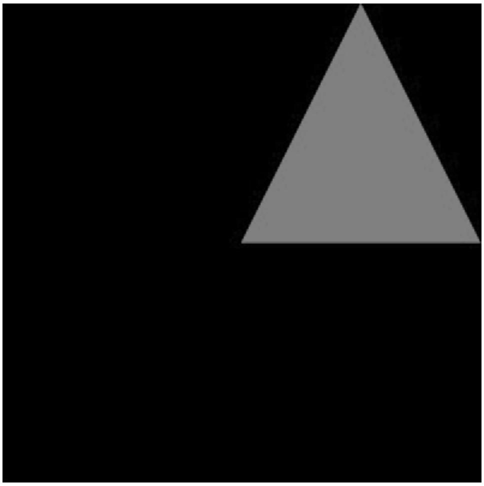


Figure 3.18 TranslatedTriangle

Translation

Let us examine what kind of operations you need to apply to each vertex coordinate of a shape to translate (move) the shape. Essentially, you just need to add a translation distance for each direction (x and y) to each component of the coordinates. Looking at Figure 3.19, the goal is to translate the point p (x, y, z) to the point p' (x', y', z'), so the translation distance for the x, y, and z direction is `Tx`, `Ty`, and `Tz`, respectively. In this figure, `Tz` is 0.

To determine the coordinates of p', you simply add the T values, as shown in Equation 3.1.

Equation 3.1

$$x' = x + Tx$$

$$y' = y + Ty$$

$$z' = z + Tz$$

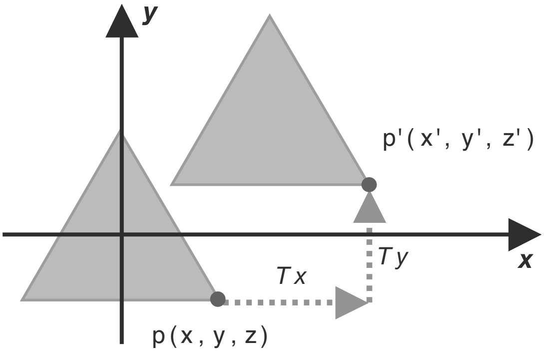


Figure 3.19 Calculating translation distances

These simple equations can be implemented in a WebGL program just by adding each constant value to each vertex coordinate. You've probably realized already that because they are a **per-vertex operation**, you need to implement the operations in a vertex shader. Conversely, they clearly aren't a per-fragment operation, so you don't need to worry about the fragment shader.

Once you understand this explanation, implementation is easy. You need to pass the translation distances `Tx`, `Ty`, and `Tz` to the vertex shader, apply Equation 3.1 using the distances, and then assign the result to `gl_Position`. Let's look at a sample program that does this.

Sample Program (TranslatedTriangle.js)

Listing 3.4 shows `TranslatedTriangle.js`, in which the vertex shader is partially modified to carry out the translation operation. However, the fragment shader is the same as in `HelloTriangle.js` in the previous section. To support the modification to the vertex shader, some extra code is added to the `main()` function in the JavaScript.

Listing 3.4 TranslatedTriangle.js

```
// TranslatedTriangle.js
// Vertex shader program
var VSHADER_SOURCE =
  'attribute vec4 a_Position;\n' +
  'uniform vec4 u_Translation;\n' +
  'void main() {\n' +
  '  gl_Position = a_Position + u_Translation;\n' +
  '}\n';

// Fragment shader program
   ...
```

```
16 // The translation distance for x, y, and z direction
17 var Tx = 0.5, Ty = 0.5, Tz = 0.0;
18
19 function main() {
    ...
23   // Get the rendering context for WebGL
24   var gl = getWebGLContext(canvas);
    ...
30   // Initialize shaders
31   if (!initShaders(gl, VSHADER_SOURCE, FSHADER_SOURCE)) {
    ...
34   }
35
36   // Set the positions of vertices
37   var n = initVertexBuffers(gl);
    ...
43   // Pass the translation distance to the vertex shader
44   var u_Translation = gl.getUniformLocation(gl.program, 'u_Translation');
    ...
49   gl.uniform4f(u_Translation, Tx, Ty, Tz, 0.0);
50
51   // Set the color for clearing <canvas>
    ...
57   // Draw a triangle
58   gl.drawArrays(gl.TRIANGLES, 0, n);
59 }
60
61 function initVertexBuffers(gl) {
62   var vertices = new Float32Array([
63     0.0.0, 0.5,   -0.5, -0.5,   0.5, -0.5
64   ]);
65   var n = 3; // The number of vertices
    ...
90   return n;
93 }
```

First, let's examine `main()` in JavaScript. Line 17 defines the variables for each translation distance of Equation 3.1:

```
17 var Tx = 0.5, Ty = 0.5, Tz = 0.0;
```

Because `Tx`, `Ty`, and `Tz` are fixed (uniform) values for all vertices, you use the uniform variable `u_Translation` to pass them to a vertex shader. Line 44 retrieves the storage location of the uniform variable, and line 49 assigns the data to the variable:

```
var u_Translation = gl.getUniformLocation(gl.program, 'u_Translation');
...
gl.uniform4f(u_Translation, Tx, Ty, Tz, 0.0);
```

Note that `gl.uniform4f()` requires a homogenous coordinate, so we supply a fourth argument (w) of 0.0. This will be explained in more detail later in this section.

Now, let's take a look at the vertex shader that uses this translation data. As you can see, the uniform variable `u_Translation` in the shader, to which the translation distances are passed, is defined as type `vec4` at line 5. This is because you want to add the components of `u_Translation` to the vertex coordinates passed to `a_Position` (as defined by Equation 3.1) and then assign the result to the variable `gl_Position`, which has type `vec4`. Remember, per Chapter 2, that the assignment operation in GLSL ES is only allowed between variables of the same types:

```
4   'attribute vec4 a_Position;\n' +
5   'uniform vec4 u_Translation;\n' +
6   'void main() {\n' +
7   '  gl_Position = a_Position + u_Translation;\n' +
8   '}\n';
```

After these preparations have been completed, the rest of tasks are straightforward. To calculate Equation 3.1 within the vertex shader, you just add each translation distance (Tx, Ty, Tz) passed in `u_Translation` to each vertex coordinate (x, y, z) passed in `a_Position`.

Because both variables are of type `vec4`, you can use the `+` operator, which will actually add the four components simultaneously (see Figure 3.20). This easy addition of vectors is a feature of GLSL ES and will be explained in more detail in Chapter 6.

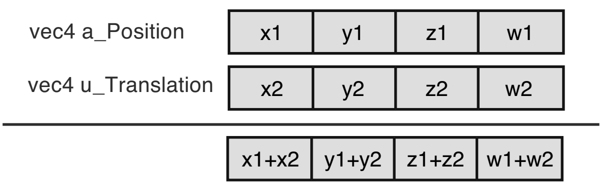


Figure 3.20 Addition of vec4 variables

Now, we'll return to the fourth element, (w), of the vector. As explained in Chapter 2, you need to specify the homogeneous coordinate to `gl_Position`, which is a four-dimensional coordinate. If the last component of the homogeneous coordinate is 1.0, the coordinate indicates the same position as the three-dimensional coordinate. In this case, because the last component is `w1+w2` to ensure that `w1+w2` is 1.0, you need to specify 0.0 to the value of w (the fourth parameter of `gl.uniform4f()`).

Finally, at line 58, `gl.drawArrays(gl.TRIANGLES, 0, n)` executes the vertex shader. For each execution, the following three steps are performed:

1. Each vertex coordinate set is passed to `a_Position`.
2. `u_Translation` is added to `a_Position`.
3. The result is assigned to `gl_Position`.

Once executed, you've achieved your goal because each vertex coordinate set is modified (translated), and then the translated shape (in this case, a triangle) is displayed on the screen. If you now load `TranslatedTriangle.html` into your browser, you will see the translated triangle.

Now that you've mastered translation (moving), the next step is to look at rotation. The basic approach to realize rotation is the same as translation, requiring you to manipulate the vertex coordinates in the vertex shader.

Rotation

Rotation is a little more complex than translation because you have to specify multiple items of information. The following three items are required:

- Rotation axis (the axis the shape will be rotated around)
- Rotation direction (the direction: clockwise or counterclockwise)
- Rotation angle (the number of degrees the shape will be rotated through)

In this section, to simplify the explanation, you can assume that the rotation is performed around the `z`-axis, in a counterclockwise direction, and for β degrees. You can use the same approach to implement other rotations around the x-axis or y-axis.

In the rotation, if β is positive, the rotation is performed in a counterclockwise direction around the rotation axis looking at the shape toward the negative direction of the z-axis (see Figure 3.21); this is called **positive rotation**. Just as for the coordinate system, your hand can define the direction of rotation. If you take your right hand and have your thumb follow the direction of the rotation axis, your fingers show the direction of rotation. This is called the **right-hand-rule rotation**. As we discussed in Chapter 2, it's the default we are using for WebGL in this book.

Now let's find the expression to calculate the rotation in the same way that you did for translation. As shown in Figure 3.22, we assume that the point p' (x', y', z') is the β degree rotated point of p (x, y, z) around the z-axis. Because the rotation is around the z-axis, the z coordinate does not change, and you can ignore it for now. The explanation is a little mathematical, so let's take it a step at a time.

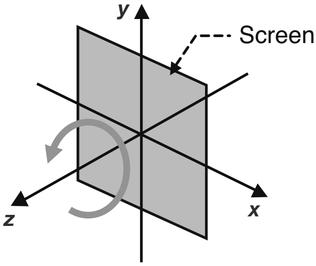


Figure 3.21 Positive rotation around the z-axis

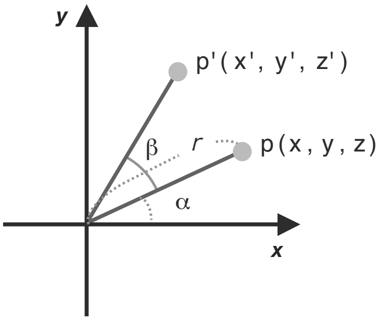


Figure 3.22 Calculating rotation around the z-axis

In Figure 3.22, `r` is the distance from the origin to the point `p`, and α is the rotation angle from the x-axis to the point. You can use these items of information to represent the coordinates of `p`, as shown in Equation 3.2.

Equation 3.2

$$x = r \cos \alpha$$

$$y = r \sin \alpha$$

Similarly, you can find the coordinate of p' by using `r`, α, and β as follows:

$$x' = r \cos (\alpha + \beta)$$

$$y' = r \sin (\alpha + \beta)$$

Then you can use the addition theorem of trigonometric functions[1] to get the following:

$$x' = r\,(\cos\alpha\cos\beta - \sin\alpha\sin\beta)$$

$$y' = r\,(\sin\alpha\cos\beta + \cos\alpha\sin\beta)$$

Finally, you get the following expressions (Equation 3.3) by assigning Equation 3.2 to the previous expressions and removing `r` and α.

Equation 3.3

$$x' = x\cos\beta - y\sin\beta$$

$$y' = x\sin\beta + y\cos\beta$$

$$z' = z$$

So by passing the values of sin β and cos β to the vertex shader and then calculating Equation 3.3 in the shader, you get the coordinates of the rotated point. To calculate sin β and cos β, you can use the methods of the JavaScript `Math` object.

Let's look at a sample program, `RotatedTriangle`, which rotates a triangle around the z-axis, in a counterclockwise direction, by 90 degrees. Figure 3.23 shows `RotatedTriangle`.

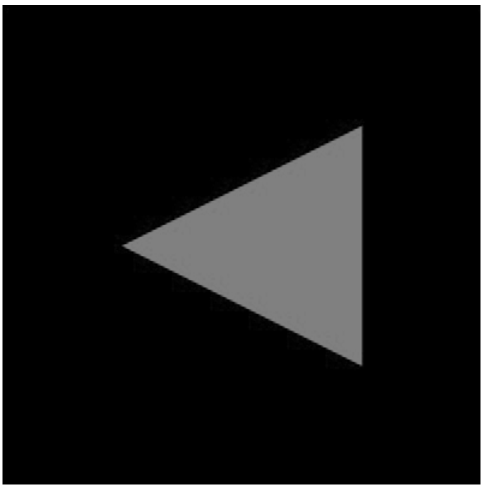


Figure 3.23 RotatedTriangle

[1] $\sin(a \pm \mathbf{b}) = \sin a\cos b \mp \cos a\sin b$
$\cos(a \pm b) = \cos a\cos b \mp \sin a\sin b$

Sample Program (RotatedTriangle.js)

Listing 3.5 shows `RotatedTriangle.js` which, in a similar manner to `TranslatedTriangle.js`, modifies the vertex shader to carry out the rotation operation. The fragment shader is the same as in `TranslatedTriangle.js` and, as usual, is not shown. Again, to support the shader modification, several processing steps are added to `main()` in the JavaScript program. Additionally, Equation 3.3 is added in the comments from lines 4 to 6 to remind you of the calculation needed.

Listing 3.5 RotatedTriangle.js

```
 1 // RotatedTriangle.js
 2 // Vertex shader program
 3 var VSHADER_SOURCE =
 4  // x' = x cos b - y sin b
 5  // y' = x sin b + y cos b                          Equation 3.3
 6  // z' = z
 7  'attribute vec4 a_Position;\n' +
 8  'uniform float u_CosB, u_SinB;\n' +
 9  'void main() {\n' +
10  '  gl_Position.x = a_Position.x * u_CosB - a_Position.y *u_SinB;\n'+
11 '  gl_Position.y = a_Position.x * u_SinB + a_Position.y * u_CosB;\n'+
12  '  gl_Position.z = a_Position.z;\n' +
13  '  gl_Position.w = 1.0;\n' +
14  '}\n';
15
16 // Fragment shader program
    ...
22 // Rotation angle
23 var ANGLE = 90.0;
24
25 function main() {
     ...
42   // Set the positions of vertices
43   var n = initVertexBuffers(gl);
     ...
49   // Pass the data required to rotate the shape to the vertex shader
50   var radian = Math.PI * ANGLE / 180.0; // Convert to radians
51   var cosB = Math.cos(radian);
52   var sinB = Math.sin(radian);
53
54   var u_CosB = gl.getUniformLocation(gl.program, 'u_CosB');
55   var u_SinB = gl.getUniformLocation(gl.program, 'u_SinB');
     ...
```

```
  gl.uniform1f(u_CosB, cosB);
  gl.uniform1f(u_SinB, sinB);

  // Set the color for clearing <canvas>
   ...
  // Draw a triangle
  gl.drawArrays(gl.TRIANGLES, 0, n);
}

function initVertexBuffers(gl) {
  var vertices = new Float32Array([
    0.0, 0.5,   -0.5, -0.5,   0.5, -0.5
  ]);
  var n = 3; // The number of vertices
   ...
  return n;
}
```

Let’s look at the vertex shader, which is straightforward:

```
 2 // Vertex shader program
 3 var VSHADER_SOURCE =
 4   // x' = x cos b - y sin b
 5   // y' = x sin b + y cos b
 6   // z' = z
 7   'attribute vec4 a_Position;\n' +
 8   'uniform float u_CosB, u_SinB;\n' +
 9   'void main() {\n' +
10   '  gl_Position.x = a_Position.x * u_CosB - a_Position.y * u_SinB;\n'+
11   '  gl_Position.y = a_Position.x * u_SinB + a_Position.y * u_CosB;\n'+
12   '  gl_Position.z = a_Position.z;\n' +
13   '  gl_Position.w = 1.0;\n' +
14   '}\n';
```

Because the goal is to rotate the triangle by 90 degrees, the sine and cosine of 90 need to be calculated. Line 8 defines two uniform variables for receiving these values, which are calculated in the JavaScript program and then passed to the vertex shader.

You could pass the rotation angle to the vertex shader and then calculate the values of sine and cosine in the shader. However, because they are identical for all vertices, it is more efficient to do it once in the JavaScript.

The name of these uniform variables, `u_CosB` and `u_SinB`, are defined following the naming rule used throughout this book. As you will remember, you use the uniform variable because the values of these variables are uniform (unchanging) per vertex.

As in the previous sample programs, x, y, z, and w are passed in a group to the attribute variable `a_Position` in the vertex shader. To apply Equation 3.3 to x, y, and z, you need to access each component in `a_Position` separately. You can do this easily using the `.` operator, such as `a_Position.x`, `a_Position.y`, and `a_Position.z` (see Figure 3.24 and Chapter 6).

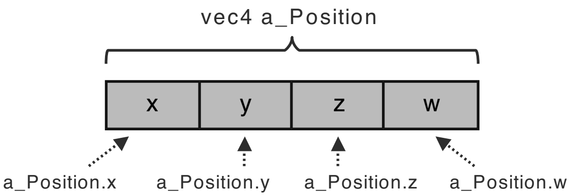


Figure 3.24 Access methods for each component in a vec4

Handily, you can use the same operator to access each component in `gl_Position` to which the vertex coordinate is written, so you can calculate x' = x cos β – y sin β from Equation 3.3 as shown at line 10:

```
10   '  gl_Position.x = a_Position.x * u_CosB - a_Position.y * u_SinB;\n'+
```

Similarly, you can calculate y' as follows:

```
11   '  gl_Position.y = a_Position.x * u_SinB + a_Position.y * u_CosB;\n'+
```

According to Equation 3.3, you just need to assign the original z coordinate to z' directly at line 12. Finally, you need to assign 1.0 to the last component `w`[2]:

```
12   '  gl_Position.z = a_Position.z;\n' +
13   '  gl_Position.w = 1.0;\n' +
```

Now look at `main()` in the JavaScript code, which starts from line 25. This code is mostly the same as in `TranslatedTriangle.js`. The only difference is passing cos β and sin β to the vertex shader. To calculate the sine and cosine of β, you can use the JavaScript `Math.sin()` and `Math.cos()` methods. However, these methods expect parameters in radians, not degrees, so you need to convert from degrees to radians by multiplying the number of degrees by pi and then dividing by 180. You can utilize `Math.PI` as the value of pi as shown at line 50, where the variable `ANGLE` is defined as 90 (degrees) at line 23:

```
50   var radian = Math.PI * ANGLE / 180.0; // Converts degrees to radians
```

[2] In this program, you can also write `gl_Position.w = a_Position.w;` because `a_Position.w` is 1.0.

Once you have the angle in radians, lines 51 and 52 calculate cos β and sin β, and then lines 60 and 61 pass them to the uniform variables in the vertex shader:

```
51    var cosB = Math.cos(radian);
52    var sinB = Math.sin(radian);
53
54    var u_CosB = gl.getUniformLocation(gl.program, 'u_CosB');
55    var u_SinB = gl.getUniformLocation(gl.program, 'u_SinB');
        ...
60    gl.uniform1f(u_CosB, cosB);
61    gl.uniform1f(u_SinB, sinB);
```

When you load this program into your browser, you can see the triangle, rotated through 90 degrees, on the screen. If you specify a negative value to `ANGLE`, you can rotate the triangle in the opposite direction (clockwise). You can also use the same equation. For example, to rotate the triangle in the clockwise direction, you can specify –90 instead of 90 at line 23, and `Math.cos()` and `Math.sin()` will deal with the remaining tasks for you.

For those of you concerned with speed and efficiency, the approach taken here (using two uniform variables to pass the values of cos β and sin β) isn't optimal. To pass the values as a group, you can define the uniform variable as follows:

```
uniform vec2 u_CosBSinB;
```

and then pass the values by:

```
gl.uniform2f(u_CosBSinB,cosB, sinB);
```

Then in the vertex shader, you can access them using `u_CosBSinB.x` and `u_CosBSinB.y`.

Transformation Matrix: Rotation

For simple transformations, you can use mathematical expressions. However, as your needs become more complex, you'll quickly find that applying a series of equations becomes quite complex. For example a "translation after rotation" as shown in Figure 3.25 can be realized by using Equations 3.1 and 3.3 to find the new mathematical expressions for the transformation and then implementing them in a vertex shader.

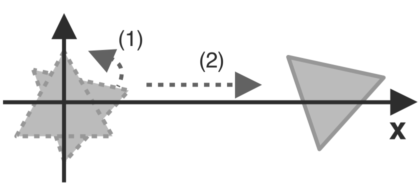


Figure 3.25 Rotate first and then translate a triangle

However, it is time consuming to determine the mathematical expressions every time you need a new set of transformation and then implement them in a vertex shader. Fortunately, there is another tool in the mathematical toolbox, the **transformation matrix**, which is excellent for manipulating computer graphics.

As shown in Figure 3.26, a matrix is a rectangular array of numbers arranged in rows (in the horizontal direction) and columns (in the vertical direction). This notation makes it easy to write the calculations explained in the previous sections. The brackets indicate that these numbers are a group.

$$\begin{bmatrix} 8 & 3 & 0 \\ 4 & 3 & 6 \\ 3 & 2 & 6 \end{bmatrix}$$

Figure 3.26 Example of a matrix

Before explaining the details of how to use a transformation matrix to replace the equations used here, you need to make sure you understand the multiplication of a matrix and a vector. A vector is an object represented by an n-tuple of numbers, such as the vertex coordinates (0.0, 0.5, 1.0).

The multiplication of a matrix and a vector can be written as shown in Equation 3.4. (Although the multiply operator × is often omitted, we explicitly write the operator in this book for clarity.) Here, our new vector (on the left) is the result of multiplying a matrix (in the center) by our original vector (on the right). Note that matrix multiplication is noncommutative. In other words, A × B is not the same as B × A. We discuss this further in Chapter 6.

Equation 3.4

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

This matrix has three rows and three columns and is called a 3×3 matrix. The rightmost part of the equation is a vector composed of x, y, and z. (In the case of a multiplication of a matrix and vector, the vector is written vertically, but it has the same meaning as when it is written horizontally.) This vector has three elements, so it is called a three-dimensional vector. Again, the brackets on both sides of the array of numbers (vector) are also just notation for recognizing that these numbers are a group.

In this case, x', y', and z' are defined using the elements of the matrix and the vector, as shown by Equation 3.5. Note that the multiplication of a matrix and vector can be defined only if the number of columns in a matrix matches the number of rows in a vector.

Equation 3.5

$$x' = ax + by + cz$$

$$y' = dx + ey + fz$$

$$z' = gx + hy + iz$$

Now, to understand how to use a matrix instead of our original equations, let's compare the matrix equations and Equation 3.3 (shown again as Equation 3.6).

Equation 3.6

$$x' = x \cos \beta - y \sin \beta$$

$$y' = x \sin \beta + y \cos \beta$$

$$z' = z$$

For example, compare the equation for x':

$$x' = ax + by + cz$$

$$x' = x \cos \beta - y \sin \beta$$

In this case, if you set a = cos β, b = -sin β, and c = 0, the equations become the same. Similarly, let us compare the equation for y':

$$y' = dx + ey + fz$$

$$y' = x \sin \beta + y \cos \beta$$

In this case, if you set d = sin β, e = cos β, and f = 0, you get the same equation. The last equation about z' is easy. If you set g = 0, h = 0, and i = 1, you get the same equation.

Then, by assigning these results to Equation 3.4, you get Equation 3.7.

Equation 3.7

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} \cos\beta & -\sin\beta & 0 \\ \sin\beta & \cos\beta & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

This matrix is called a **transformation matrix** because it "transforms" the right-side vector (x, y, z) to the left-side vector (x', y', z'). The transformation matrix representing a rotation is called a **rotation matrix**.

You can see that the elements of the matrix in Equation 3.7 are an array of coefficients in Equation 3.6. Once you become accustomed to matrix notation, it is easier to write and use matrices than to have to deal with a set of transformation equations.

As you would expect, because matrices are used so often in 3DCG, multiplication of a matrix and a vector is easy to implement in shaders. However, before exploring how, let's quickly look at other types of transformation matrices, and then we will start to use them in shaders.

Transformation Matrix: Translation

Obviously, if we can use a transformation matrix to represent a rotation, we should be able to use it for other types of transformation, such as translation. For example, let us compare the equation for x' in Equation 3.1 to that in Equation 3.5 as follows:

$x' = ax + by + cz$ --- from Equation (3.5)

$x' = x + T_x$ --- from Equation (3.1)

Here, the second equation has the constant term T_x, but the first one does not, meaning that you cannot deal with the second one by using the 3×3 matrix of the first equation. To solve this problem, you can use a 4×4 matrix and the fourth components of the coordinate, which are set to 1 to introduce the constant terms. That is to say, we assume that the coordinates of point p are (x, y, z, 1), and the coordinates of the translated point p (p') are (x', y', z', 1). This gives us Equation 3.8.

Equation 3.8

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

This multiplication is defined as follows:

Equation 3.9

$$x' = ax + by + cz + d$$

$$y' = ex + fy + gz + h$$

$$z' = ix + jy + kz + l$$

$$1 = mx + ny + oz + p$$

From the equation1 = mx + ny + oz + p, it is easy to find that the coefficients are m = 0, n = 0, o = 0, and p = 1. In addition, these equations have the constant terms d, h, and l, which look helpful to deal with Equation 3.1 because it also has constant terms. Let us compare Equation 3.9 and Equation 3.1 (translation), which is reproduced again:

$$x' = x + T_x$$

$$y' = y + T_y$$

$$z' = z + T_z$$

When you compare the x' component of both equations, you can see that a=1, b=0, c=0, and d=T_x. Similarly, when comparing y' from both equations, you find e = 0, f = 1, g = 0, and h = T_y; when comparing z' you see i=0, j=0, k=1, and *l*=T_z. You can use these results to write a matrix that represents a translation, called a **translation matrix**, as shown in Equation 3.10.

Equation 3.10

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & Tx \\ 0 & 1 & 0 & Ty \\ 0 & 0 & 1 & Tz \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Rotation Matrix, Again

At this stage you have successfully created a rotation and a translation matrix, which are equivalent to the two equations you used in the example programs earlier. The final step is to combine these two matrices; however, the rotation matrix (3×3 matrix) and transformation matrix (4×4 matrix) have different numbers of elements. Unfortunately, you cannot combine matrices of different sizes, so you need a mechanism to make them the same size.

To do that, you need to change the rotation matrix (3×3 matrix) into a 4×4 matrix. This is straightforward and requires you to find the coefficient of each equation in Equation 3.9 by comparing it with Equation 3.3. The following shows both equations:

$$x' = x \cos \beta - y \sin \beta$$

$$y' = x \sin \beta + y \cos \beta$$

$$z' = z$$

$$x' = ax + by + cz + d$$

$$y' = ex + fy + gz + h$$

$$z' = ix + iy + kz + l$$

$$1 = mx + ny + oz + p$$

For example, when you compare $x' = x \cos \beta - y \sin \beta$ with $x' = ax + by + cz + d$, you find $a = \cos \beta$, $b = -\sin \beta$, $c = 0$, and $d = 0$. In the same way, after comparing in terms of y and z, you get the rotation matrix shown in Equation 3.11:

Equation 3.11

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\beta & -\sin\beta & 0 & 0 \\ \sin\beta & \cos\beta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

This allows you to represent both a rotation matrix and translation matrix in the same 4×4 matrix, achieving the original goal!

Sample Program (RotatedTriangle_Matrix.js)

Having constructed a 4×4 rotation matrix, let's go ahead and use this matrix in a WebGL program by rewriting the sample program `RotatedTriangle`, which rotates a triangle 90 degrees around the z-axis in a counterclockwise direction, using the rotation matrix. Listing 3.6 shows `RotatedTriangle_Matrix.js`, whose output will be the same as Figure 3.23 shown earlier.

Listing 3.6 RotatedTriangle_Matrix.js

```
// RotatedTriangle_Matrix.js
// Vertex shader program
var VSHADER_SOURCE =
  'attribute vec4 a_Position;\n' +
  'uniform mat4 u_xformMatrix;\n' +
  'void main() {\n' +
  '  gl_Position = u_xformMatrix * a_Position;\n' +
  '}\n';

// Fragment shader program
...
// Rotation angle
var ANGLE = 90.0;
```

```

function main() {
  ...
  // Set the positions of vertices
  var n = initVertexBuffers(gl);
  ...
  // Create a rotation matrix
  var radian = Math.PI * ANGLE / 180.0; // Convert to radians
  var cosB = Math.cos(radian), sinB = Math.sin(radian);

  // Note: WebGL is column major order
  var xformMatrix = new Float32Array([
     cosB, sinB, 0.0, 0.0,
    -sinB, cosB, 0.0, 0.0,
     0.0,  0.0, 1.0, 0.0,
     0.0,  0.0, 0.0, 1.0
  ]);

  // Pass the rotation matrix to the vertex shader
  var u_xformMatrix = gl.getUniformLocation(gl.program, 'u_xformMatrix');
  ...
  gl.uniformMatrix4fv(u_xformMatrix, false, xformMatrix);

  // Set the color for clearing <canvas>
  ...
  // Draw a triangle
  gl.drawArrays(gl.TRIANGLES, 0, n);
}

function initVertexBuffers(gl) {
  var vertices = new Float32Array([
    0.0, 0.5,   -0.5, -0.5,   0.5, -0.5
  ]);
  var n = 3; // Number of vertices
  ...
  return n;
}
```

First, let us examine the vertex shader:

```
// Vertex shader program
var VSHADER_SOURCE =
  'attribute vec4 a_Position;\n' +
  'uniform mat4 u_xformMatrix;\n' +
```

```
6    'void main() {\n' +
7    '  gl_Position = u_xformMatrix * a_Position;\n' +
8    '}\n';
```

At line 7, `u_xformMatrix`, containing the rotation matrix described in Equation 3.11, and `a_Position`, containing the vertex coordinates (this is the right-side vector in Equation 3.11), are multiplied, literally implementing Equation 3.11.

In the sample program `TranslatedTriangle`, you were able to implement the addition of two vectors in one line (`gl_Position = a_Position + u_Translation`). In the same way, a multiplication of a matrix and vector can be written in one line in GLSL ES. This is convenient, allowing the calculation of the four equations (Equation 3.9) in one line. Again, this shows how GLSL ES has been designed specifically for 3D computer graphics by supporting powerful operations like this.

Because the transformation matrix is a 4×4 matrix and GLSL ES requires the data type for all variables, line 5 declares `u_xformMatrix` as type `mat4`. As you would expect, `mat4` is a data type specifically for holding a 4×4 matrix.

Within the main JavaScript program, the rest of the changes just calculate the rotation matrix from Equation 3.11 and then pass it to `u_xformMatrix`. This part starts from line 44:

```
43    // Create a rotation matrix
44    var radian = Math.PI * ANGLE / 180.0; // Convert to radians
45    var cosB = Math.cos(radian), sinB = Math.sin(radian);
46
47    // Note: WebGL is column major order
48    var xformMatrix = new Float32Array([
49       cosB, sinB, 0.0, 0.0,
50      -sinB, cosB, 0.0, 0.0,
51        0.0,  0.0, 1.0, 0.0,
52        0.0,  0.0, 0.0, 1.0
53    ]);
54
55    // Pass the rotation matrix to the vertex shader
     ...
61    gl.uniformMatrix4fv(u_xformMatrix, false, xformMatrix);
```

Lines 44 and 45 calculate the values of cosine and sine, which are required in the rotation matrix. Then line 48 creates the matrix `xformMatrix` using a `Float32Array`. Unlike GLSL ES, because JavaScript does not have a dedicated object for representing a matrix, you need to use the `Float32Array`. One question that arises is in which order you should store the elements of the matrix (which is arranged in rows and columns) in the elements of the array (which is arranged in a line). There are two possible orders: **row major order** and **column major order** (see Figure 3.27).

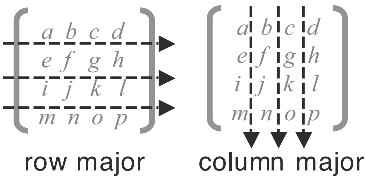


Figure 3.27 Row major order and column major order

WebGL, just like OpenGL, requires you to store the elements of a matrix in the elements of an array in column major order. So, for example, the matrix shown in Figure 3.27 is stored in an array as follows: [*a, e, i, m, b, f, j, n, c, g, k, o, d, h, l, p*]. In the sample program, the rotation matrix is stored in the `Float32Array` in this order in lines 49 to 52.

The array created is then passed to the uniform variable `u_xformMatrix` by using `gl.uniformMatrix4fv()` at line 61. Note that the last letter of this method name is `v`, which indicates that the method can pass multiple data values to the variable.

`gl.uniformMatrix4fv``(location, transpose, array)`

Assign the 4×4 matrix specified by *array* to the uniform variable specified by *location.*

Parameters	location	Specifies the storage location of the uniform variable.
	Transpose	Must be `false` in WebGL.[3]
	array	Specifies an array containing a 4×4 matrix in column major order (typed array).
Return value	None	
Errors	INVALID_OPERATION	There is no current program object.
	INVALID_VALUE	*transpose* is not `false`, or the length of *array* is less than 16.

If you load and run the sample program in your browser, you'll see the rotated triangle. Congratulations! You have successfully learned how to use a transformation matrix to rotate a triangle.

3 This parameter specifies whether to transpose the matrix or not. The transpose operation, which exchanges the column and row elements of the matrix (see Chapter 7), is not supported by WebGL's implementation of this method and must always be set to `false`.

Reusing the Same Approach for Translation

Now, as you have seen with Equations 3.10 and 3.11, you can represent both a translation and a rotation using the same type of 4×4 matrix. Both equations use the matrices in the form `<new coordinates> = <transformation matrix> * <original coordinates>`. This is coded in the vertex shader as follows:

```
7    '  gl_Position = u_xformMatrix * a_Position;\n' +
```

This means that if you change the elements of the array `xformMatrix` from those of a rotation matrix to those of a translation matrix, you will be able to apply the translation matrix to the triangle to achieve the same result as shown earlier but which used an equation (Figure 3.18).

To do that, change line 17 in `RotatedTriangle_Matrix.js` using the translation distances from the previous example:

```
17  varTx = 0.5, Ty = 0.5, Tz = 0.0;
```

You need to rewrite the code for creating the matrix, remembering that you need to store the elements of the matrix in column major order. Let's keep the same name for the array variable, `xformMatrix`, even though it's now being used to hold a translation matrix, because it reinforces the fact that we are using essentially the same code. Finally, you are not using the variable `ANGLE`, so lines 43 to 45 are commented out:

```
43   // Create a rotation matrix
44   // var radian = Math.PI * ANGLE / 180.0; // Convert to radians
45   // var cosB = Math.cos(radian), sinB = Math.sin(radian);
46
47    // Note: WebGL is column major order
48    var xformMatrix = new Float32Array([
49       1.0, 0.0, 0.0, 0.0,
50       0.0, 1.0, 0.0, 0.0,
51       0.0, 0.0, 1.0, 0.0,
52       Tx, Ty, Tz, 1.0
53    ]);
```

Once you've made the changes, run the modified program, and you will see the same output as shown in Figure 3.18. By using a transformation matrix, you can apply various transformations using the same vertex shader. This is why the transformation matrix is such a convenient and powerful tool for 3D graphics, and it's why we've covered it in detail in this chapter.

Transformation Matrix: Scaling

Finally, let's define the transformation matrix for scaling using the same assumption that the original point is `p` and the point after scaling is `p'`.

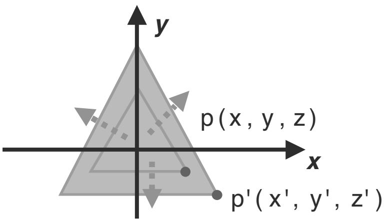


Figure 3.28 A scaling transformation

Assuming the scaling factor for the x-axis, y-axis, and z-axis is S_x, S_y, and S_z respectively, you obtain the following equations:

$$x' = S_x \times x$$

$$y' = S_y \times y$$

$$z' = S_z \times z$$

The following transformation matrix can be obtained by comparing these equations with Equation 3.9.

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

As with the previous example, if you store this matrix in `xformMatrix`, you can scale the triangle by using the same vertex shader you used in `RotatedTriangle_Matrix.js`. For example, the following sample program will scale the triangle by a factor of 1.5 in a vertical direction, as shown in Figure 3.29:

```
varSx = 1.0, Sy = 1.5, Sz = 1.0;
  ...
   // Note: WebGL is column major order
   var xformMatrix = new Float32Array([
      Sx,  0.0,  0.0,  0.0,
      0.0,  Sy,  0.0,  0.0,
      0.0,  0.0,  Sz,  0.0,
      0.0,  0.0,  0.0,  1.0
   ]
```

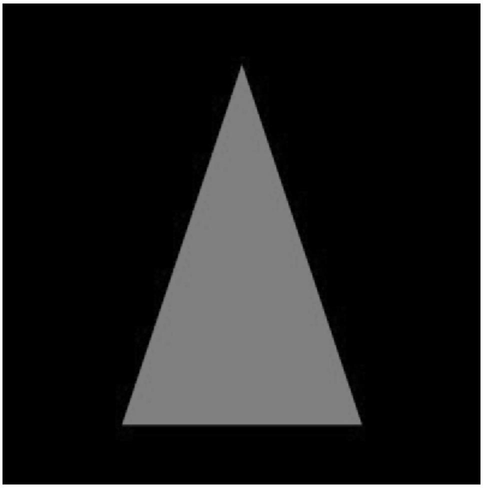


Figure 3.29 Triangle scaled in a vertical direction

Note that if you specify 0.0 to `Sx`, `Sy`, or `Sz`, the scaled size will be 0.0. If you want to keep the original size, specify 1.0 as the scaling factor.

Summary

In this chapter, you explored the process of passing multiple items of information about vertices to a vertex shader, the different types of shapes available to be drawn using that information, and the process of transforming those shapes. The shapes dealt with in this chapter changed from a point to a triangle, but the method of using shaders remained the same, as in the examples in the previous chapter. You were also introduced to matrices and learned how to use transformation matrices to apply translation, rotation, or scaling to 2D shapes. Although it's a little complicated, you should now have a good understanding of the math behind calculating the individual transformation matrices.

In the next chapter, you'll explore more complex transformations but will use a handy library to hide the details, allowing you to focus on the higher-level tasks.

This page intentionally left blank

Index

Symbols

2D graphics
- coloring vertices different colors, 151-160
  - geometric shape assembly and rasterization, 151-155
  - invoking fragment shader, 155
  - varying variables and interpolation process, 157-160
  - verifying fragment shader invocation, 156-157
- combining multiple transformations, 119-121
- drawing
  - rectangles, 13-16, 89-91
  - triangles, 85-91
- pasting images on, 160-183
  - activating texture units, 171-172
  - assigning texture images to texture objects, 177-179
  - binding texture objects to target, 173-174
  - changing texture coordinates, 182-183
  - flipping image y-axis, 170-171
  - mapping texture and vertex coordinates, 162-163, 166
  - multiple texture mapping, 183-190
  - passing texture coordinates from vertex to fragment shader, 180-181
  - passing texture unit to fragment shader, 179-180
  - retrieving texel color in fragment shader, 181-182
  - setting texture object parameters, 174-177
  - setting up and loading images, 166-170
  - texture coordinates, explained, 162
  - TexturedQuad.js, 163-166
- restoring clipped parts, 251-253
- rotating, 96-102, 107-110, 234-235
- RotatingTranslatedTriangle.js, 135-136
- RotatingTriangle.js, 126-129
  - calling drawing function, 129-130
  - draw() function, 130-131
  - requestAnimationFrame() function, 131-133
  - updating rotation angle, 133-135
- TranslatedTriangle.js, 92-96
- translating, 92-96, 111

3D graphics. *See also* WebGL
alpha blending, 384
applications
browser functionality in, 5
publishing, ease of, 4
writing in text editors, 3-4
displaying on web pages (3DoverWeb), 372
lighting, 291-293
light sources, 293
reflected light, 294-296
loading, 414-416
modeling tools, 415, 473-475
point light objects, 314-315
shading, 292
3D models
MTL file format, 418
OBJ file format, 417
OBJViewer.js, 419-421
parser code, 423-430
user-defined objects, 422-423
3DoverWeb, 372
[ ] (array indexing) operator, 203-204
. (dot) operator, 201-202
; (semicolon), in GLSL ES, 193

A

abs() function, 444
access to members
of arrays in GLSL ES, 209
of structures in GLSL ES, 207
of vector and matrix data types, 201-204
acos() function, 442
activating texture units, 171-172
adding
color to each face (Hello Cube), 284-285
shading to ambient light, 307-308
affine transformations, 91
all() function, 450
alpha blending, 380
3D objects, 384
blending function, 382-383
drawing when alpha values coexist, 385-386
implementing, 380-381
LookAtBlendedTriangles.js, 381-382
ambient light, 294
shading, adding, 307-308
ambient reflection, 295-296
angle functions, 216, 441-442
animate() function, 129, 133-135
animation, 124-136
multiple transformations in, 135-136
RotatingTriangle.js, 126-129
calling drawing function, 129-130
draw() function, 130-131
requestAnimationFrame() function, 131-133
updating rotation angle, 133-135
anonymous functions, 52-53
any() function, 450
applications
3D graphics applications
browser functionality in, 5
publishing, ease of, 4
writing in text editors, 3-4
WebGL application structure, 6-7

array indexing ([ ]) operator, 203-204
arrays
  in GLSL ES, 208-209
  interleaving, 141-145
  typed arrays, 78-79
asin() function, 442
assigning
  buffer objects to attribute variables, 79-81
  texture images to texture objects, 177-179
  values
    in GLSL ES structures, 207
    in GLSL ES variables, 196-197
    in matrix data types, 199-201
    in vector data types, 199-201
asynchronous loading of texture images, 169-170
atan() function, 442
atmospheric effects, fog, 372-373
attaching shader objects to program objects, 350-351
attribute variables, 217-218
  assiging buffer objects to, 79-81
  declaring, 43
  enabling assignment, 81-82
  explained, 41-42
  for point size (MultiAttributeSize.js), 139-140
  setting value, 45-49
  storage location, 44-45

B

back color buffer, 437
background objects, 267-269
  z fighting, 273-275
binding
  buffer objects to targets, 75-76
  renderbuffer objects, 399-400
  texture objects to targets, 173-174
BlendedCube.js, 384
Blender 3D modeling tool, 415, 473-475
blending function, alpha blending, 382-383
<body> element, 12
bool data type, 196
Boolean values in GLSL ES, 194
boxed-shape viewing volume
  defining, 243-244
  OrthoView.html, 245-246
  OrthoView.js, 246-247
break statement in GLSL ES, 212-213
browsers
  <canvas> element support, 12
  console, viewing, 14
  enabling local file access, 161
  functionality in 3D graphics applications, 5
  JavaScript to WebGL processing flow, 27, 438
  WebGL settings, 479-480
buffer objects
  assigning to attribute variables, 79-81
  binding to targets, 75-76
  creating, 74-75

creating multiple, 140-141
defined, 69
enabling attribute variable assignment, 81-82
explained, 72-74
writing vertex coordinates, colors, and indices in, 281-284
writing data to, 76-78
buffers
color buffers, 22
drawing to, 437-439
saving content from, 56
swapping, 437
depth buffer, 22
types of, 22
built-in functions in GLSL ES, 215-216

C

calculating
color per fragment, 319
diffuse reflection, 297-299
cancelAnimationFrame() function, 133
canvas.addEventListener() function, 432
<canvas> element, 9-11, 243
browser support, 12
clearing drawing area, 16-23
coordinates for center, 23
coordinate system, 16
DrawRectangle.html, 11-13
DrawRectangle.js, 13-16
HelloCanvas.html, 17-18
HelloCanvas.js, 18-23
mapping to WebGL coordinate system, 39, 54-57
retrieving, 14, 19
canvas.getContext() function, 15
case sensitivity of GLSL ES, 193
ceil() function, 444
changing
color with varying variables, 146-151
eye point using keyboard, 238
near value, 250-251
checkFace() function, 368
Chrome
console, viewing, 14
enabling local file access, 161
WebGL browser settings, 479
clamp() function, 445
clear color, setting, 21-23
clearing
color buffer, 22
drawing area, 16-23
ClickedPoints.js, 50-52
clip coordinate system, viewing volume and, 460-462
clipped parts, restoring, 251-253
color
adding to each face (Hello Cube), 284-285
changing with varying variables, 146-151
of points, changing, 58-66
setting, 15, 21-23
texel color, retrieving in fragment shader, 181-182
color buffers, 22
drawing to, 437-439

saving content from, 56
swapping, 437
ColoredCube.js, 285-289
ColoredPoints.js, 59-61
ColoredTriangle.js, 159
coloring vertices, 151-160
geometric shape assembly and rasterization, 151-155
invoking fragment shader, 155
varying variables and interpolation process, 157-160
verifying fragment shader invocation, 156-157
color per fragment, calculating, 319
column major order, 109-110
combining multiple transformations, 119-121
comments in GLSL ES, 193
common functions, 216, 444-446
compiling shader objects, 347-349
conditional control flow in GLSL ES, 211-213
console, viewing, 14
constant index, 203
constants of typed arrays, 79
constructors in GLSL ES, 199-201
for structures, 207
const variables in GLSL ES, 217
context, retrieving, 15, 20-21
continue statement in GLSL ES, 212-213
coordinates
center of canvas, 23
homogeneous coordinates, 35
for mouse clicks, 54-57
WebGL coordinate system, 38-39
CoordinateSystem.js, 456-459
coordinate systems
for <canvas> element, 16
clip coordinate system and viewing volume, 460-462
CoordinateSystem.js, 456-459
handedness in default behavior, 455-464
Hidden Surface Removal tool, 459-460
local coordinate system, 474-475
projection matrices for, 462-464
texture coordinates
changing, 182-183
explained, 162
flipping image y-axis, 170-171
mapping to vertex coordinates, 162-163, 166
passing from vertex to fragment shader, 180-181
transformations and, 477
world coordinate system, 475-477
CoordinateSystem_viewVolume.js, 461
cos() function, 442
createProgram() function, 354
cross() function, 447
ctx.fillRect() function, 16
cubes, 301
cuboids, 301
cuon-matrix.js, 116
cuon-utils.js, 20

D

data, passing
  to fragment shaders with varying variable, 146-151
  to vertex shaders, 137-151. *See also* drawing; rectangles; shapes; triangles
    color changes, 146-151
    creating multiple buffer objects, 140-141
    interleaving, 141-145
    MultiAttributeSize.js, 139-140
data types
  in GLSL ES, 34, 194-196
    arrays, 208-209
    operators on, 197-198
    precision qualifiers, 219-221
    samplers, 209-210
    structures, 207-208
    type conversion, 196-197
    type sensitivity, 195
    vector and matrix types, 198-206
  typed arrays, 78-79
#define preprocessor directive, 222
degrees() function, 441
deleting
  shader objects, 346
  texture objects, 167
depth buffer, 22
DepthBuffer.js, 272-273
diffuse reflection, 294-295
  calculating, 297-299
  shading, 296-297
Direct3D, 5
directional light, 293
  shading, 296-297
discard statement in GLSL ES, 212-213
displaying 3D objects on web pages (3DoverWeb), 372
distance() function, 447
document.getElementById() function, 14, 19
dot() function, 447
dot (.) operator, 201-202
draw() function, 129-131
  objects composed of other objects, 332-334
  processing flow of, 249
drawArrays() function, 284
drawbox() function, 339-340
drawing
  to color buffers, 437-439
  Hello Cube with indices and vertices coordinates, 277-278
  multiple points/vertices, 68-85
    assigning buffer objects to attribute variables, 79-81
    binding buffer objects to targets, 75-76
    buffer object usage, 72-74
    creating buffer objects, 74-75
    enabling attribute variable assignments, 81-82
    gl.drawArrays() function, 82-83
    writing data to buffer objects, 76-78
  objects composed of other objects, 324-325
  points
    assigning uniform variable values, 63-66
    attribute variables, 41-42

attribute variable storage location, 44-45
attribute variable value, 45-49
changing point color, 58-66
ClickedPoints.js, 50-52
ColoredPoints.js, 59-61
fragment shaders, 35-36
gl.drawArrays() function, 36-37
handling mouse clicks, 53-57
HelloPoint1.html, 25
HelloPoint1.js, 25-26
HelloPoint2.js, 42-43
initializing shaders, 30-33
method one, 23-41
method two, 41-50
with mouse clicks, 50-58
registering event handlers, 52-53
shaders, explained, 27-28
uniform variables, 61-62
uniform variable storage location, 62-63
vertex shaders, 33-35
WebGL coordinate system, 38-39
WebGL program structure, 28-30
rectangles, 13-16, 89-91
shapes, 85-91
animation, 124-136
multiple vertices, 68-85
rotating, 96-110
scaling, 111-113
transformation libraries, 115-124
translating, 92-96, 105-106, 111
triangles, 85-81
coloring vertices different colors, 151-160
combining multiple transformations, 119-121
HelloTriangle.js, 85-86
restoring clipped parts, 251-253
rotating, 96-102, 107-110, 234-235
RotatingTranslatedTriangle.js, 135-136
RotatingTriangle.js, 126-135
TranslatedTriangle.js, 92-96
translating, 92-96, 111
using framebuffer objects, 403-404
when alpha values coexist, 385-386
drawing area
clearing, 16-23
defining, 12
mapping to WebGL coordinate system, 39
drawing context. *See* context, retrieving
drawing function (tick()), calling repeatedly, 129-130
DrawRectangle.html, 11-13
DrawRectangle.js, 13-16
drawSegment() function, 340
draw segments, objects composed of other objects, 339-344
dynamic web pages, WebGL web pages versus, 7

E

#else preprocessor directive, 222
enabling
attribute variable assignment, 81-82
local file access, 161
equal() function, 450
event handlers
for mouse clicks, 53-57
registering, 52-53

execution order in GLSL ES, 193
exp2() function, 443
exp() function, 443
exponential functions, 216, 443
eye point, 228
  changing using keyboard, 238
  LookAtTrianglesWithKeys.js, 238-241
  visible range, 241

F

faceforward() function, 447
face of objects, selecting, 365
  PickFace.js, 366-368
files, loading shader programs from, 471-472
fill color, setting, 15
Firefox
  console, viewing, 14
  enabling local file access, 161
  WebGL browser settings, 480
flipping image y-axis, 170-171
Float32Array object, 78
float data type, 196
floor() function, 444
flow of vertex shaders, processing, 248-249
fog, 372-373
  implementing, 373-374
  w value, 376-377
Fog.js, 374-376
Fog_w.js, 376-377
foreground objects, 267-269
  DepthBuffer.js, 272-273
  hidden surface removal, 270-271
  z fighting, 273-275
for statement in GLSL ES, 211-212
fract() function, 444
fragments, 27, 35
fragment shaders, 27
  drawing points, 35-36
  example of, 192
  geometric shape assembly and rasterization, 151-155
  invoking, 155
  passing
    data to, 61-62, 146-151
    texture coordinates to, 180-181
    texture units to, 179-180
  program structure, 29-30
  retrieving texel color in, 181-182
  varying variables and interpolation process, 157-160
  verifying invocation, 156-157
FramebufferObject.js, 395-396, 403
framebuffer objects, 392-393
  checking configurations, 402-403
  creating, 397
  drawing with, 403-404
  renderbuffer objects set to, 401-402
  setting to renderbuffer objects, 400-401
front color buffer, 437
functions
  abs() function, 444
  acos() function, 442

all() function, 450
angle and trigonometry functions, 441-442
animate() function, 129, 133-135
anonymous functions, 52-53
any() function, 450
asin() function, 442
atan() function, 442
built-in functions in GLSL ES, 215-216
cancelAnimationFrame() function, 133
canvas.addEventListener() function, 432
canvas.getContext() function, 15
ceil() function, 444
checkFace() function, 368
clamp() function, 445
common functions, 444-446
cos() function, 442
createProgram() function, 354
cross() function, 447
ctx.fillRect() function, 16
degrees() function, 441
distance() function, 447
document.getElementById() function, 14, 19
dot() function, 447
draw() function, 129-131
  objects composed of other objects, 332-334
  processing flow of, 249
drawArrays() function, 284
drawbox() function, 339-340
drawSegment() function, 340
equal() function, 450
exp2() function, 443
exp() function, 443
exponential functions, 216, 443
faceforward() function, 447
floor() function, 444
fract() function, 444
geometric functions, 216, 447-448
getWebGLContext() function, 20
gl.activeTexture() function, 171-172
gl.attachShader() function, 350
gl.bindBuffer() function, 75-76
gl.bindFramebuffer() function, 400
gl.bindRenderbuffer() function, 399
gl.bindTexture() function, 173-174
gl.blendFunc() function, 382-383
gl.bufferData() function, 76-78
gl.checkFramebufferStatus() function, 402-403
gl.clearColor() function, 20-21
gl.clear() function, 22, 125
gl.compileShader() function, 347-349
gl.createBuffer() function, 74-75
gl.createFramebuffer() function, 397
gl.createProgram() function, 349-350
gl.createRenderbuffer() function, 398
gl.createShader() function, 345-346
gl.createTexture() function, 167
gl.deleteBuffer() function, 75
gl.deleteFramebuffer() function, 397
gl.deleteProgram() function, 350
gl.deleteRenderbuffer() function, 398
gl.deleteShader() function, 346
gl.deleteTexture() function, 167

gl.depthMask() function, 385
gl.detachShader() function, 351
gl.disable() function, 271
gl.disableVertexAttribArray() function, 82
gl.drawArrays() function, 36-37, 72, 82-83, 87, 131
gl.drawElements() function, 278
gl.enable() function, 270
gl.enableVertexAttribArray() function, 81-82
gl.framebufferRenderbuffer() function, 401-402
gl.framebufferTexture2D() function, 401
gl.getAttribLocation() function, 44-45
gl.getProgramInfoLog() function, 352
gl.getProgramParameter() function, 352
gl.getShaderInfoLog() function, 348
gl.getShaderParameter() function, 348
gl.getUniformLocation() function, 63
gl.linkProgram() function, 351-352
gl.pixelStorei() function, 171
gl.polygonOffset() function, 274
gl.readPixels() function, 364
gl.renderbufferStorage() function, 399
gl.shaderSource() function, 346-347
gl.texImage2D() function, 177-179, 398
gl.texParameteri() function, 174-177
glTranslatef() function, 116
gl.uniform1f() function, 65-66
gl.uniform1i() function, 179-180
gl.uniform2f() function, 65-66
gl.uniform3f() function, 65-66
gl.uniform4f() function, 63-66, 95
gl.uniformMatrix4fv() function, 110
gl.useProgram() function, 353, 387
gl.vertexAttrib1f() function, 47-49
gl.vertexAttrib2f() function, 47-49
gl.vertexAttrib3f() function, 45-49
gl.vertexAttrib4f() function, 47-49
gl.vertexAttribPointer() function, 79-81, 142-145
gl.viewport() function, 404
in GLSL ES, 213-215
  built-in functions, 215-216
  parameter qualifiers, 214-215
  prototype declarations, 214
greaterThanEqual() function, 449
greaterThan() function, 449
initShaders() function, 31-32, 344-345, 353-355, 387
initTextures() function, 166-170, 187
initVertexBuffers() function, 72, 140, 152, 166, 187, 281
inversesqrt() function, 443
length() function, 447
lessThanEqual() function, 449
lessThan() function, 449
loadShader() function, 355
loadShaderFile() function, 472
loadTexture() function, 168, 170, 187-189
log() function, 443
log2() function, 443
main() function, processing flow of, 19
mathematical common functions, 444-446
Matrix4.setOrtho() function, 453
Matrix4.setPerspective() function, 453
matrix functions, 216, 448
max() function, 445

maxtrixCompMult() function, 448
min() function, 445
mix() function, 446
mod() function, 444
normalize() function, 447
notEqual() function, 450
not() function, 450
onLoadShader() function, 472
OpenGL functions, naming conventions, 48-49
popMatrix() function, 338
pow() function, 443
pushMatrix() function, 338
radians() function, 441
reflect() function, 448
refract() function, 448
requestAnimationFrame() function, 130-133
setInterval() function, 131
setLookAt() function, 228-229
setOrtho() function, 243
setPerspective() function, 257
setRotate() function, 117, 131
sign() function, 444
sin() function, 442
smoothstep() function, 446
sqrt() function, 443
stencil buffer, 22
step() function, 446
tan() function, 442
texture lookup functions, 451
texture2D() function, 181-182, 451
texture2DLod() function, 451
texture2DProj() function, 451
texture2DProjLod() function, 451
textureCube() function, 451
textureCubeLod() function, 451
tick() function, 129-130
trigonometry functions, 216, 441-442
type conversion, 197
vec4() function, 34-35
vector functions, 48, 216, 449

G

geometric functions, 216, 447-448
geometric shape assembly, 151-155
getWebGLContext() function, 20
gl.activeTexture() function, 171-172
gl.attachShader() function, 350
gl.bindBuffer() function, 75-76
gl.bindFramebuffer() function, 400
gl.bindRenderbuffer() function, 399
gl.bindTexture() function, 173-174
gl.blendFunc() function, 382-383
gl.bufferData() function, 76-78
gl.checkFramebufferStatus() function, 402-403
gl.clearColor() function, 20-21
gl.clear() function, 22, 125
gl.compileShader() function, 347-349
gl.createBuffer() function, 74-75
gl.createFramebuffer() function, 397
gl.createProgram() function, 349-350
gl.createRenderbuffer() function, 398
gl.createShader() function, 345-346
gl.createTexture() function, 167
gl.deleteBuffer() function, 75
gl.deleteFramebuffer() function, 397

gl.deleteProgram() function, 350
gl.deleteRenderbuffer() function, 398
gl.deleteShader() function, 346
gl.deleteTexture() function, 167
gl.depthMask() function, 385
gl.detachShader() function, 351
gl.disable() function, 271
gl.disableVertexAttribArray() function, 82
gl.drawArrays() function, 36-37, 72, 82-83, 87, 131
gl.drawElements() function, 278
gl.enable() function, 270
gl.enableVertexAttribArray() function, 81-82
gl.framebufferRenderbuffer() function, 401-402
gl.framebufferTexture2D() function, 401
gl.getAttribLocation() function, 44-45
gl.getProgramInfoLog() function, 352
gl.getProgramParameter() function, 352
gl.getShaderInfoLog() function, 348
gl.getShaderParameter() function, 348
gl.getUniformLocation() function, 63
gl.linkProgram() function, 351-352
global coordinate system. *See* world coordinate system
global variables in GLSL ES, 216
gl.pixelStorei() function, 171
gl.polygonOffset() function, 274
gl.readPixels() function, 364
gl.renderbufferStorage() function, 399
gl.shaderSource() function, 346-347
GLSL ES (OpenGL ES shading language), 6, 30
  case sensitivity, 193
  comments, 193
  conditional control flow and iteration, 211-213
  data types, 34, 194
    arrays, 208-209
    precision qualifiers, 219-221
    samplers, 209-210
    structures, 207-208
    vector and matrix types, 198-206
  functions, 213-215
    built-in functions, 215-216
    parameter qualifiers, 214-215
    prototype declarations, 214
  order of execution, 193
  overview of, 192
  preprocessor directives, 221-223
  semicolon (;) usage, 193
  variables
    assignment of values, 196-197
    data types for, 196
    global and local variables, 216
    keywords and reserved words, 194-195
    naming conventions, 194
    operator precedence, 210
    operators on, 197-198
    storage qualifiers, 217-219
    type conversion, 196-197
    type sensitivity, 195
GLSL (OpenGL shading language), 6
gl.texImage2D() function, 177-179, 398
gl.texParameteri() function, 174-177
glTranslatef() function, 116
gl.uniform1f() function, 65-66
gl.uniform1i() function, 179-180

gl.uniform2f() function, 65-66
gl.uniform3f() function, 65-66
gl.uniform4f() function, 63-66, 95
gl.uniformMatrix4fv() function, 110
gl.useProgram() function, 353, 387
gl.vertexAttrib1f() function, 47-49
gl.vertexAttrib2f() function, 47-49
gl.vertexAttrib3f() function, 45-49
gl.vertexAttrib4f() function, 47-49
gl.vertexAttribPointer() function, 79-81, 142-145
gl.viewport() function, 404
greaterThanEqual() function, 449
greaterThan() function, 449

H

handedness of coordinate systems, 455-464
  clip coordinate system and viewing volume, 460-462
  CoordinateSystem.js, 456-459
  Hidden Surface Removal tool, 459-460
  projection matrices for, 462-464
Head Up Display (HUD), 368
  HUD.html, 369-370
  HUD.js, 370-372
  implementing, 369
HelloCanvas.html, 17-18
HelloCanvas.js, 18-23
Hello Cube, 275-277
  adding color to each face, 284-285
  ColoredCube.js, 285-289
  drawing with indices and vertices coordinates, 277-278
  HelloCube.js, 278-281
  writing vertex coordinates, colors, and indices in the buffer object, 281-284
HelloCube.js, 278-281
HelloPoint1.html, 25
HelloPoint1.js, 25-26
HelloPoint2.js, 42-43
HelloQuad.js, 89-91
HelloTriangle.js, 85-86, 151-152
hidden surface removal, 270-271, 459-460
hierarchical structure, 325-326
highp precision qualifier, 220
homogeneous coordinates, 35
HTML5
  <body> element, 12
  <canvas> element, 9-11
    browser support, 12
    clearing drawing area, 16-23
    coordinates for center, 23
    coordinate system, 16
    DrawRectangle.html, 11-13
    DrawRectangle.js, 13-16
    HelloCanvas.html, 17-18
    HelloCanvas.js, 18-23
    mapping to WebGL coordinate system, 39, 54-57
    retrieving, 14, 19
  defined, 2
  elements, modifying using JavaScript, 247-248
  <img> element, 9

HUD (Head Up Display), 368
HUD.html, 369-370
HUD.js, 370-372
implementing, 369
HUD.html, 369-370
HUD.js, 370-372

I

identifiers, assigning, 12
identity matrix, 119
handedness of coordinate systems, 462-463
if-else statement in GLSL ES, 211
if statement in GLSL ES, 211
images, pasting on rectangles, 160-183
activating texture units, 171-172
assigning texture images to texture objects, 177-179
binding texture objects to target, 173-174
changing texture coordinates, 182-183
flipping image y-axis, 170-171
mapping texture and vertex coordinates, 162-163, 166
multiple texture mapping, 183-190
passing coordinates from vertex to fragment shader, 180-181
passing texture unit to fragment shader, 179-180
retrieving texel color in fragment shader, 181-182
setting texture object parameters, 174-177
setting up and loading images, 166-170
texture coordinates, explained, 162
TexturedQuad.js, 163-166
<img> element, 9
implementing
alpha blending, 380-381
fog, 373-374
HUD, 369
lost context, 431-432
object rotation, 358
object selection, 361-362
rounded points, 377-378
shadows, 405-406
switching shaders, 387
texture images, 394
indices, 282
infinity in homogenous coordinates, 35
initializing shaders, 30-33
initShaders() function, 31-32, 344-345, 353-355, 387
initTextures() function, 166-170, 187
initVertexBuffers() function, 72, 140, 152, 166, 187, 281
int data type, 196
integral constant expression, 208
interleaving, 141-145
interpolation process, varying variables and, 157-160
inversesqrt() function, 443
inverse transpose matrix, 311-312, 465-469
iteration in GLSL ES, 211-213

J–K

JavaScript
  drawing area, mapping to WebGL coordinate system, 39
  HTML elements, modifying, 247-248
  loading, 12
  processing flow into WebGL, 27, 438
JointModel.js, 328-332
joints, 325
  JointModel.js, 328-332
  multijoint model, 334
  MultiJointModel.js, 335-338
  single joint model, objects composed of other objects, 326-327
keyboard, changing eye point, 238
keywords in GLSL ES, 194-195
Khronos Group, 6

L

left-handedness of coordinate systems in default behavior, 455-464
length() function, 447
lessThanEqual() function, 449
lessThan() function, 449
libraries, transformation, 115-124
  combining multiple transformations, 119-121
  cuon-matrix.js, 116
  RotatedTranslatedTriangle.js, 121-124
  RotatedTriangle_Matrix4.js, 117-119
light direction, calculating diffuse reflection, 297-299
LightedCube_ambient.js, 308-309
LightedCube.js, 302-303
  processing in JavaScript, 306
  processing in vertex shader, 304-305
LightedTranslatedRotatedCube.js, 312-314
lighting
  3D objects, 291-293
    light sources, 293
    reflected light, 294-296
  ambient light, 294
  directional light, 293
  point light, 293
  reflected light, 294-296
  translated-rotated objects, 310-311
light sources, 293
linking program objects, 351-352
listings
  array with multiple vertex information items, 141
  BlendedCube.js, 384
  ClickedPoints.js, 51-52
  ColoredCube.js, 286-287
  ColoredPoints.js, 59-61
  ColoredTriangle.js, 159
  CoordinateSystem.js, 456-458
  CoordinateSystem_viewVolume.js, 461
  createProgram(), 354
  DepthBuffer.js, 272-273
  drawing multiple points, 69
  DrawRectangle.html, 11
  DrawRectangle.js, 13-14
  Fog.js, 374-375
  Fog_w.js, 376-377
  fragment shader example, 192

FramebufferObject.js
Processes for Steps 1 to 7, 395-396
Process for Step 8, 403-404
HelloCanvas.html, 18
HelloCanvas.js, 18-19
HelloCube.js, 279-280
HelloPoint1.html, 25
HelloPoint1.js, 26
HelloPoint2.js, 42-43
HelloTriangle.js, 85-86
code snippet, 151-152
HUD.html, 369
HUD.js, 370-371
initShaders(), 353-354
JointModel.js, 328-330
LightedCube_ambient.js, 308-309
LightedCube.js, 302-303
LightedTranslatedRotatedCube.js, 312-313
loadShader(), 355
LoadShaderFromFiles, 471-472
LookAtBlenderTriangles.js, 381-382
LookAtRotatedTriangles.js, 235-236
LookAtRotatedTriangles_mvMatrix.js, 237
LookAtTriangles.js, 229-231
LookAtTrianglesWithKeys.js, 239-240
LookAtTrianglesWithKeys_View Volume.js, 252-253
MultiAttributeColor.js, 147-148
MultiAttributeSize_Interleaved.js, 142-143
MultiAttributeSize.js, 139-140
MultiJointModel.js
drawing the hierarchy structure, 336-337
key processing, 335-336
MultiJointModel_segment.js, 340-342
MultiPoint.js, 70-72
MultiTexture.js, 185-186
OBJViewer.js, 419-420
onReadComplete(), 428
parser part, 424-426
retrieving the drawing information, 428-429
OrthoView.html, 245-246
OrthoView.js, 246-247
PerspectiveView.js, 258-259
PerspectiveView_mvp.js, 263-265
PickFace.js, 366-367
PickObject.js, 362-363
PointLightedCube.js, 316-317
PointLightedCube_perFragment.js, 319-320
ProgramObject.js
Processes for Steps 1 to 4, 387-389
Processes for Steps 5 through 10, 390-391
RotatedTranslatedTriangle.js, 122
RotatedTriangle.js, 99-100
RotatedTriangle_Matrix4.html, 116
RotatedTriangle_Matrix.js, 107-108
RotateObject.js, 358-359
RotatingTriangle_contextLost.js, 433-434
RotatingTriangle.js, 126-128
RoundedPoint.js, 379

Shadow_highp.js, 413-414
Shadow.js
JavaScript part, 410-411
Shader part, 406-407
TexturedQuad.js, 163-165
TranslatedTriangle.js, 93-94
vertex shader example, 192
Zfighting.js, 274-275
loading
3D objects, 414-416
images for texture mapping, 166-170
JavaScript, 12
shader programs from files, 471-472
loadShader() function, 355
loadShaderFile() function, 472
loadTexture() function, 168, 170, 187-189
local coordinate system, 474-475
local file access, enabling, 161
local variables in GLSL ES, 216
log() function, 443
log2() function, 443
LookAtBlendedTriangles.js, 381-382
look-at point, 228
LookAtRotatedTriangles.js, 235-238
LookAtRotatedTriangles_mvMatrix.js, 237
LookAtTriangles.js, 229-233
LookAtTrianglesWithKeys.js, 238-241
LookAtTrianglesWithKeys_ViewVolume.js, 251-253
lost context, 430-431
implementing, 431-432
RotatingTriangle_contextLost.js, 432-434
lowp precision qualifier, 220
luminance, 178

M

Mach band, 409
macros, predefined names, 222
main() function, processing flow of, 19
manipulating objects composed of other objects, 324-325
mapping textures, 160-183
activating texture units, 171-172
assigning texture images to texture objects, 177-179
binding texture objects to target, 173-174
changing texture coordinates, 182-183
flipping image y-axis, 170-171
mapping vertex and texture coordinates, 162-163, 166
passing coordinates from vertex to fragment shader, 180-181
passing texture unit to fragment shader, 179-180
pasting multiple textures, 183-190
retrieving texel color in fragment shader, 181-182
setting texture object parameters, 174-177
setting up and loading images, 166-170
texture coordinates, explained, 162
TexturedQuad.js, 163-166
mathematical common functions, 444-446
matrices
defined, 103
identity matrix, 119
handedness of coordinate systems, 462-463

inverse transpose matrix, 311-312, 465-469
model matrix, 121
PerspectiveView, 262, 265
multiplication, 103, 121, 205-206
projection matrix
handedness of coordinate systems, 462-464
quadrangular pryamid, 260-261
Matrix4 object, supported methods and properties, 118
Matrix4.setOrtho() function, 453
Matrix4.setPerspective() function, 453
matrix data types in GLSL ES, 198-206
access to components, 201-204
assignment of values, 199-201
constructors, 199-201
operators, 204-206
matrix functions, 216, 448
max() function, 445
maxtrixCompMult() function, 448
mediump precision qualifier, 220
member access in GLSL ES
arrays, 209
structures, 207
methods. *See also* functions
for Matrix4 object, 118
of typed arrays, 79
WebGL methods, naming conventions, 48-49
min() function, 445
MIPMAP texture format, 176
mix() function, 446
model matrix, 121
PerspectiveView, 262, 265
model transformation, 121
mod() function, 444
modifying HTML elements using JavaScript, 247-248
mouse
drawing points, 50-58
ClickedPoints.js, 50-52
event handling, 53-57
registering event handlers, 52-53
rotating objects, 357
moving shapes, 92-96
MTL file format (3D models), 418
MultiAttributeColor.js, 147-150
MultiAttributeSize_Interleaved.js, 142-145
MultiAttributeSize.js, 139-140
multijoint model
MultiJointModel.js, 335-338
objects composed of other objects, 334
MultiJointModel.js, 335-338
MultiJointMode_segment.js, 340-342
multiple buffer objects, creating, 140-141
multiple points, drawing, 68-85
multiple textures, mapping to shapes, 183-190
multiple transformations, 115-124
in animation, 135-136
combining, 119-121
cuon-matrix.js, 116
RotatedTranslatedTriangle.js, 121-124
RotatedTriangle_Matrix4.js, 117-119
multiple vertices
basic shapes, drawing, 85-91
drawing, 68-85
assigning buffer objects to attribute variables, 79-81
binding buffer objects to targets, 75-76

buffer object usage, 72-74
creating buffer objects, 74-75
enabling attribute variable assignments, 81-82
gl.drawArrays() function, 82-83
writing data to buffer objects, 76-78
multiplication
of matrices, 121
of vectors and matrices, 103, 205-206
MultiPoint.js, 70-72
MultiTexture.js, 184-190

N

naming conventions
GLSL ES variables, 194
variables, 43
WebGL methods, 48-49
near value, changing, 250-251
normal orientation of a surface, 299-301
normalize() function, 447
notEqual() function, 450
not() function, 450
numerical values in GLSL ES, 194

O

objects
composed of other objects
draw() function, 332-334
drawing, 324-325
draw segments, 339-344
hierarchical structure, 325-326
JointModel.js, 328-332
manipulating, 324-325
multijoint model, 334
single joint model, 326-327
rotation
implementing, 358
with mouse, 357
RotateObject.js, 358-360
selection, 360-362
face of objects, 365
implementing, 361-362
PickObject.js, 362-365
OBJ file format (3D models), 417
OBJViewer.js, 419-421
parser code, 423-430
onLoadShader() function, 472
OpenGL
color buffers, swapping, 437
functions, naming conventions, 48-49
in history of WebGL, 5
WebGL and, 5
OpenGL ES (Embedded Systems), 5-6, 30
OpenGL shading language (GLSL), 6
operator precedence in GLSL ES, 210
operators in GLSL ES
on arrays, 209
on structures, 208
on variables, 197-198
on vector and matrix data types, 204-206
order of execution in GLSL ES, 193
orientation of a surface
calculating diffuse reflection, 297-299
normal, 299-301

origin
in coordinate systems, 55
in local coordinate system, 474-475
in world coordinate system, 475-477
origins of WebGL, 5-6
orthographic projection matrix, 252-253, 261, 453
OrthoView.html, 245-246
OrthoView.js, 246-247

P

parameter qualifiers in GLSL ES functions, 214-215
parameters of texture objects, setting, 174-177
parser code (OBJViewer.js), 423-430
passing data
to fragment shaders
texture units, 179-180
with varying variable, 146-151
to vertex shaders, 137-151. *See also* drawing; rectangles; shapes; triangles
color changes, 146-151
creating multiple buffer objects, 140-141
interleaving, 141-145
MultiAttributeSize.js, 139-140
pasting images on rectangles, 160-183
activating texture units, 171-172
assigning texture images to texture objects, 177-179
binding texture objects to target, 173-174
changing texture coordinates, 182-183
flipping image y-axis, 170-171
mapping texture and vertex coordinates, 162-163, 166
multiple texture mapping, 183-190
passing coordinates from vertex to fragment shader, 180-181
passing texture unit to fragment shader, 179-180
retrieving texel color in fragment shader, 181-182
setting texture object parameters, 174-177
setting up and loading images, 166-170
texture coordinates, explained, 162
TexturedQuad.js, 163-166
perspective projection matrix, 257, 453
PerspectiveView.js, 255, 260-263
model matrix, 262, 265
PerspectiveView_mvp.js, 263-266
per-vertex operations, 93
PickFace.js, 365-368
PickObject.js, 362-365
point light, 293
point light objects, 314-315
PointLightedCube.js, 315-319
PointLightedCube_perFragment.js, 319-321
points, drawing, 23-50
attribute variables, 41-42
setting value, 45-49
storage location, 44-45
changing point color, 58-66
ClickedPoints.js, 50-52
ColoredPoints.js, 59-61
gl.drawArrays() function, 36-37
HelloPoint1.html, 25

HelloPoint1.js, 25-26
HelloPoint2.js, 42-43
with mouse clicks, 50-58
multiple points, 68-85
registering event handlers, 52-53
shaders
explained, 27-28
fragment shaders, 35-36
initializing, 30-33
vertex shaders, 33-35
uniform variables, 61-62
assigning values, 63-66
storage location, 62-63
WebGL coordinate system 38-39
WebGL program structure, 28-30
point size, attribute variables for, 139-140
popMatrix() function, 338
positive rotation, 96
pow() function, 443
precedence of operators in GLSL ES, 210
precision qualifiers, 62, 219-221
predefined single parameters, 53
preprocessor directives in GLSL ES, 221-223
primitive assembly process. *See* geometric shape assembly
primitives. *See* shapes
process flow
initializing shaders, 31
InitShaders() function, 353-355
JavaScript to WebGL, 27, 438
mouse click event handling, 53-57
multiple vertice drawing, 70
vertex shaders, 248-249
programmable shader functions, 6
ProgramObject.js, 387-391
program objects, 44, 353
attaching shader objects, 350-351
creating, 349-350
linking, 351-352
projection matrices, 453
handedness of coordinate systems, 462-464
quadrangular pryamid, 260-261
properties
Matrix4 object, 118
typed arrays, 79
prototype declarations in GLSL ES functions, 214
publishing 3D graphics applications, ease of, 4
pushMatrix() function, 338

Q

quadrangular pyramid
PerspectiveView.js, 258-260
projection matrix, 260-261
viewing volume, 256-258
visible range, 254-256
qualifiers for parameters in GLSL ES functions, 214-215

R

radians() function, 441
rasterization, 137, 151-155

rectangles. *See also* shapes; triangles
drawing, 13-16, 89-91
pasting images on, 160-183
activating texture units, 171-172
assigning texture images to texture objects, 177-179
binding texture objects to target, 173-174
changing texture coordinates, 182-183
flipping image y-axis, 170-171
mapping texture and vertex coordinates, 162-163, 166
multiple texture mapping, 183-190
passing texture coordinates from vertex to fragment shader, 180-181
passing texture unit to fragment shader, 179-180
retrieving texel color in fragment shader, 181-182
setting texture object parameters, 174-177
setting up and loading images, 166-170
texture coordinates, explained, 162
TexturedQuad.js, 163-166
reflected light, 294-296
ambient reflection, 295-296
diffuse reflection, 294-295
reflect() function, 448
refract() function, 448
registering event handlers, 52-53
renderbuffer objects, 392-393
binding, 399-400
creating, 398
setting to framebuffer objects, 401-402
rendering context. *See* context, retrieving
requestAnimationFrame() function, 130-133
reserved words in GLSL ES, 194-195
resizing rotation matrix, 106-107
restoring clipped parts of triangles, 251-253
retrieving
<canvas> element, 14, 19
context, 15
for WebGL, 20-21
storage location of uniform variables, 62-63
texel color in fragment shader, 181-182
RGBA components, 409
RGBA format, 15
RGB format, 15
right-handedness of coordinate systems, 38
in default behavior, 455-464
right-hand-rule rotation, 96
RotatedTranslatedTriangle.js, 121-124
RotatedTriangle.js, 98-102
RotatedTriangle_Matrix.js, 107-110
RotatedTriangle_Matrix4.html, 116
RotatedTriangle_Matrix4.js, 117-119
LookAtTriangles.js versus, 232-233
rotated triangles from specified positions, 234-235
RotateObject.js, 358-360
rotating
objects
implementing, 358
with mouse, 357
RotateObject.js, 358-360

shapes, 96-102
calling drawing function, 129-130
combining multiple transformations, 119-121
draw() function, 130-131
multiple transformations in, 135-136
requestAnimationFrame() function, 131-133
RotatingTriangle.js, 126-129
transformation matrix, 102-105
updating rotation angle, 133-135
triangles, 107-110
RotatingTranslatedTriangle.js, 135-136
RotatingTriangle_contextLost.js, 432-434
RotatingTriangle.js, 126-129
calling drawing function, 129-130
draw() function, 130-131
requestAnimationFrame() function, 131-133
updating rotation angle, 133-135
rotation angle, updating, 133-135
rotation matrix
creating, 102-105
defined, 104
inverse transpose matrix and, 465-469
resizing, 106-107
RotatedTriangle_Matrix.js, 107-110
RoundedPoint.js, 378-379
rounded points, 377
implementing, 377-378
RoundedPoint.js, 378-379
row major order, 109-110

S

sample programs
BlendedCube.js, 384
ClickedPoints.js, 50-52
ColoredCube.js, 285-289
ColoredPoints.js, 59-61
ColoredTriangle.js, 159
CoordinateSystem.js, 456-459
CoordinateSystem_viewVolume.js, 461
cuon-matrix.js, 116
cuon-utils.js, 20
DepthBuffer.js, 272-273
DrawRectangle.html, 11-13
DrawRectangle.js, 13-16
Fog.js, 374-376
Fog_w.js, 376-377
FramebufferObject.js, 395-396, 403
HelloCanvas.html, 17-18
HelloCanvas.js, 18-23
HelloCube.js, 278-281
HelloPoint1.html, 25
HelloPoint1.js, 25-26
HelloPoint2.js, 42-43
HelloQuad.js, 89-91
HelloTriangle.js, 85-86, 151-152
HUD.html, 369-370
HUD.js, 370-372
JointModel.js, 328-332
LightedCube_ambient.js, 308-309
LightedCube.js, 302-303
processing in JavaScript, 306
processing in vertex shader, 304-305
LightedTranslatedRotatedCube.js, 312-314

LookAtBlendedTriangles.js, 381-382
LookAtRotatedTriangles.js, 235-238
LookAtRotatedTriangles_mvMatrix.js, 237
LookAtTriangles.js, 229-233
LookAtTrianglesWithKeys.js, 238-241
LookAtTrianglesWithKeys_ViewVolume.js, 251-253
MultiAttributeColor.js, 147-150
MultiAttributeSize_Interleaved.js, 142-145
MultiAttributeSize.js, 139-140
MultiJointModel.js, 335-338
MultiJointMode_segment.js, 340-342
MultiPoint.js, 70-72
MultiTexture.js, 184-190
OBJViewer.js, 419-421
  parser code, 423-430
OrthoView.html, 245-246
OrthoView.js, 246-247
PerspectiveView.js, 255, 260-263
  model matrix, 262, 265
PerspectiveView_mvp.js, 263-266
PickFace.js, 365-368
PickObject.js, 362-365
PointLightedCube.js, 315-319
PointLightedCube_perFragment.js, 319-321
ProgramObject.js, 387-391
RotatedTranslatedTriangle.js, 121-124
RotatedTriangle.js, 98-102
RotatedTriangle_Matrix.js, 107-110
RotatedTriangle_Matrix4.html, 116
RotatedTriangle_Matrix4.js, 117-119
  LookAtTriangles.js versus, 232-233
RotateObject.js, 358-360
RotatingTranslatedTriangle.js, 135-136
RotatingTriangle_contextLost.js, 432-434
RotatingTriangle.js, 126-129
  calling drawing function, 129-130
  draw() function, 130-131
  requestAnimationFrame() function, 131-133
  updating rotation angle, 133-135
RoundedPoint.js, 378-379
Shadow_highp.js, 413-414
Shadow.js, 406-412
TexturedQuad.js, 163-166
TranslatedTriangle.js, 92-96
samplers in GLSL ES, 209-210
saving color buffer content, 56
scaling matrix
  handedness of coordinate systems, 464
  inverse transpose matrix and, 465-469
scaling shapes, 111-113
selecting
  face of objects, 365-368
  objects, 360-365
semicolon (;) in GLSL ES, 193
setInterval() function, 131
setLookAt() function, 228-229
setOrtho() function, 243
setPerspective() function, 257
setRotate() function, 117, 131
shader objects
  attaching to program objects, 350-351
  compiling, 347-349
  creating
    gl.createShader() function, 345-346
    program objects, 349-350

deleting, 346
InitShaders() function, 344-345
linking program objects, 351-352
storing shader source code, 346-347
shader programs, loading from files, 471-472
shaders, 6, 25
explained, 27-28
fragment shaders, 27
drawing points, 35-36
example of, 192
geometric shape assembly and rasterization, 151-155
invoking, 155
passing data to, 61-62, 146-151
passing texture coordinates to, 180-181
passing texture units to, 179-180
program structure, 29-30
retrieving texel color in, 181-182
varying variables and interpolation process, 157-160
verifying invocation, 156-157
GLSL ES. *See* GLSL ES
initializing, 30-33
InitShaders() function, 344-345
source code, storing, 346-347
vertex shaders, 27, 232
drawing points, 33-35
example of, 192
geometric shape assembly and rasterization, 151-155
passing data to, 41-42, 137-151. *See also* drawing; rectangles; shapes; triangles
passing texture coordinates to fragment shaders, 180-181
program structure, 29-30
WebGL program structure, 28-30
shading
3D objects, 292
adding ambient light, 307-308
calculating color per fragment, 319
directional light and diffuse reflection, 296-297
shading languages, 6
Shadow_highp.js, 413-414
Shadow.js, 406-412
shadow maps, 405
shadows
implementing, 405-406
increasing precision, 412
Shadow_highp.js, 413-414
Shadow.js, 406-412
shapes. *See also* rectangles; triangles
animation, 124-136
calling drawing function, 129-130
draw() function, 130-131
multiple transformations in, 135-136
requestAnimationFrame() function, 131-133
RotatingTriangle.js, 126-129
updating rotation angle, 133-135
drawing, 85-91
HelloTriangle.js, 85-86
list of, 87-88
multiple vertices, drawing, 68-85

rotating, 96-102
RotatedTriangle_Matrix.js, 107-110
transformation matrix, 102-105
scaling, 111-113
transformation libraries, 115-124
combining multiple transformations, 119-121
cuon-matrix.js, 116
RotatedTranslatedTriangle.js, 121-124
RotatedTriangle_Matrix4.js, 117-119
translating, 92-96
combining with rotation, 111
transformation matrix, 105-106
sign() function, 444
sin() function, 442
single joint model, objects composed of other objects, 326-327
smoothstep() function, 446
sqrt() function, 443
stencil buffer, 22
step() function, 446
storage location
attribute variables, 44-45
uniform variables, 62-63
storage qualifiers, 43, 217-219
attribute variables, 218
const, 217
uniform variables, 218
varying variables, 219
storing shader source code, 346-347
StringParser object, 426
striped patterns, 409
structures in GLSL ES, 207-208
access to members, 207
assignment of values, 207
constructors, 207
operators, 208
swapping color buffers, 437
switching shaders, 386
implementing, 387
ProgramObject.js, 387-391
swizzling, 202

T

tan() function, 442
targets, binding texture objects to, 173-174
texels, 160
data formats, 178
data types, 179
retrieving color in fragment shader, 181-182
text editors, 3D graphics development with, 3-4
texture2D() function, 181-182, 451
texture2DLod() function, 451
texture2DProj() function, 451
texture2DProjLod() function, 451
texture coordinates
changing, 182-183
explained, 162
flipping image y-axis, 170-171
mapping to vertex coordinates, 162-163, 166
passing from vertex to fragment shader, 180-181
textureCube() function, 451
textureCubeLod() function, 451
TexturedQuad.js, 163-166

texture images, 392
- FramebufferObject.js, 395-396
- framebuffer objects, 392-393
  - creating, 397
- implementing, 394
- renderbuffer objects, 392-393
  - creating, 398

texture lookup functions, 216, 451
texture mapping, 160-183
- activating texture units, 171-172
- assigning texture images to texture objects, 177-179
- binding texture objects to target, 173-174
- changing texture coordinates, 182-183
- flipping image y-axis, 170-171
- mapping texture and vertex coordinates, 162-163, 166
- with multiple textures, 183-190
- passing coordinates from vertex to fragment shader, 180-181
- passing texture unit to fragment shader, 179-180
- retrieving texel color in fragment shader, 181-182
- setting texture object parameters, 174-177
- setting up and loading images, 166-170
- texture coordinates, explained, 162
- TexturedQuad.js, 163-166

texture objects, 170
- assigning texture images to, 177-179
- binding to target, 173-174
- creating, 397-398
- setting parameters, 174-177
- setting to framebuffer objects, 400-401

texture units
- activating, 171-172
- passing to fragment shader, 179-180
- pasting multiple, 183-190

tick() function, 129-130
transformation libraries, 115-124
- combining multiple transformations, 119-121
- cuon-matrix.js, 116
- RotatedTranslatedTriangle.js, 121-124
- RotatedTriangle_Matrix4.js, 117-119

transformation matrix
- defined, 103
- inverse transpose matrix and, 465-469
- rotating shapes, 102-105
- scaling shapes, 111-113
- translating shapes, 105-106

transformations
- coordinate systems and, 477
- defined, 91
- multiple transformations in animation, 135-136
- world transformation, 476

translated-rotated objects
- inverse transpose matrix, 311-312
- lighting, 310-311

TranslatedTriangle.js, 92-96
translating
- shapes, 92-96
  - combining multiple transformations, 119-121
  - transformation matrix, 105-106
- triangles, 111

translation matrix
- combining with rotation matrix, 111
- creating, 105-106

defined, 106
inverse transpose matrix and, 465-469
triangles, 225-226. *See also* rectangles; shapes
coloring vertices different colors, 151-160
geometric shape assembly and rasterization, 151-155
invoking fragment shader, 155
varying variables and interpolation process, 157-160
verifying fragment shader invocation, 156-157
combining multiple transformations, 119-121
drawing, 85-91
restoring clipped parts, 251-253
rotating, 96-102, 107-110, 234-235
RotatingTranslatedTriangle.js, 135-136
RotatingTriangle.js, 126-129
calling drawing function, 129-130
draw() function, 130-131
requestAnimationFrame() function, 131-133
updating rotation angle, 133-135
TranslatedTriangle.js, 92-96
translating, 92-96, 111
trigonometry functions, 216, 441-442
type conversion in GLSL ES, 196-197
typed arrays, 78-79
typed programming languages, 34
type sensitivity in GLSL ES, 195

U

#undef preprocessor directive, 222
uniform variables, 61-62, 217-218
assigning values to, 63-66
retrieving storage location, 62-63
u_NormalMatrix, 314
updating rotation angle, 133-135
up direction, 228
user-defined objects (3D models), 422-423

V

values, assigning
to attribute variables, 45-49
to uniform variables, 63-66
variables
attribute variables, 218
declaring, 43
explained, 41-42
setting value, 45-49
storage location, 44-45
in fragment shaders, 36
in GLSL ES
arrays, 208-209
assignment of values, 196-197
data types for, 34, 196
global and local variables, 216
keywords and reserved words, 194-195
naming conventions, 194

operator precedence, 210
operators on, 197-198
precision qualifiers, 219-221
samplers, 209-210
storage qualifiers, 217-219
structures, 207-208
type conversion, 196-197
type sensitivity, 195
vector and matrix types, 198-206
naming conventions, 43
uniform variables, 61-62, 218
assigning values to, 63-66
retrieving storage location, 62-63
in vertex shaders, 33
varying variables, 217, 219
color changes with, 146-151
interpolation process and, 157-160
vec4() function, 34-35
vector data types in GLSL ES, 198-206
access to components, 201-204
assignment of values, 199-201
constructors, 199-201
operators, 204-206
vector functions, 48, 216, 449
vector multiplication, 103, 205-206
#version preprocessor directive, 223
vertex coordinates, mapping to texture coordinates, 162-163, 166
vertex shaders, 27, 232
drawing points, 33-35
example of, 192
geometric shape assembly and rasterization, 151-155
passing data to, 41-42, 137-151. *See also* drawing; rectangles; shapes; triangles
color changes, 146-151
creating multiple buffer objects, 140-141
interleaving, 141-145
MultiAttributeSize.js, 139-140
passing texture coordinates to fragment shaders, 180-181
program structure, 29-30
vertices, 27
basic shapes
drawing, 85-91
rotating, 96-102
scaling, 111-113
translating, 92-96
coloring different colors, 151-160
geometric shape assembly and rasterization, 151-155
invoking fragment shader, 155
varying variables and interpolation process, 157-160
verifying fragment shader invocation, 156-157
multiple vertices, drawing, 68-85
transformation matrix
rotating shapes, 102-105
translating shapes, 105-106
view matrix, 229, 231
viewing console, 14
viewing direction, 226-227
eye point, 228
look-at point, 228

LookAtRotatedTriangles.js, 235-236
LookAtTriangles.js, 229-232
specifying, 226-227
up direction, 228
viewing volume
clip coordinate system and, 460-462
quadrangular pyramid, 256-258
visible range, 242-243
visible range, 241-242
defining box-shaped viewing volume, 243-244
eye point, 241
quadrangular pyramid, 254-256
viewing volume, 242-243

W

web browsers
<canvas> element support, 12
console, viewing, 14
enabling local file access, 161
functionality in 3D graphics applications, 5
JavaScript to WebGL processing flow, 27, 438
WebGL settings, 479-480
WebGL
advantages of, 3-5
application structure, 6-7
browser settings, 479-480
color, setting, 21-23
color buffer, drawing to, 437-439
coordinate system, 38-39
clip coordinate system and viewing volume, 460-462
CoordinateSystem.js, 456-459
handedness in default behavior, 455-464
Hidden Surface Removal tool, 459-460
projection matrices for, 462-464
transforming <canvas> element coordinates to, 54-57
defined, 1-2
JavaScript processing flow, 27, 438
methods, naming conventions, 48-49
OpenGL and, 5
origins of, 5-6
processing flow for initializing shaders, 31
program structure for shaders, 28-30
rendering context, retrieving, 20-21
web pages (3DoverWeb), displaying 3D objects, 372
world coordinate system, 475-477
world transformation, 476
writing
data to buffer objects, 76-78
Hello Cube vertex coordinates, colors, and indices in the buffer object, 281-284
w value (fog), 376-377

X-Z

y-axis, flipping, 170-171
z fighting
background objects, 273-275
foreground objects, 273-275
Zfighting.js, 274-275