



Build Your Own 2D Game Engine and Create Great Web Games

Using HTML5, JavaScript, and WebGL2

Second Edition

Kelvin Sung
Jebediah Pavleas
Matthew Munson
Jason Pace

Apress®

Build Your Own 2D Game Engine and Create Great Web Games

Using HTML5, JavaScript,
and WebGL2

Second Edition

Kelvin Sung
Jebediah Pavleas
Matthew Munson
Jason Pace

With

Original Dye character designs by Nathan Evers
Other game character and art design by Kasey Quevedo
Figures and illustrations by Clover Wai

Apress®

Build Your Own 2D Game Engine and Create Great Web Games: Using HTML5, JavaScript, and WebGL2

Kelvin Sung Jebediah Pavleas
Bothell, WA, USA Kenmore, WA, USA

Matthew Munson
Lake Forest Park, WA, USA

Jason Pace
Portland, OR, USA

ISBN-13 (pbk): 978-1-4842-7376-0 ISBN-13 (electronic): 978-1-4842-7377-7
<https://doi.org/10.1007/978-1-4842-7377-7>

Copyright © 2022 by Kelvin Sung, Jebediah Pavleas, Matthew Munson, and Jason Pace

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahn
Acquisitions Editor: Spandana Chatterjee
Development Editor: Laura Berendson
Coordinating Editors: Shrikant Vishwakarma

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Apress Media, LLC, 1 New York Plaza, New York, NY 10004, U.S.A. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at <http://www.apress.com/bulk-sales>.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484273760. For more detailed information, please visit <http://www.apress.com/source-code>.

Printed on acid-free paper

<i>To my wife, Clover, and our girls, Jean and Ruth, for completing my life.</i>	62
— <i>Kelvin Sung</i>	63
<i>To my family, for their eternal support throughout my life.</i>	64
— <i>Jebediah Pavleas</i>	65
<i>To my mom, Linda, for showing me the value of having fun at work.</i>	66
— <i>Jason Pace</i>	67

Uncorrected Proof

Table of Contents

		68
About the Authors.....	xiii	69
About the Technical Reviewers	xv	70
Acknowledgments	xvii	71
Introduction	xix	72
Chapter 1: Introducing 2D Game Engine Development with JavaScript.....	1	73
The Technologies	2	74
Setting Up Your Development Environment	4	75
Downloading and Installing JavaScript Syntax Checker	5	76
Downloading and Installing LiveServer	5	77
Working in the VS Code Development Environment.....	5	78
Creating an HTML5 Project in VS Code.....	6	79
How to Use This Book	10	80
How Do You Make a Great Video Game?	12	81
References.....	17	82
Technologies.....	18	83
Chapter 2: Working with HTML5 and WebGL	19	84
Introduction.....	19	85
Canvas for Drawing.....	20	86
The HTML5 Canvas Project.....	20	87
Separating HTML and JavaScript.....	24	88
The JavaScript Source File Project.....	24	89
Observations.....	29	90

TABLE OF CONTENTS

91	Elementary Drawing with WebGL.....	29
92	The Draw One Square Project	30
93	Observations.....	41
94	Abstraction with JavaScript Classes.....	41
95	The JavaScript Objects Project.....	42
96	Observations.....	51
97	Separating GLSL from HTML.....	51
98	The Shader Source File Project	52
99	Source Code Organization	57
100	Changing the Shader and Controlling the Color.....	57
101	The Parameterized Fragment Shader Project.....	57
102	Summary.....	61
103	Chapter 3: Drawing Objects in the World.....	63
104	Introduction.....	63
105	Encapsulating Drawing	64
106	The Renderable Objects Project	65
107	Observations.....	73
108	Transforming a Renderable Object.....	73
109	Matrices as Transform Operators	74
110	The glMatrix Library	77
111	The Matrix Transform Project	78
112	Observations.....	83
113	Encapsulating the Transform Operator	84
114	The Transform Objects Project	84
115	The Camera Transform and Viewports	89
116	Coordinate Systems and Transformations.....	90
117	The Camera Transform and Viewport Project	93
118	The Camera.....	101
119	The Camera Objects Project	102
120	Summary.....	110

TABLE OF CONTENTS

Chapter 4: Implementing Common Components of Video Games	111	121
Introduction.....	111	122
The Game Loop	112	123
Typical Game Loop Implementations.....	113	124
The Game Loop Project.....	114	125
Keyboard Input.....	123	126
The Keyboard Support Project.....	123	127
Resource Management and Asynchronous Loading	131	128
The Resource Map and Shader Loader Project	133	129
Game Level from a Scene File	144	130
The Scene File Project.....	144	131
Scene Object: Client Interface to the Game Engine.....	156	132
The Scene Objects Project.....	157	133
Audio.....	170	134
The Audio Support Project.....	171	135
Summary.....	183	136
Game Design Considerations.....	183	137
Chapter 5: Working with Textures, Sprites, and Fonts.....	185	138
Introduction.....	185	139
Texture Mapping and Texture Coordinates	187	140
The Texture Shaders Project.....	188	141
Drawing with Sprite Sheets	218	142
The Sprite Shaders Project.....	220	143
Sprite Animations.....	233	144
Overview of Animated Sprite Sheets	233	145
The Sprite Animation Project.....	234	146
Fonts and Drawing of Text	245	147
Bitmap Fonts	245	148
The Font Support Project.....	247	149
Summary.....	263	150
Game Design Considerations.....	263	151

TABLE OF CONTENTS

152	Chapter 6: Defining Behaviors and Detecting Collisions	267
153	Introduction.....	267
154	Game Objects.....	268
155	The Game Objects Project	269
156	Creating a Chase Behavior.....	280
157	Vectors Review	281
158	The Front and Chase Project	286
159	Collisions Between GameObjects	295
160	Axis-Aligned Bounding Box (AABB)	295
161	The Bounding Box and Collisions Project	296
162	Per-Pixel Collisions	304
163	The Per-Pixel Collisions Project.....	305
164	Generalized Per-Pixel Collisions.....	318
165	Vector Review: Components and Decomposition	318
166	The General Pixel Collisions Project	320
167	Per-Pixel Collisions for Sprites.....	326
168	The Sprite Pixel Collisions Project.....	326
169	Summary.....	331
170	Game Design Considerations.....	332
171	Chapter 7: Manipulating the Camera.....	335
172	Introduction.....	335
173	Camera Manipulations	337
174	The Camera Manipulations Project.....	337
175	Interpolation.....	347
176	The Camera Interpolations Project	349
177	Camera Shake and Object Oscillation Effects.....	357
178	The Camera Shake and Object Oscillate Project	358
179	Multiple Cameras	369
180	The Multiple Cameras Project	370

TABLE OF CONTENTS

Mouse Input Through Cameras	377	181
The Mouse Input Project.....	379	182
Summary.....	388	183
Game Design Considerations.....	389	184
Chapter 8: Implementing Illumination and Shadow	397	185
Introduction.....	397	186
Overview of Illumination and GLSL Implementation	398	187
Ambient Light.....	399	188
The Global Ambient Project	400	189
Light Source.....	410	190
GLSL Implementation and Integration into the Game Engine	411	191
The Simple Light Shader Project	412	192
Multiple Light Sources and Distance Attenuation	428	193
The Multiple Lights Project.....	429	194
Diffuse Reflection and Normal Mapping	441	195
The Normal Maps and Illumination Shaders Project	446	196
Specular Reflection and Materials.....	456	197
Integration of Material in the Game Engine and GLSL Shaders.....	459	198
The Material and Specularity Project	460	199
Light Source Types.....	473	200
The Directional and Spotlights Project	474	201
Shadow Simulation	486	202
The Shadow Simulation Algorithm	490	203
The Shadow Shaders Project	490	204
Summary.....	513	205
Game Design Considerations.....	514	206

TABLE OF CONTENTS

207	Chapter 9: Simulating the World with RigidShapes.....	525
208	Introduction.....	525
209	Chapter Overview	527
210	Rigid Shapes and Bounds	529
211	The Rigid Shapes and Bounds Project.....	529
212	Collision Detection	548
213	Broad and Narrow Phase Methods.....	549
214	Collision Information.....	550
215	The Circle Collisions and CollisionInfo Project.....	551
216	Separating Axis Theorem.....	563
217	The Rectangle Collisions Project	569
218	Collision Between Rectangles and Circles	576
219	The Rectangle and Circle Collisions Project	577
220	Movement	585
221	Explicit Euler Integration	586
222	Symplectic Euler Integration	587
223	The Rigid Shape Movements Project.....	587
224	Interpenetration of Colliding Objects.....	597
225	Collision Position Correction.....	599
226	The Collision Position Correction Project.....	600
227	Collision Resolution.....	606
228	The Impulse Method	607
229	The Collision Resolution Project	613
230	Angular Components of Collision Responses.....	619
231	Collisions with Rotation Consideration	619
232	Relative Velocity with Rotation	621
233	Impulse Method with Rotation.....	622
234	The Collision Angular Resolution Project	626
235	Observations.....	631
236	Summary.....	632
237	Game Design Considerations.....	633

TABLE OF CONTENTS

Chapter 10: Creating Effects with Particle Systems	639	238
Introduction.....	639	239
Particles and Particle Systems	640	240
The Particles Project.....	641	241
Particle Collisions.....	656	242
The Particle Collisions Project.....	656	243
Particle Emitters	663	244
The Particle Emitters Project.....	663	245
Summary.....	669	246
Game Design Considerations.....	670	247
Chapter 11: Supporting Camera Background	671	248
Introduction.....	671	249
Tiling of the Background	673	250
The Tiled Objects Project.....	674	251
Simulating Motion Parallax with Parallax Scrolling	681	252
The Parallax Objects Project.....	683	253
Layer Management	689	254
The Layer Manager Project.....	690	255
Summary.....	698	256
Game Design Considerations.....	698	257
Chapter 12: Building a Sample Game: From Design to Completion	703	258
Part 1: Refining the Concept	704	259
Part 2: Integrating a Setting	707	260
Contextual Images Bring the Setting to Life	707	261
Defining the Playable Space.....	708	262
Adding Layout to the Playable Space	710	263
Tuning the Challenge and Adding Fun	711	264
Further Tuning: Introducing Enemies.....	713	265
General Considerations	714	266

TABLE OF CONTENTS

267	Part 3: Integrating Additional Design Elements.....	715
268	Visual Design.....	716
269	Game Audio	717
270	Interaction Model.....	718
271	Game Systems and Meta-game	718
272	User Interface (UI) Design.....	720
273	Game Narrative.....	721
274	Bonus Content: Adding a Second Stage to the Level	723
275	Summary.....	724
276	Index.....	727

About the Authors

277

Kelvin Sung is a Professor with the Computing and Software Systems Division at the University of Washington Bothell (UWB). He received his Ph.D. in Computer Science from the University of Illinois at Urbana-Champaign. Kelvin's background is in computer graphics, hardware, and machine architecture. He came to UWB from Alias|Wavefront (now part of Autodesk), where he played a key role in designing and implementing the Maya Renderer, an Academy Award-winning image generation system. At UWB, funded by Microsoft Research and the National Science Foundation, Kelvin's work focuses on the intersection of video game mechanics, solutions to real-world problems, and supports for remote collaboration. Together with his students and colleagues, Kelvin has co-authored five books: one on computer graphics (*Essentials of Interactive Computer Graphics: Concepts and Implementation*, A.K. Peters, 2008) and the others on 2D game engines (*Learn 2D Game Development with C#*, Apress, 2013; *Build Your Own 2D Game Engine and Create Great Web Games*, Apress, October 2015; *Building a 2D Game Physics Engine*, Apress, 2016; and *Basic Math for Game Development with Unity 3D*, Apress, 2019).

278
279
280
281
282
283
284
285
286
287
288
289
290
291
292

this figure will be printed in b/w



Jebediah Pavleas received his Master of Science Degree in Computer Science and Software Engineering from the University of Washington Bothell (UWB) in 2016. He also received a Bachelor of Science Degree from UWB in 2012 and was the recipient of the Chancellor's Medal for his class. During his graduate program, Jebediah interned for Microsoft Research's Enable team where he contributed to their Eye-Gaze Wheelchair project (a wheelchair driven with only your eyes for those with ALS). He has co-authored three books on 2D games and game engines (*Learn 2D Game Development with C#*, Apress, 2013; *Build Your Own 2D Game Engine and Create Great Web Games*, Apress, October 2015; *Building a 2D Game Physics Engine*, Apress, 2016). During his time at UWB, his projects included an interactive math application that utilizes Microsoft's Kinect sensor

293
294
295
296
297
298
299
300
301
302
303
304
305
306

ABOUT THE AUTHORS

307 to teach algebra called Kinect Math. Relating to this and other projects, he co-authored
308 publications in *IEEE Computer* and *The Journal of Computing Sciences in Colleges*
309 (CCSC). Jebediah enjoys designing, building, and playing games of all kinds as well as
310 adapting technology for improved accessibility for himself and others.

AU1

311 **Matthew Munson** is a graduate student in the Computer Science and Software
312 Engineering program at the University of Washington Bothell. He received
313 undergraduate degrees in Computer Science and Software Engineering and Mechanical
314 Engineering at the University of Washington Bothell in 2020. Matthew is interested in
315 operating system development, networking, and embedded systems. As a research
316 assistant, Matthew used cloud computing to analyze years of audio data recorded by
317 hydrophones off the Oregon coast. This data was used to study the effects of climate
318 change and shipping noise on marine mammals. Currently, Matthew is working on a
319 networked augmented reality library that focuses on allowing users to view the same
320 virtual scene from different perspectives.

AU2

321 **Jason Pace** contributed to a wide range of games as a producer, designer, and creative
322 director over 15 years in the interactive entertainment industry, from ultra-casual
323 puzzlers on mobile to Halo on Xbox. As a designer, Jason builds game mechanics and
324 systems that start from a simple palette of thoughtful interactions (known as the core
325 gameplay loop), progressively introducing variety and complexity to create interactive
326 experiences that engage and delight players while maintaining focus on what makes
327 each e-game uniquely fun.

About the Technical Reviewers

328

this figure will be printed in b/w



Yusuf Pisan is an Associate Teaching Professor in the School of Computing & Software Systems Division at the University of Washington Bothell. Previously, he has worked at the University of Technology, Sydney, and has been a visiting professor at Harvey Mudd College, University of Southern California, Worcester Polytechnic Institute (WPI), and IT University of Copenhagen (ITU).

329
330
331
332
333
334
335

His research interests include enabling technologies for computer games, the design of virtual environments that support collaborative work, and computer science education. He founded the Australasian Conference on Interactive Entertainment conference series and helped foster the Australian games community. His list of publications can be found at Google Scholar.

336
337
338
339
340
341

Yusuf has a Ph.D. in Artificial Intelligence from Northwestern University. Before moving to Seattle in 2017, Yusuf lived in the Chicago area for 10 years and Sydney for 20 years.

342
343

For more information, see <https://pisanorg.github.io/yusuf/>.

344

this figure will be printed in b/w



Yogendra Sharma is a developer with experience in the architecture, design, and development of scalable and distributed applications, with a core interest in Microservices and Spring. He currently works as an IoT and Cloud Architect at Intelizign Engineering Services Pvt. Ltd., Pune.

345
346
347
348
349

He also has hands-on experience in technologies such as AWS, IoT, Python, J2SE, J2EE, NodeJS, VueJs, Angular, MongoDB, and Docker.

350
351
352

He constantly explores technical novelties, and he is open-minded and eager to learn about new technologies and frameworks. He has reviewed several books and video courses published by Packt.

353
354
355
356

Acknowledgments

357

This book project was a direct result of the authors learning from building games for the Game-Themed CS1/2: Empowering the Faculty project, funded by the Transforming Undergraduate Education in Science, Technology, Engineering, and Mathematics (TUES) Program, National Science Foundation (NSF) (award number DUE-1140410). We would like to thank NSF officers Suzanne Westbrook for believing in our project and Jane Prey, Valerie Bar, and Paul Tymann for their encouragements.

358
359
360
361
362
363

This second edition is encouraged by many students and collaborators. In particular, students from CSS452: Game Engine Development (see <https://myuwbclasses.github.io/CSS452/>) at the University of Washington Bothell have been the most critical, demanding, and yet supportive. Through the many games and API extension projects (see <https://html5gameengingroup.github.io/GTCS-Engine-Student-Projects/>), it became clear that updates are required of the JavaScript and WebGL (Web Graphics Library) versions, the bottom-line synchronization mechanism, and, most significantly, the coverage of the physics engine. Fernando Arnez, our co-author from the first edition, taught us JavaScript. Yaniv Schwartz pointed us toward JavaScript `async/await` and `promise`. The discussions and collaborations with Huaming Chen and Michael Tanaya contributed directly to the chapter on game engine physics. Akilas Mebrahtom and Donald Hawkins constructed the extra example at the end of Chapter 9 illustrating potential presets for commonly encountered physical materials. The audio volume control was first investigated and integrated by Kyla NeSmith. Nicholas Carpenetti and Kyla NeSmith developed a user interface API for the initial game engine, which unfortunately did not make it into this edition. These and countless other feedbacks have contributed to the quality and improvements of the book's content.

364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380

The hero character Dye and many of the visual and audio assets used throughout the example projects of the book are based on the Dye Hard game, designed for teaching concepts of objects and object-oriented hierarchy. The original Dye Hard development team members included Matthew Kipps, Rodelle Ladia, Chuan Wang, Brian Hecox, Charles Chiou, John Louie, Emmett Scout, Daniel Ly, Elliott White, Christina Jugovic, Rachel Harris, Nathan Evers, Kasey Quevedo, Kaylin Norman-Slack, David Madden, Kyle Kraus, Suzi Zuber, Aina Braxton, Kelvin Sung, Jason Pace, and Rob Nash. Kyle Kraus

381
382
383
384
385
386
387

ACKNOWLEDGMENTS

388 composed the background music used in the Audio Support project from Chapter 4,
389 originally for the Linx game, which was designed to teach loops. The background audio
390 for the game in Chapter 12 was composed by David Madden and arranged by Aina
391 Braxton. Thanks to Clover Wai for the figures and illustrations.

392 We also want to thank Spandana Chatterjee for believing in our ideas, her patience,
393 and continual efficient and effective support. A heartfelt thank-you to Mark Powers, for
394 his diligence and lightning-fast email responses. Mark should learn about and consider
395 the option of sleeping some of the time. Nirmal Selvaraj organized everything and
396 ensured proper progress was ongoing.

397 Finally, we would like to thank Yusuf Pisan for his insightful, effective, and, above all,
398 quick turnaround for the technical review.

399 All opinions, findings, conclusions, and recommendations in this work are those of
400 the authors and do not necessarily reflect the views of the sponsors.

Introduction

401

Welcome to *Build Your Own 2D Game Engine and Create Great Web Games*. Because you have picked up this book, you are likely interested in the details of a game engine and the creation of your own games to be played over the Internet. This book teaches you how to build a 2D game engine by covering the involved technical concepts, demonstrating sample implementations, and showing you how to organize the large number of source code and asset files to support game development. This book also discusses how each covered technical topic area relates to elements of game design so that you can build, play, analyze, and learn about the development of 2D game engines and games. The sample implementations in this book are based on HTML5, JavaScript, and WebGL2, which are technologies that are freely available and supported by virtually all web browsers. After reading this book, the game engine you develop and the associated games will be playable through a web browser from anywhere on the Internet.

This book presents relevant concepts from software engineering, computer graphics, mathematics, physics, game development, and game design—all in the context of building a 2D game engine. The presentations are tightly integrated with the analysis and development of source code; you'll spend much of the book building game-like concept projects that demonstrate the functionality of game engine components. By building on source code introduced early on, the book leads you on a journey through which you will master the basic concepts behind a 2D game engine while simultaneously gaining hands-on experience developing simple but working 2D games. Beginning from Chapter 4, a “Design Considerations” section is included at the end of each chapter to relate the covered technical concepts to elements of game design. By the end of the book, you will be familiar with the concepts and technical details of 2D game engines, feel competent in implementing functionality in a 2D game engine to support commonly encountered 2D game requirements, and capable of considering game engine technical topics in the context of game design elements in building fun and engaging games.

429 **New in the Second Edition**

430 The key additions to the second edition include JavaScript language and WebGL API update
431 and dedicated chapters with substantial details on physics and particle systems components.

432 All examples throughout the entire book are refined for the latest features of the
433 JavaScript language. While some updates are mundane, for example, prototype chain
434 syntax replacements, the latest syntax allows significant improvements in overall
435 presentation and code readability. The new and much cleaner asynchronous support
436 facilitated a completely new resource loading architecture with a single synchronization
437 point for the entire engine (Chapter 4). The WebGL context is updated to connect to
438 WebGL 2.0. The dedicated chapters allow more elaborate and gradual introduction
439 to the complex physics and particle systems components. Detailed mathematical
440 derivations are included where appropriate.

441 **Who Should Read This Book**

442 This book is targeted toward programmers who are familiar with basic object-oriented
443 programming concepts and have a basic to intermediate knowledge of an object-
444 oriented programming language such as Java or C#. For example, if you are a student
445 who has taken a few introductory programming courses, an experienced developer who
446 is new to games and graphics programming, or a self-taught programming enthusiast,
447 you will be able to follow the concepts and code presented in this book with little
448 trouble. If you're new to programming in general, it is suggested that you first become
449 comfortable with the JavaScript programming language and concepts in object-oriented
450 programming before tackling the content provided in this book.

451 **Assumptions**

452 You should be experienced with programming in an object-oriented programming
453 language, such as Java or C#. Knowledge and expertise in JavaScript would be a plus but
454 are not necessary. The examples in this book were created with the assumption that you
455 understand data encapsulation and inheritance. In addition, you should be familiar with
456 basic data structures such as linked lists and dictionaries and be comfortable working
457 with the fundamentals of algebra and geometry, particularly linear equations and
458 coordinate systems.

Who Should Not Read This Book

459

This book is not designed to teach readers how to program, nor does it attempt to explain the intricate details of HTML5, JavaScript, or WebGL2. If you have no prior experience developing software with an object-oriented programming language, you will probably find the examples in this book difficult to follow.

460

461

462

463

On the other hand, if you have an extensive background in game engine development based on other platforms, the content in this book will be too basic; this is a book intended for developers without 2D game engine development experience. However, you might still pick up a few useful tips about 2D game engine and 2D game development for the platforms covered in this book.

464

465

466

467

468

Organization of This Book

469

This book teaches how to develop a game engine by describing the foundational infrastructure, graphics system, game object behaviors, camera manipulations, and a sample game creation based on the engine.

470

471

472

Chapters 2–4 construct the foundational infrastructure of the game engine. Chapter 2 establishes the initial infrastructure by separating the source code system into folders and files that contain the following: JavaScript-specific core engine logics, WebGL2 GLSL-specific shader programs, and HTML5-specific web page contents. This organization allows ongoing engine functionality expansion while maintaining localized source code system changes. For example, only JavaScript source code files need to be modified when introducing enhancements to game object behaviors. Chapter 3 builds the drawing framework to encapsulate and hide the WebGL2 drawing specifics from the rest of the engine. This drawing framework allows the development of game object behaviors without being distracted by how they are drawn. Chapter 4 introduces and integrates core game engine functional components including game loop, keyboard input, efficient resource and game-level loading, and audio support.

473

474

475

476

477

478

479

480

481

482

483

484

Chapters 5–7 present the basic functionality of a game engine: drawing system, behavior and interactions, and camera manipulation. Chapter 5 focuses on working with texture mapping, including sprite sheets, animation with sprite sheets, and the drawing of bitmap fonts. Chapter 6 puts forward abstractions for game objects and their behaviors including per-pixel-accurate collision detection. Chapter 7 details the manipulation and interactions with the camera including programming with multiple cameras and supporting mouse input.

485

486

487

488

489

490

491

INTRODUCTION

492 Chapters 8–11 elevate the introduced functionality to more advanced levels.

493 Chapter 8 covers the simulation of 3D illumination effects in 2D game scenes.

494 Chapter 9 discusses physically based behavior simulations. Chapter 10 presents

495 the basics of particle systems that are suitable for modeling explosions. Chapter 11

496 examines more advanced camera functionality including infinite scrolling through

497 tiling and parallax.

498 Chapter 12 summarizes the book by leading you through the design of a complete

499 game based on the game engine you have developed.

500 Code Samples

501 Every chapter in this book includes examples that let you interactively experiment

502 with and learn the new materials. You can access the source code for all the projects,

503 including the associated assets (images, audio clips, or fonts), by clicking the **Download**

504 **Source Code** button located at www.apress.com/9781484273760. You should see a

505 folder structure that is organized by chapter numbers. Within each folder are subfolders

506 containing NetBeans projects that correspond to sections of this book.

Author Queries

Chapter No.: 0005244106

Queries	Details Required	Author's Response
AU1	Please check if “IEEE Computer” is okay as edited here.	
AU2	Jebediah Payleas’s photo if needed. Unsure how to insert above.	

Uncorrected Proof

CHAPTER 1

Introducing 2D Game Engine Development with JavaScript

Video games are complex, interactive, multimedia software systems. They must, in real time, process player input, simulate the interactions of semiautonomous objects, and generate high-fidelity graphics and audio outputs, all while trying to keep players engaged. Attempts at building a video game can quickly become overwhelming with the need to be well versed in software development as well as in how to create appealing player experiences. The first challenge can be alleviated with a software library, or game engine, that contains a coherent collection of utilities and objects designed specifically for developing video games. The player engagement goal is typically achieved through careful gameplay design and fine-tuning throughout the video game development process. This book is about the design and development of a game engine; it will focus on implementing and hiding the mundane operations of the engine while supporting many complex simulations. Through the projects in this book, you will build a practical game engine for developing video games that are accessible across the Internet.

A game engine relieves game developers from having to implement simple routine tasks such as decoding specific key presses on the keyboard, designing complex algorithms for common operations such as mimicking shadows in a 2D world, and understanding nuances in implementations such as enforcing accuracy tolerance of a physics simulation. Commercial and well-established game engines such as *Unity*, *Unreal Engine*, and *Panda3D* present their systems through a graphical user interface (GUI). Not only does the friendly GUI simplify some of the tedious processes of game design such as creating and placing objects in a level, but more importantly, it ensures that

CHAPTER 1 INTRODUCING 2D GAME ENGINE DEVELOPMENT WITH JAVASCRIPT

these game engines are accessible to creative designers with diverse backgrounds who may find software development specifics distracting.

This book focuses on the core functionality of a game engine independent from a GUI. While a comprehensive GUI system can improve the end-user experience, the implementation requirements can also distract and complicate the fundamentals of a game engine. For example, issues concerning the enforcement of compatible data types in the user interface system, such as restricting objects from a specific class to be assigned as shadow receivers, are important to GUI design but are irrelevant to the core functionality of a game engine.

This book approaches game engine development from two important aspects: programmability and maintainability. As a software library, the interface of the game engine should facilitate programmability by game developers with well-abstracted utility methods and objects that hide simple routine tasks and support complex yet common operations. As a software system, the code base of the game engine should support maintainability with a well-designed infrastructure and well-organized source code systems that enable code reuse, ongoing system upkeep, improvement, and expansion.

This chapter describes the implementation technology and organization of this book. The discussion leads you through the steps of downloading, installing, and setting up the development environment, guides you to build your first HTML5 application, and uses this first application development experience to explain the best approach to reading and learning from this book.

The Technologies

The goal of building a game engine that allows games to be accessible across the World Wide Web is enabled by freely available technologies.

JavaScript is supported by virtually all web browsers because an interpreter is installed on almost every personal computer in the world. As a programming language, JavaScript is dynamically typed, supports inheritance and functions as first-class objects, and is easy to learn with well-established user and developer communities. With the strategic choice of this technology, video games developed based on JavaScript can be accessible by anyone over the Internet through appropriate web browsers. Therefore, JavaScript is one of the best programming languages for developing video games for the masses.

While JavaScript serves as an excellent tool for implementing the game logic and
algorithms, additional technologies in the form of software libraries, or application
programming interfaces (APIs), are necessary to support the user input and media
output requirements. With the goal of building games that are accessible across the
Internet through web browsers, HTML5 and WebGL provide the ideal complementary
input and output APIs.

HTML5 is designed to structure and present content across the Internet. It
includes detailed processing models and the associated APIs to handle user input
and multimedia outputs. These APIs are native to JavaScript and are perfect for
implementing browser-based video games. While HTML5 offers a basic Scalable
Vector Graphics (SVG) API, it does not support the sophistication demanded by video
games for effects such as real-time lighting, explosions, or shadows. The Web Graphics
Library (WebGL) is a JavaScript API designed specifically for the generation of 2D and
3D computer graphics through web browsers. With its support for OpenGL Shading
Language (GLSL) and the ability to access the graphics processing unit (GPU) on client
machines, WebGL has the capability of producing highly complex graphical effects in
real time and is perfect as the graphics API for browser-based video games.

This book is about the concepts and development of a game engine where JavaScript,
HTML5, and WebGL are simply tools for the implementation. The discussion in this
book focuses on applying the technologies to realize the required implementations and
does not try to cover the details of the technologies. For example, in the game engine,
inheritance is implemented with the JavaScript class functionality which is based on
object prototype chain; however, the merits of prototype-based scripting languages are
not discussed. The engine audio cue and background music functionalities are based
on the HTML5 AudioContext interface, and yet its range of capabilities is not described.
The game engine objects are drawn based on WebGL texture maps, while the features of
the WebGL texture subsystem are not presented. The specifics of the technologies would
distract from the game engine discussion. The key learning outcomes of the book are the
concepts and implementation strategies for a game engine and not the details of any of
the technologies. In this way, after reading this book, you will be able to build a similar
game engine based on any comparable set of technologies such as C# and MonoGame,
Java and JOGL, C++ and Direct3D, and so on. If you want to learn more about or brush
up on JavaScript, HTML5, or WebGL, please refer to the references in the “Technologies”
section at the end of this chapter.

92 Setting Up Your Development Environment

93 The game engine you are going to build will be accessible through web browsers that
94 could be running on any operating system (OS). The development environment you are
95 about to set up is also OS agnostic. For simplicity, the following instructions are based
96 on a Windows 10 OS. You should be able to reproduce a similar environment with minor
97 modifications in a Unix-based environment like MacOS or Ubuntu.

98 Your development environment includes an integrated development environment (IDE)
99 and a runtime web browser that is capable of hosting the running game engine. The most
100 convenient systems we have found is the Visual Studio Code (VS Code) IDE with the Google
101 Chrome web browser as runtime environment. Here are the details:

- 102 • **IDE:** All projects in this book are based on VS Code IDE. You
103 can download and install the program from [https://code.
104 visualstudio.com/](https://code.visualstudio.com/).
- 105 • **Runtime environment:** You will execute your video game projects in
106 the Google Chrome web browser. You can download and install this
107 browser from www.google.com/chrome/browser/.
- 108 • **glMatrix math library:** This is a library that implements the
109 foundational mathematical operations. You can download this
110 library from <http://glMatrix.net/>. You will integrate this library
111 into your game engine in Chapter 3, so more details will be provided
112 there.

113 Notice that there are no specific system requirements to support the JavaScript
114 programming language, HTML5, or WebGL. All these technologies are embedded in the
115 web browser runtime environment.

116 **Note** As mentioned, we chose the VS Code-based development environment
117 because we found it to be the most convenient. There are many other alternatives
118 that are also free, including and not limited to NetBeans, IntelliJ IDEA, Eclipse, and
119 Sublime.

Downloading and Installing JavaScript Syntax Checker	120
We have found ESLint to be an effective tool in detecting potential JavaScript source code errors. You can integrate ESLint into VS Code with the following steps:	121
• Go to https://marketplace.visualstudio.com/items?itemName=dbaeumer.vscode-eslint and click install.	122
• You will be prompted to open VS Code and may need to click install again within the application.	125
The following are some useful references for working with ESLint:	127
• For instructions on how to work with ESLint, see https://eslint.org/docs/user-guide/ .	128
• For details on how ESLint works, see https://eslint.org/docs/developer-guide/ .	130
Downloading and Installing LiveServer	132
The LiveServer extension to the VS Code is required to run your game engine. It launches a web server locally on your computer through VS Code to host developed games. Much like ESLint, you can install LiveServer with the following steps:	133
• Go to https://marketplace.visualstudio.com/items?itemName=ritwickdey.LiveServer and click install.	136
• You will be prompted to open VS Code and may need to click install again within the application.	138
Working in the VS Code Development Environment	140
The VS Code IDE is easy to work with, and the projects in this book require only the editor. Relevant source code files organized under a parent folder are interpreted by VS Code as a project. To open a project, select File ▶ Open Folder and navigate and select the parent folder that contains the source code files of the project. Once a project is open, you need to become familiar with the basic windows of VS Code, as illustrated in Figure 1-1.	141
	142
	143
	144
	145
	146

- **Explorer window:** This window displays the source code files of the project. If you accidentally close this window, you can recall it by selecting View ➤ Explorer.
- **Editor window:** This window displays and allows you to edit the source code of your project. You can select the source code file to work with by clicking once the corresponding file name in the Explorer window.
- **Output window:** This window is not used in our projects; feel free to close it by clicking the “x” icon on the top right of the window.

this figure will be printed in b/w

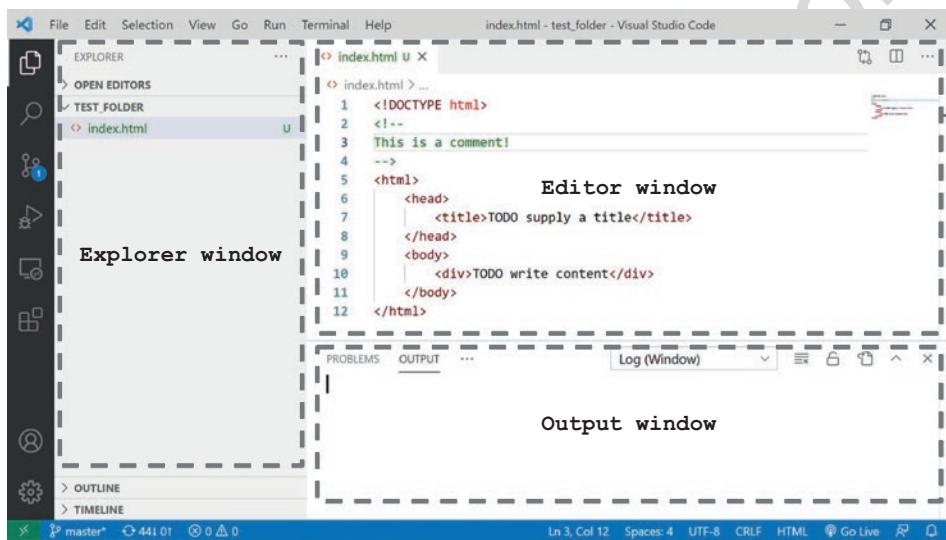


Figure 1-1. The VS Code IDE

Creating an HTML5 Project in VS Code

- You are now ready to create your first HTML5 project:
- Using File Explorer, create a directory in the location where you would like to keep your projects. This directory will contain all source code files related to your projects. In VS Code, select File ➤ Open Folder and navigate to the directory you created.

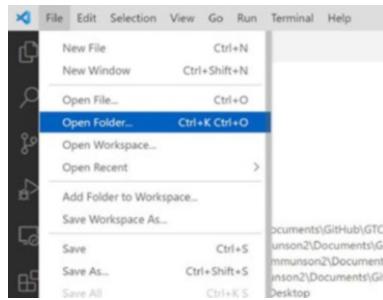


Figure 1-2. Opening a project folder

- VS Code will open the project folder. Your IDE should look similar to Figure 1-3; notice that the Explorer window is empty when your project folder is empty. 162
- 163
- 164

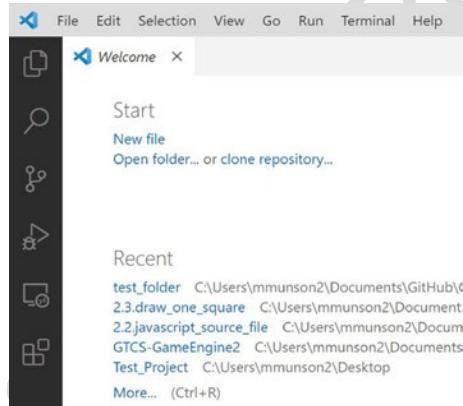


Figure 1-3. An empty VS Code project

- You can now create your first HTML file, `index.html`. Select File ➤ New File and name the file `index.html`. This will serve as the home or landing page when your application is launched. 165
- 166
- 167

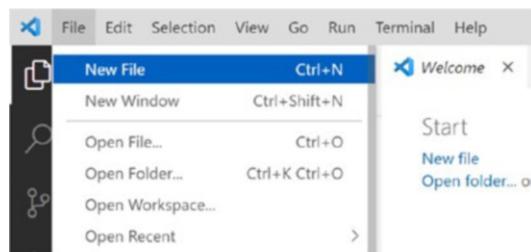


Figure 1-4. Creating the index.html file

- 168 • In the Editor window, enter the following text into your index.html:

```
169 <!DOCTYPE html>
170 <!--
171 This is a comment!
172 -->
173 <html>
174   <head>
175     <title>TODO supply a title</title>
176   </head>
177   <body>
178     <div>TODO write content</div>
179   </body>
180 </html>
```

181 The first line declares the file to be an HTML file. The block that follows within the
182 <!-- and --> tags is a comment block. The complementary <html></html> tags contain
183 all the HTML code. In this case, the template defines the head and body sections. The
184 head sets the title of the web page, and the body is where all the content for the web page
185 will be located.

186 As illustrated in Figure 1-5, you can run this project by clicking the “Go Live” button
187 in the bottom-right corner of your VS Code or by pressing Alt+L Alt+O. There is a chance
188 that right after you entered the previous HTML code for the first time, the “Go Live”
189 button may not appear. In this case, simply right-mouse-click the index.html file in the
190 Explorer window, and click “Open with Live Server” menu item to launch the web page.
191 After the first time, the “Go Live” button will appear in the lower-right region of the IDE,
192 as illustrated in Figure 1-5.



this figure will be printed in b&w

Figure 1-5. Click the Go Live button to run a project

Note To run a project, the `index.html` file of that project must be opened in the editor when the “Go Live” button is clicked or when the Alt+L Alt+O keys are typed. This will become important in the subsequent chapters when there are other JavaScript source code files in the project.

193
194
195
196

Figure 1-6 shows an example of what the default project looks like when you run it. Notice that after the project begins to run, the “Go Live” button updates its label to show “Port:5500.” You can click this button again to disconnect the IDE from the web page to observe the “Go Live” label again. Clicking the button one more time will rerun the project.

197
198
199
200
201

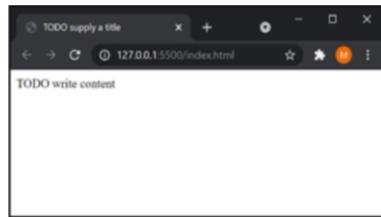


Figure 1-6. Running the simple HTML5 project

202 To stop the program, simply close the web page. You have successfully run your
203 first HTML5 project. Through the development of this very simple project, you have
204 familiarized yourself with the IDE environment.

205 **Note** For debugging, we recommend the Chrome Developer tools. These tools
206 can be accessed by typing **Ctrl+Shift+I** (or the F12 key) in the browser window
207 when your project is running. To find out more about these tools, please refer to
208 <https://developer.chrome.com/docs/devtools/>.

209 How to Use This Book

210 This book guides you through the development of a game engine by building projects
211 similar to the one you have just experienced. Each chapter covers an essential
212 component of a typical game engine, and the sections in each chapter describe the
213 important concepts and implementation projects that construct the corresponding
214 component. Throughout the text, the project from each section builds upon the results
215 from the projects that precede it. While this makes it a little challenging to skip around
216 in the book, it will give you practical experience and a solid understanding of how
217 the different concepts relate. In addition, rather than always working with new and
218 minimalistic projects, you gain experience with building larger and more interesting
219 projects while integrating new functionality into your expanding game engine.

220 The projects start with demonstrating simple concepts, such as drawing a simple
221 square, but evolve quickly into presenting more complex concepts, such as working
222 with user-defined coordinate systems and implementing pixel-accurate collision
223 detection. Initially, as you have experienced in building the first HTML5 application,
224 you will be guided with detailed steps and complete source code listings. As you become

familiar with the development environment and the technologies, the guides and source code listings accompanying each project will shift to highlight on the important implementation details. Eventually, as the complexity of the projects increases, the discussion will focus only on the vital and relevant issues, while straightforward source code changes will not be mentioned.

The final code base, which you will have developed incrementally over the course of the book, is a complete and practical game engine; it's a great platform on which you can begin building your own 2D games. This is exactly what the last chapter of the book does, leading you from the conceptualization to design to implementation of a casual 2D game.

There are several ways for you to follow along with this book. The most obvious is to enter the code into your project as you follow each step in the book. From a learning perspective, this is the most effective way to absorb the information presented; however, we understand that it may not be the most realistic because of the amount of code or debugging this approach may require. Alternatively, we recommend that you run and examine the source code of the completed project when you begin a new section. Doing so lets you preview the current section's project, gives you a clear idea of the end goal, and lets you see what the project is trying to achieve. You may also find the completed project code useful when you have problems while building the code yourself, because during difficult debugging situations, you can compare your code with the code of the completed project.

Note We have found the WinMerge program (<http://winmerge.org/>) to be an excellent tool for comparing source code files and folders. Mac users can check out the FileMerge utility for a similar purpose.

Finally, after completing a project, we recommend that you compare the behavior of your implementation with the completed implementation provided. By doing so, you can observe whether your code is behaving as expected.

252 How Do You Make a Great Video Game?

253 While the focus of this book is on the design and implementation of a game engine, it is
254 important to appreciate how different components can contribute to the creation of a
255 fun and engaging video game. Beginning in Chapter 4, a “Game Design Consideration”
256 section is included at the end of each chapter to relate the functionality of the engine
257 component to elements of game design. This section presents the framework for these
258 discussions.

259 It’s a complex question, and there’s no exact formula for making a video game that
260 people will love to play, just as there’s no exact formula for making a movie that people
261 will love to watch. We’ve all seen big-budget movies that look great and feature top
262 acting, writing, and directing talent but that bomb at the box office, and we’ve all seen
263 big-budget games from major studios that fail to capture the imaginations of players. By
264 the same token, movies by unknown directors can grab the world’s attention, and games
265 from small, unknown studios can take the market by storm.

266 While no explicit instructions exist for making a great game, a number of elements
267 work together in harmony to create a final experience greater than the sum of its parts,
268 and all game designers must successfully address each of them in order to produce
269 something worth playing. The elements include the following:

- 270 • **Technical design:** This includes all game code and the game
271 platform and is generally not directly exposed to players; rather,
272 it forms the foundation and scaffolding for all aspects of the game
273 experience. This book is primarily focused on issues related to the
274 technical design of games, including specific tasks such as the lines of
275 code required to draw elements on the screen and more architectural
276 considerations such as determining the strategy for how and when
277 to load assets into memory. Technical design issues impact the
278 player experience in many ways (e.g., the number of times a player
279 experiences “loading” delays during play or how many frames per
280 second the game displays), but the technical design is typically
281 invisible to players because it runs under what’s referred to as the
282 presentation layer or all of the audiovisual and/or haptic feedback
283 the player encounters during play.

- **Game mechanic(s):** The game mechanic is an abstract description of what can be referred to as the foundation of play for a given game experience. Types of game mechanics include puzzles, dexterity challenges such as jumping or aiming, timed events, combat encounters, and the like. The game mechanic is a framework; specific puzzles, encounters, and game interactions are implementations of the framework. A real-time strategy (RTS) game might include a resource-gathering mechanic, for example, where the mechanic might be described as “Players are required to gather specific types of resources and combine them to build units which they can use in combat.” The specific implementation of that mechanic (how players locate and extract the resources in the game, how they transport them from one place to another, and the rules for combining resources to produce units) is an aspect of systems design, level design, and the interaction model (described later in this section). 284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
- **Systems design:** The internal rules and logical relationships that provide structured challenge to the core game mechanic are referred to as the game’s systems design. Using the previous RTS example, a game might require players to gather a certain amount of metal ore and combine it with a certain amount of wood to make a game object; the specific rules for how many of each resource is required to make the objects and the unique process for creating the objects (e.g., objects can be produced only in certain structures on the player’s base and take x number of minutes to appear after the player starts the process) are aspects of systems design. Casual games may have basic systems designs. A simple puzzle game like *Pull the Pin* from Popcore Games, for example, is a game with few systems and low complexity, while major genres like RTS games may have deeply complex and interrelated systems designs created and balanced by entire teams of designers. Game systems designs are often where the most hidden complexity of game design exists; as designers go through the exercise of defining all variables that contribute to an implementation of a game mechanic, it’s easy to become lost in a sea of complexity and balance dependencies. Systems that appear fairly simple to players may require many components working together 299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318

319 and balanced perfectly against each other, and underestimating
320 system complexity is perhaps one of the biggest pitfalls encountered
321 by new (and veteran!) game designers. Until you know what you're
322 getting into, always assume the systems you create will prove to be
323 considerably more complex than you anticipate.

- 324 • **Level design:** A game's level design reflects the specific ways each of
325 the other eight elements combines within the context of individual
326 "chunks" of gameplay, where players must complete a certain chunk
327 of objectives before continuing to the next section (some games may
328 have only one level, while others will have dozens). Level designs
329 within a single game can all be variations of the same core mechanic
330 and systems design (games like *Tetris* and *Bejeweled* are examples of
331 games with many levels all focusing on the same mechanic), while
332 other games will mix and match mechanics and systems designs for
333 variety among levels. Most games feature one primary mechanic and
334 a game-spanning approach to systems design and will add minor
335 variations between levels to keep things feeling fresh (changing
336 environments, changing difficulty, adding time limits, increasing
337 complexity, and the like), although occasionally games will introduce
338 new levels that rely on completely separate mechanics and systems
339 to surprise players and hold their interest. Great level design in
340 games is a balance between creating "chunks" of play that showcase
341 the mechanic and systems design and changing enough between
342 these chunks to keep things interesting for players as they progress
343 through the game (but not changing so much between chunks that
344 the gameplay feels disjointed and disconnected).
- 345 • **Interaction model:** The interaction model is the combination of
346 keys, buttons, controller sticks, touch gestures, and so on, used to
347 interact with the game to accomplish tasks and the graphical user
348 interfaces that support those interactions within the game world.
349 Some game theorists break the game's user interface (UI) design
350 into a separate category (game UI includes things such as menu
351 designs, item inventories, heads-up displays [HUDs]), but the
352 interaction model is deeply connected to UI design, and it's a good
353 practice to think of these two elements as inseparable. In the case

of the RTS game referenced earlier, the interaction model includes the actions required to select objects in the game, to move those objects, to open menus and manage inventories, to save progress, to initiate combat, and to queue build tasks. The interaction model is completely independent of the mechanic and systems design and is concerned only with the physical actions the player must take to initiate behaviors (e.g., click mouse button, press key, move stick, scroll wheel); the UI is the audiovisual or haptic feedback connected to those actions (onscreen buttons, menus, statuses, audio cues, vibrations, and the like).	354
• Game setting: Are you on an alien planet? In a fantasy world? In an abstract environment? The game setting is a critical part of the game experience and, in partnership with the audiovisual design, turns what would otherwise be a disconnected set of basic interactions into an engaging experience with context. Game settings need not be elaborate to be effective; the perennially popular puzzle game <i>Tetris</i> has a rather simple setting with no real narrative wrapper, but the combination of abstract setting, audiovisual design, and level design is uniquely well matched and contributes significantly to the millions of hours players invest in the experience year after year.	364
• Visual design: Video games exist in a largely visual medium, so it's not surprising that companies frequently spend as much or more on the visual design of their games as they spend on the technical execution of the code. Large games are aggregations of thousands of visual assets, including environments, characters, objects, animations, and cinematics; even small casual games generally ship with hundreds or thousands of individual visual elements. Each object a player interacts with in the game must be a unique asset, and if that asset includes more complex animation than just moving it from one location on the screen to another or changing the scale or opacity, the object most likely will need to be animated by an artist. Game graphics need not be photorealistic or stylistically elaborate to be visually excellent or to effectively represent the setting (many games intentionally utilize a simplistic visual style), but the best games consider art direction and visual style to be core to the player	374

389 experience, and visual choices will be intentional and well matched
390 to the game setting and mechanic.

- 391 • **Audio design:** This includes music and sound effects, ambient
392 background sounds, and all sounds connected to player actions
393 (select/use/swap item, open inventory, invoke menu, and the like).
394 Audio design functions hand in hand with visual design to convey
395 and reinforce game setting, and many new designers significantly
396 underestimate the impact of sound to immerse players into game
397 worlds. Imagine *Star Wars*, for example, without the music, the light
398 saber sound effect, Darth Vader's breathing, or R2D2's characteristic
399 beeps; the audio effects and musical score are as fundamental to the
400 experience as the visuals.
- 401 • **Meta-game:** The meta-game centers on how individual objectives
402 come together to propel players through the game experience (often
403 via scoring, unlocking individual levels in sequence, playing through
404 a narrative, and the like). In many modern games, the meta-game
405 is the narrative arc or story; players often don't receive a "score" per
406 se but rather reveal a linear or semi-linear story as they progress
407 through game levels, driving forward to complete the story. Other
408 games (especially social and competitive games) involve players
409 "leveling up" their characters, which can happen as a result of
410 playing through a game-spanning narrative experience or by simply
411 venturing into the game world and undertaking individual challenges
412 that grant experience points to characters. Other games, of course,
413 continue focusing on scoring points or winning rounds against other
414 players.

415 The magic of video games typically arises from the interplay between these nine
416 elements, and the most successful games finely balance each as part of a unified
417 vision to ensure a harmonious experience; this balance will always be unique to each
418 individual effort and is found in games ranging from Nintendo's *Animal Crossing* to
419 Rockstar's *Red Dead Redemption 2*. The core game mechanic in many successful games
420 is often a variation on one or more fairly simple, common themes (*Pull the Pin*, e.g., is a
421 game based entirely on pulling virtual pins from a container to release colored balls), but
422 the visual design, narrative context, audio effects, interactions, and progression system

work together with the game mechanic to create a unique experience that's considerably more engaging than the sum of its individual parts, making players want to return to it again and again. Great games range from the simple to the complex, but they all feature an elegant balance of supporting design elements. 423
424
425
426

References 427

The examples in this book are created with the assumptions that you understand data encapsulation, inheritance, and basic data structures, such as linked lists and dictionaries, and are comfortable working with the fundamentals of algebra and geometry, particularly linear equations and coordinate systems. Many examples in this book apply and implement concepts in computer graphics and linear algebra. These concepts warrant much more in-depth examinations. Interested readers can learn more about these topics in other books. 428
429
430
431
432
433
434

- Computer graphics:
 - Marschner and Shirley. *Fundamentals of Computer Graphics*, 4th edition. CRC Press, 2016. 436
437
 - Angle and Shreiner. *Interactive Computer Graphics: A Top Down Approach with WebGL*, 7th edition. Pearson Education, 2014. 438
439
- Linear algebra:
 - Sung and Smith. *Basic Math for Game Development with Unity 3D: A Beginner's Guide to Mathematical Foundations*. Apress, 2019. 441
442
443
 - Johnson, Riess, and Arnold. *Introduction to Linear Algebra*, 5th edition. Addison-Wesley, 2002. 444
445
 - Anton and Rorres. *Elementary Linear Algebra: Applications Version*, 11th edition. Wiley, 2013. 446
447

448 Technologies

449 The following list offers links for obtaining additional information on technologies used
450 in this book:

- 451 • **JavaScript:** www.w3schools.com/js
- 452 • **HTML5:** www.w3schools.com/html/html5_intro.asp
- 453 • **WebGL:** www.khronos.org/webgl
- 454 • **OpenGL:** www.opengl.org
- 455 • **Visual Studio Code:** <https://code.visualstudio.com/>
- 456 • **Chrome:** www.google.com/chrome
- 457 • **glMatrix:** <http://glMatrix.net>
- 458 • **ESLint:** www.eslint.org

Author Queries

Chapter No.: 1 0005244094

Queries	Details Required	Author's Response
AU1	Please check if all occurrences of “Alt+L Alt+O” should be changed to “Alt+L and then Alt+O”.	
AU2	Please check if “right-mouse-click” should be changed to “right-click.”	

Uncorrected Proof

CHAPTER 2

1

Working with HTML5 and WebGL

2

3

After completing this chapter, you will be able to

- Create a new JavaScript source code file for your simple game engine
- Draw a simple constant color square with WebGL
- Define JavaScript modules and classes to encapsulate and implement core game engine functionality
- Appreciate the importance of abstraction and the organization of your source code structure to support growth in complexity

4

5

6

7

8

9

10

Introduction

11

Drawing is one of the most essential functionalities common to all video games. A game engine should offer a flexible and programmer-friendly interface to its drawing system. In this way, when building a game, the designers and developers can focus on the important aspects of the game itself, such as mechanics, logic, and aesthetics.

12

13

14

15

WebGL is a modern JavaScript graphical application programming interface (API) designed for web browser-based applications that offers quality and efficiency via direct access to the graphics hardware. For these reasons, WebGL serves as an excellent base to support drawing in a game engine, especially for video games that are designed to be played across the Internet.

16

17

18

19

This chapter examines the fundamentals of drawing with WebGL, designs abstractions to encapsulate irrelevant details to facilitate programming, and builds the foundational infrastructure to organize a complex source code system to support future expansion.

20

21

22

23

24

25 **Note** The game engine you will develop in this book is based on the latest
26 version of WebGL specification: version 2.0. For brevity, the term WebGL will be
27 used to refer to this API.

28 **Canvas for Drawing**

29 To draw, you must first define and dedicate an area within the web page. You can achieve this
30 easily by using the HTML canvas element to define an area for WebGL drawing. The canvas
31 element is a container for drawing that you can access and manipulate with JavaScript.

32 **The HTML5 Canvas Project**

33 This project demonstrates how to create and clear a canvas element on a web
34 page. Figure 2-1 shows an example of running this project, which is defined in the
35 chapter2/2.1.html5_canvas folder.

this figure will be printed in b/w



The above is WebGL draw area!

Figure 2-1. Running the HTML5 Canvas project

The goals of the project are as follows:	36
• To learn how to set up the HTML canvas element	37
• To learn how to retrieve the canvas element from an HTML document for use in JavaScript	38
	39
• To learn how to create a reference context to WebGL from the retrieved canvas element and manipulate the canvas through the WebGL context	40
	41
	42

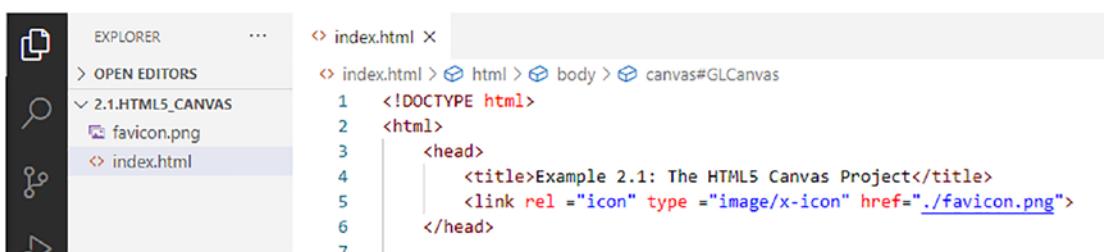
Creating and Clearing the HTML Canvas

In this first project, you will create an empty HTML5 canvas and clear the canvas to a specific color with WebGL:

1. Create a new project by creating a new folder named html5_canvas in your chosen directory and copying and pasting the index.html file you created in the previous project in Chapter 1.

Note From this point on, when asked to create a new project, you should follow the process described previously. That is, create a new folder with the project's name and copy/paste the previous project's files. In this way, your new projects can expand upon your old ones while retaining the original functionality.

2. Open the index.html file in the editor by opening the html5_canvas folder, expanding it if needed and clicking the index.html file, as illustrated in Figure 2-2.



this figure will be printed in b/w

Figure 2-2. Editing the index.html file in your project

- 56 3. Create the HTML canvas for drawing by adding the following lines
57 in the index.html file within the body element:

58 <canvas id="GLCanvas" width="640" height="480">
59 Your browser does not support the HTML5 canvas.
60 </canvas>

61 The code defines a canvas element named GLCanvas with the specified width and
62 height attributes. As you will experience later, you will retrieve the reference to the
63 GLCanvas to draw into this area. The text inside the element will be displayed if your
64 browser does not support drawing with WebGL.

65 **Note** The lines between the <body> and </body> tags are referred to as
66 “within the body element.” For the rest of this book, “within the AnyTag element”
67 will be used to refer to any line between the beginning (<AnyTag>) and end
68 (</AnyTag>) of the element.

- 69 4. Create a script element for the inclusion of JavaScript
70 programming code, once again within the body element:

71 <script type="text/javascript">
72 // JavaScript code goes here.
73 </script>

74 This takes care of the HTML portion of this project. You will now write JavaScript
75 code for the remainder of the example:

- 76 5. Retrieve a reference to the GLCanvas in JavaScript code by adding
77 the following line within the script element:

78 "use strict";
79 let canvas = document.getElementById("GLCanvas");

80 **Note** The let JavaScript keyword defines variables.

The first line, “use strict” is a JavaScript directive indicating that the code should
be executed in “strict mode”, where the use of undeclared variables is a runtime error.
The second line creates a new variable named canvas and references the variable to the
GLCanvas drawing area.

Note All local variable names begin with a lowercase letter, as in canvas.

6. Retrieve and bind a reference to the WebGL context to the drawing
area by adding the following code:

```
let gl = canvas.getContext("webgl2") ||  
    canvas.getContext("experimental-webgl2");
```

As the code indicates, the retrieved reference to the WebGL version 2 context is
stored in the local variable named gl. From this variable, you have access to all the
functionality of WebGL 2.0. Once again, in the rest of this book, the term WebGL will be
used to refer to the WebGL version 2.0 API.

7. Clear the canvas drawing area to your favorite color through
WebGL by adding the following:

```
if (gl !== null) {  
    gl.clearColor(0.0, 0.8, 0.0, 1.0);  
    gl.clear(gl.COLOR_BUFFER_BIT);  
}
```

This code checks to ensure that the WebGL context is properly retrieved, sets the
clear color, and clears the drawing area. Note that the clearing color is given in RGBA
format, with floating-point values ranging from 0.0 to 1.0. The fourth number in the
RGBA format is the alpha channel. You will learn more about the alpha channel in later
chapters. For now, always assign 1.0 to the alpha channel. The specified color, (0.0,
0.8, 0.0, 1.0), has zero values for the red and blue channels and a 0.8, or 80 percent,
intensity on the green channel. For this reason, the canvas area is cleared to a light green
color.

- 108 8. Add a simple `write` command to the document to identify the
109 canvas by inserting the following line:

110 `document.write("
The above is WebGL draw area!");`

111 You can refer to the final source code in the `index.html` file in the
112 `chapter2/2.1.html5_canvas` project. Run the project, and you should see a light green
113 area on your browser window as shown in Figure 2-1. This is the 640×480 canvas drawing
114 area you defined.

115 You can try changing the cleared color to white by setting the RGBA of
116 `gl.clearColor()` to 1 or to black by setting the color to 0 and leaving the alpha value 1.
117 Notice that if you set the alpha channel to 0, the canvas color will disappear. This is
118 because a 0 value in the alpha channel represents complete transparency, and thus,
119 you will “see through” the canvas and observe the background color of the web page.
120 You can also try altering the resolution of the canvas by changing the 640×480 values to
121 any number you fancy. Notice that these two numbers refer to the pixel counts and thus
122 must always be integers.

123 Separating HTML and JavaScript

124 In the previous project, you created an HTML canvas element and cleared the area
125 defined by the canvas using WebGL. Notice that all the functionality is clustered in the
126 `index.html` file. As the project complexity increases, this clustering of functionality
127 can quickly become unmanageable and negatively impact the programmability of
128 your system. For this reason, throughout the development process in this book, after a
129 concept is introduced, efforts will be spent on separating the associated source code into
130 either well-defined source code files or classes in an object-oriented programming style.
131 To begin this process, the HTML and JavaScript source code from the previous project
132 will be separated into different source code files.

133 The JavaScript Source File Project

134 This project demonstrates how to logically separate the source code into appropriate
135 files. You can accomplish this by creating a separate JavaScript source code file named
136 `core.js` which implements the corresponding functionality in the `index.html` file.
137 The web page will load the JavaScript source code as instructed by the code in

the `index.html` file. As illustrated in Figure 2-3, this project looks identical as the previous project when running. The source code of this project is located in the `chapter2/2.2.javascript_source_file` folder.

138
139
140



this figure will be printed in b/w

The above is WebGL draw area!

Figure 2-3. Running the JavaScript Source File project

The goals of the project are as follows:

- To learn how to separate source code into different files
- To organize your code in a logical structure

141
142
143

Separate JavaScript Source Code File

144
145
146
147

This section details how to create and edit a new JavaScript source code file. You should familiarize yourself with this process because you'll create numerous source code files throughout this book.

148
149
150
151
152
153
154
155
156

1. Create a new HTML5 project titled javascript_source_file.
Recall that a new project is created by creating a folder with the appropriate name, copying files from the previous project, and editing the <Title> element of the index.html to reflect the new project.
2. Create a new folder named src inside the project folder by clicking the new folder icon while hovering over the project folder, as illustrated in Figure 2-4. This folder will contain all of your source code.

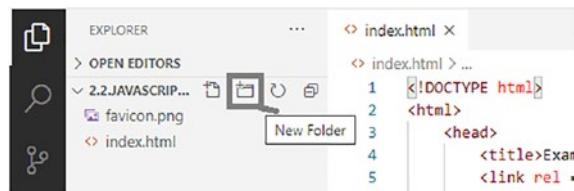


Figure 2-4. Creating a new source code folder

157
158
159

3. Create a new source code file within the src folder by right-clicking the src folder, as illustrated in Figure 2-5. Name the new source file core.js.

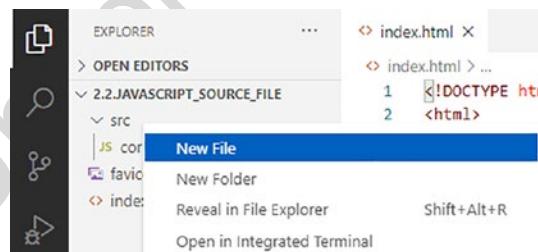


Figure 2-5. Adding a new JavaScript source code file

Note In VS Code, you can create/copy/rename folders and files by using the right-click menus in the Explorer window.

4. Open the new core.js source file for editing. 160
5. Define a variable for referencing the WebGL context, and add a function which allows you to access the variable: 161

```
"use strict"; 163  
let mGL = null; 164  
function getGL() { return mGL; } 165
```

Note Variables that are accessible throughout a file, or a module, have names that begin with lowercase "m," as in mGL. 166

6. Define the initWebGL() function to retrieve GLCanvas by passing in the proper canvas id as a parameter, bind the drawing area to the WebGL context, store the results in the defined mGL variable, and clear the drawing area: 168

```
function initWebGL(htmlCanvasID) { 172  
    let canvas = document.getElementById(htmlCanvasID); 173  
  
    mGL = canvas.getContext("webgl2") || 174  
        canvas.getContext("experimental-webgl2"); 175  
  
    if (mGL === null) { 176  
        document.write("<br><b>WebGL 2 is not supported!</b>"); 177  
        return; 178  
    } 179  
    mGL.clearColor(0.0, 0.8, 0.0, 1.0); 180  
} 181
```

Notice that this function is similar to the JavaScript source code you typed in the previous project. This is because all you are doing differently, in this case, is separating JavaScript source code from HTML code. 182

Note All function names begin with a lowercase letter, as in initWebGL(). 183

- 186 7. Define the `clearCanvas()` function to invoke the WebGL context
187 to clear the canvas drawing area:

```
188 function clearCanvas() {  
189     mGL.clear(mGL.COLOR_BUFFER_BIT);  
190 }
```

- 191 8. Define a function to carry out the initialization and clearing of the
192 canvas area after the web browser has completed the loading of
193 the `index.html` file:

```
194 window.onload = function() {  
195     initWebGL("GLCanvas");  
196     clearCanvas();  
197 }
```

198 Load and Run JavaScript Source Code from `index.html`

199 With all the JavaScript functionality defined in the `core.js` file, you now need to load
200 this file to operate on your web page through the `index.html` file:

- 201 1. Open the `index.html` file for editing.
202 2. Create the HTML canvas, `GLCanvas`, as in the previous project.
203 3. Load the `core.js` source code by including the following code
204 within the head element:

```
205 <script type="module" src="./src/core.js"></script>
```

206 With this code, the `core.js` file will be loaded as part of the `index.html` defined web
207 page. Recall that you have defined a function for `window.onload` and that function will
208 be invoked when the loading of `index.html` is completed.

209 You can refer to the final source code in the `core.js` and `index.html` files in the
210 `chapter2/2.2.javascript_source_file` project folder. Although the output from this
211 project is identical to that from the previous project, the organization of your code will
212 allow you to expand, debug, and understand the game engine as you continue to add
213 new functionality.

Note Recall that to run a project, you click the “Go Live” button on the lower right of the VS Code window, or type Alt+L Alt+O keys, while the associated `index.html` file is opened in the Editor window. In this case, the project will not run if you click the “Go Live” button while the `core.js` file is opened in the Editor window.

214
215
216
217

Observations

218
219
220
221
222
223
224
225

Examine your `index.html` file closely and compare its content to the same file from the previous project. You will notice that the `index.html` file from the previous project contains two types of information (HTML and JavaScript code) and that the same file from this project contains only the former, with all JavaScript code being extracted to `core.js`. This clean separation of information allows for easy understanding of the source code and improves support for more complex systems. From this point on, all JavaScript source code will be added to separate source code files.

Elementary Drawing with WebGL

226
227
228
229
230
231
232
233

In general, drawing involves geometric data and the instructions for processing the data. In the case of WebGL, the instructions for processing the data are specified in the OpenGL Shading Language (GLSL) and are referred to as shaders. In order to draw with WebGL, programmers must define the geometric data and GLSL shaders in the CPU and load both to the drawing hardware, or the graphics processing unit (GPU). This process involves a significant number of WebGL function calls. This section presents the WebGL drawing steps in detail.

It is important to focus on learning these basic steps and avoid being distracted by the less important WebGL configuration nuances such that you can continue to learn the overall concepts involved when building your game engine.

234
235
236

In the following project, you will learn about drawing with WebGL by focusing on the most elementary operations. This includes the loading of the simple geometry of a square from the CPU to the GPU, the creation of a constant color shader, and the basic instructions for drawing a simple square with two triangles.

237
238
239
240

The Draw One Square Project

This project leads you through the steps required to draw a single square on the canvas. Figure 2-6 shows an example of running this project, which is defined in the chapter2/2.3.draw_one_square folder.

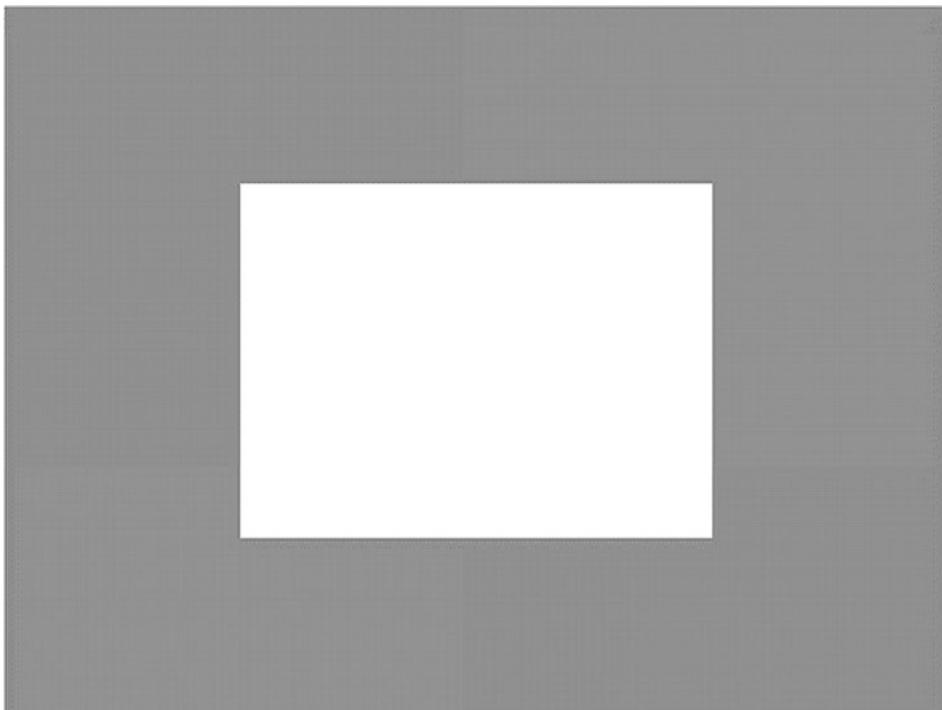


Figure 2-6. Running the Draw One Square project

The goals of the project are as follows:

- To understand how to load geometric data to the GPU
- To learn about simple GLSL shaders for drawing with WebGL
- To learn how to compile and load shaders to the GPU
- To understand the steps required to draw with WebGL
- To demonstrate the implementation of a singleton-like JavaScript module based on simple source code files

Set Up and Load the Primitive Geometry Data

To draw efficiently with WebGL, the data associated with the geometry to be drawn, such as the vertex positions of a square, should be stored in the GPU hardware. In the following steps, you will create a contiguous buffer in the GPU, load the vertex positions of a unit square into the buffer, and store the reference to the GPU buffer in a variable. Learning from the previous project, the corresponding JavaScript code will be stored in a new source code file, `vertex_buffer.js`.

Note A unit square is a 1×1 square centered at the origin.

1. Create a new JavaScript source file in the `src` folder and name it `vertex_buffer.js`. 260
261
2. Import all the exported functionality from the `core.js` file as `core` 262
with the JavaScript `import` statement: 263

```
"use strict";
import * as core from "./core.js";
```

Note With the JavaScript `import` and, soon to be encountered, `export` statements, features and functionalities defined in a file can be conveniently encapsulated and accessed. In this case, the functionality exported from `core.js` is imported in `vertex_buffer.js` and accessible via the module identifier, `core`. For example, as you will see, in this project, `core.js` defines and exports a `getGL()` function. With the given `import` statement, this function can be accessed as `core.getGL()` in the `vertex_buffer.js` file.

3. Declare the variable `mGLVertexBuffer` to store the reference to 273
the WebGL buffer location. Remember to define a function for 274
accessing this variable. 275

```
let mGLVertexBuffer = null;
function get() { return mGLVertexBuffer; }
```

CHAPTER 2 WORKING WITH HTML5 AND WEBGL

- 278 4. Define the variable `mVerticesOfSquare` and initialize it with
279 vertices of a unit square:

```
280 let mVerticesOfSquare = [  
281     0.5, 0.5, 0.0,  
282     -0.5, 0.5, 0.0,  
283     0.5, -0.5, 0.0,  
284     -0.5, -0.5, 0.0  
285 ];
```

286 In the code shown, each row of three numbers is the x-, y-, and z-coordinate position
287 of a vertex. Notice that the z dimension is set to 0.0 because you are building a 2D game
288 engine. Also notice that 0.5 is being used so that we define a square in 2D space which
289 has sides equal to 1 and centered at the origin or a unit square.

- 290 5. Define the `init()` function to allocate a buffer in the GPU via
291 the `gl` context, and load the vertices to the allocated buffer in
292 the GPU:

```
293 function init() {  
294     let gl = core.getGL();  
295     // Step A: Create a buffer on the gl context for our vertex positions  
296     mGLVertexBuffer = gl.createBuffer();  
297     // Step B: Activate vertexBuffer  
298     gl.bindBuffer(gl.ARRAY_BUFFER, mGLVertexBuffer);  
299     // Step C: Loads mVerticesOfSquare into the vertexBuffer  
300     gl.bufferData(gl.ARRAY_BUFFER,  
301         new Float32Array(mVerticesOfSquare), gl.STATIC_DRAW);  
302 }
```

303 This code first gets access to the WebGL drawing context through the `core.getGL()`
304 function. After which, Step A creates a buffer on the GPU for storing the vertex
305 positions of the square and stores the reference to the GPU buffer in the variable
306 `mGLVertexBuffer`. Step B activates the newly created buffer, and step C loads the vertex
307 position of the square into the activated buffer on the GPU. The keyword `STATIC_DRAW`
308 informs the drawing hardware that the content of this buffer will not be changed.

Tip Remember that the mGL variable accessed through the getGL() function is
defined in the core.js file and initialized by the initWebGL() function. You will
define an export statement in the core.js file to provide access to this function
in the coming steps.

6. Provide access to the init() and get() functions to the rest of
your engine by exporting them with the following code:

```
export {init, get}
```

With the functionality of loading vertex positions defined, you are now ready to
define and load the GLSL shaders.

Set Up the GLSL Shaders

The term shader refers to programs, or a collection of instructions, that run on the
GPU. In the context of the game engine, shaders must always be defined in pairs
consisting of a vertex shader and a corresponding fragment shader. The GPU will execute
the vertex shader once per primitive vertex and the fragment shader once per pixel
covered by the primitive. For example, you can define a square with four vertices and
display this square to cover a 100×100 pixel area. To draw this square, WebGL will invoke
the vertex shader 4 times (once for each vertex) and execute the fragment shader 10,000
times (once for each of the 100×100 pixels)!

In the case of WebGL, both the vertex and fragment shaders are implemented in
the OpenGL Shading Language (GLSL). GLSL is a language with syntax that is similar to
the C programming language and designed specifically for processing and displaying
graphical primitives. You will learn sufficient GLSL to support the drawing for the game
engine when required.

In the following steps, you will load into GPU memory the source code for both
vertex and fragment shaders, compile and link them into a single shader program, and
load the linked program into the GPU memory for drawing. In this project, the shader
source code is defined in the index.html file, while the loading, compiling, and linking
of the shaders are defined in the shader_support.js source file.

337 **Note** The WebGL context can be considered as an abstraction of the GPU
338 hardware. To facilitate readability, the two terms WebGL and GPU are sometimes
339 used interchangeably.

340 **Define the Vertex and Fragment Shaders**

341 GLSL shaders are simply programs consisting of GLSL instructions:

- 342 1. Define the vertex shader by opening the `index.html` file, and
343 within the head element, add the following code:

```
344 <script type="x-shader/x-vertex" id="VertexShader">  
345     // this is the vertex shader  
346     attribute vec3 aVertexPosition; // Expects one vertex position  
347         // naming convention, attributes always begin with "a"  
348     void main(void) {  
349         // Convert the vec3 into vec4 for scan conversion and  
350         // assign to gl_Position to pass vertex to the fragment shader  
351         gl_Position = vec4(aVertexPosition, 1.0);  
352     }  
353     // End of vertex shader  
354 </script>
```

355 **Note** Shader attribute variables have names that begin with a lowercase “a,” as
356 in `aVertexPosition`.

357 The `script` element type is set to `x-shader/x-vertex` because that is a common
358 convention for shaders. As you will see, the `id` field with the value `VertexShader` allows
359 you to identify and load this vertex shader into memory.

360 The GLSL `attribute` keyword identifies per-vertex data that will be passed to
361 the vertex shader in the GPU. In this case, the `aVertexPosition` attribute is of data
362 type `vec3` or an array of three floating-point numbers. As you will see in later steps,
363 `aVertexPosition` will be set to reference the vertex positions for the unit square.

The `gl_Position` is a GLSL built-in variable, specifically an array of four floating-point numbers that must contain the vertex position. In this case, the fourth position of the array will always be 1.0. The code shows the shader converting the `aVertexPosition` into a `vec4` and passing the information to WebGL.

2. Define the fragment shader in `index.html` by adding the following code within the `head` element:

```
<script type="x-shader/x-fragment" id="FragmentShader">
    // this is the fragment (or pixel) shader
    void main(void) {
        // for every pixel called (within the square) sets
        // constant color white with alpha-channel value of 1.0
        gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
    }
    // End of fragment/pixel shader
</script>
```

Note the different type and `id` fields. Recall that the fragment shader is invoked once per pixel. The variable `gl_FragColor` is the built-in variable that determines the color of the pixel. In this case, a color of `(1,1,1,1)`, or white, is returned. This means all pixels covered will be shaded to a constant white color.

With both the vertex and fragment shaders defined in the `index.html` file, you are now ready to implement the functionality to compile, link, and load the resulting shader program to the GPU.

Compile, Link, and Load the Vertex and Fragment Shaders

To maintain source code in logically separated source files, you will create shader support functionality in a new source code file, `shader_support.js`.

1. Create a new JavaScript file, `shader_support.js`.
2. Import functionality from the `core.js` and `vertex_buffer.js` files:

```
"use strict"; // Variables must be declared before used!
import * as core from "./core.js"; // access as core module
import * as vertexBuffer from "./vertex_buffer.js"; //vertexBuffer module
```

CHAPTER 2 WORKING WITH HTML5 AND WEBGL

- 394 3. Define two variables, `mCompiledShader` and `mVertexPositionRef`,
395 for referencing to the shader program and the vertex position
396 attribute in the GPU:

```
397 let mCompiledShader = null;  
398 let mVertexPositionRef = null;
```

- 399 4. Create a function to load and compile the shader you defined in
400 the `index.html`:

```
401 function loadAndCompileShader(id, shaderType) {  
402     let shaderSource = null, compiledShader = null;  
403  
404     // Step A: Get the shader source from index.html  
405     let shaderText = document.getElementById(id);  
406     shaderSource = shaderText.firstChild.textContent;  
407  
408     let gl = core.getGL();  
409     // Step B: Create shader based on type: vertex or fragment  
410     compiledShader = gl.createShader(shaderType);  
411  
412     // Step C: Compile the created shader  
413     gl.shaderSource(compiledShader, shaderSource);  
414     gl.compileShader(compiledShader);  
415  
416     // Step D: check for errors and return results (null if error)  
417     // The log info is how shader compilation errors are displayed.  
418     // This is useful for debugging the shaders.  
419     if (!gl.getShaderParameter(compiledShader, gl.COMPILE_STATUS)) {  
420         throw new Error("A shader compiling error occurred: " +  
421                         gl.getShaderInfoLog(compiledShader));  
422     }  
423  
424     return compiledShader;  
425 }
```

426 Step A of the code finds shader source code in the `index.html` file using the `id` field
427 you specified when defining the shaders, either `VertexShader` or `FragmentShader`. Step
428 B creates a specified shader (either vertex or fragment) in the GPU. Step C specifies the

source code and compiles the shader. Finally, step D checks and returns the reference to the compiled shader while throwing an error if the shader compilation is unsuccessful. 424
425

5. You are now ready to create, compile, and link a shader program 426
by defining the `init()` function: 427

```
function init(vertexShaderID, fragmentShaderID) { 428
    let gl = core.getGL(); 429

    // Step A: load and compile vertex and fragment shaders 430
    let vertexShader = loadAndCompileShader(vertexShaderID, 431
                                              gl.VERTEX_SHADER); 432
    let fragmentShader = loadAndCompileShader(fragmentShaderID, 433
                                              gl.FRAGMENT_SHADER); 434

    // Step B: Create and link the shaders into a program. 435
    mCompiledShader = gl.createProgram(); 436
    gl.attachShader(mCompiledShader, vertexShader); 437
    gl.attachShader(mCompiledShader, fragmentShader); 438
    gl.linkProgram(mCompiledShader); 439

    // Step C: check for error 440
    if (!gl.getProgramParameter(mCompiledShader, gl.LINK_STATUS)) { 441
        throw new Error("Error linking shader"); 442
        return null; 443
    } 444

    // Step D: Gets reference to aVertexPosition attribute in the shader 445
    mVertexPositionRef = gl.getAttributeLocation(mCompiledShader, 446
                                                "aVertexPosition"); 447
}
```

Step A loads and compiles the shader code you defined in `index.html` by calling the `loadAndCompileShader()` function with the corresponding parameters. Step B attaches the compiled shaders and links the two shaders into a program. The reference to this program is stored in the variable `mCompiledShader`. After error checking in step C, step D locates and stores the reference to the `aVertexPosition` attribute defined in your vertex shader. 449
450
451
452
453
454

- 455 6. Define a function to allow the activation of the shader so that it
 456 can be used for drawing the square:

```

457   function activate() {
458     // Step A: access to the webgl context
459     let gl = core.getGL();
460
461     // Step B: identify the compiled shader to use
462     gl.useProgram(mCompiledShader);
463
464     // Step C: bind vertex buffer to attribute defined in vertex shader
465     gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer.get());
466     gl.vertexAttribPointer(this.mVertexPositionRef,
467       3,           // each element is a 3-float (x,y,z)
468       gl.FLOAT,    // data type is FLOAT
469       false,       // if the content is normalized vectors
470       0,           // number of bytes to skip in between elements
471       0);          // offsets to the first element
472     gl.enableVertexAttribArray(this.mVertexPositionRef);
473   }
  
```

472 In the code shown, step A sets the `gl` variable to the WebGL context through the `core`
 473 module. Step B loads the compiled shader program to the GPU memory, while step C
 474 binds the vertex buffer created in `vertex_buffer.js` to the `aVertexPosition` attribute
 475 defined in the vertex shader. The `gl.vertexAttribPointer()` function captures the
 476 fact that the vertex buffer was loaded with vertices of a unit square consisting of three
 477 floating-point values for each vertex position.

- 478 7. Lastly, provide access to the `init()` and `activate()` functions to the
 479 rest of the game engine by exporting them with the `export` statement:

```

480   export { init, activate }
  
```

481 **Note** Notice that the `loadAndCompileShader()` function is excluded from the
 482 export statement. This function is not needed elsewhere and thus, following the
 483 good development practice of hiding local implementation details, should remain
 484 private to this file.

The shader loading and compiling functionality is now defined. You can now utilize and activate these functions to draw with WebGL. 485
486

Set Up Drawing with WebGL 487

With the vertex data and shader functionality defined, you can now execute the following steps to draw with WebGL. Recall from the previous project that the initialization and drawing code is defined in the `core.js` file. Now open this file for editing. 488
489
490

1. Import the defined functionality from `vertex_buffer.js` and `shader_support.js` files: 491
492

```
import * as vertexBuffer from "./vertex_buffer.js"; 493
import * as simpleShader from "./shader_support.js"; 494
```

2. Modify the `initWebGL()` function to include the initialization of the vertex buffer and the shader program: 495
496

```
function initWebGL(htmlCanvasID) { 497
    let canvas = document.getElementById(htmlCanvasID); 498
    // Get standard or experimental webgl and bind to the Canvas area 499
    // store the results to the instance variable mGL 500
    mGL = canvas.getContext("webgl2") || 501
        canvas.getContext("experimental-webgl2"); 502

    if (mGL === null) { 503
        document.write("<br><b>WebGL 2 is not supported!</b>"); 504
        return; 505
    } 506
    mGL.clearColor(0.0, 0.8, 0.0, 1.0); // set the color to be cleared 507
    // 1. initialize buffer with vertex positions for the unit square 508
    vertexBuffer.init(); // function defined in the vertex_buffer.js 509

    // 2. now load and compile the vertex and fragment shaders 510
    simpleShader.init("VertexShader", "FragmentShader"); 511
    // the two shaders are defined in the index.html file 512
    // init() function is defined in shader_support.js file 513
}
```

CHAPTER 2 WORKING WITH HTML5 AND WEBGL

515 As shown in the code, after successfully obtaining the reference to the WebGL
516 context and setting the clear color, you should first call the `init()` function defined in
517 `vertex_buffer.js` to initialize the GPU vertex buffer with the unit square vertices and
518 then call the `init()` function defined in `shader_support.js` to load and compile the
519 vertex and fragment shaders.

520 3. Add a `drawSquare()` function for drawing the defined square:

```
521 function drawSquare() {  
522     // Step A: Activate the shader  
523     simpleShader.activate();  
  
524     // Step B. draw with the above settings  
525     mGL.drawArrays(mGL.TRIANGLE_STRIP, 0, 4);  
526 }
```

527 This code shows the steps to draw with WebGL. Step A activates the shader program
528 to use. Step B issues the WebGL draw command. In this case, you are issuing a command
529 to draw the four vertices as two connected triangles that form a square.

530 4. Now you just need to modify the `window.onload` function to call
531 the newly defined `drawSquare()` function:

```
532 window.onload = function() {  
533     initWebGL("GLCanvas"); // Binds mGL context to WebGL functionality  
534     clearCanvas(); // Clears the GL area  
535     drawSquare(); // Draws one square  
536 }
```

537 5. Finally, provide access to the WebGL context to the rest of the
538 engine by exporting the `getGL()` function. Remember that this
539 function is imported and has been called to access the WebGL
540 context in both `vertex_buffer.js` and `simple_shader.js`.

```
541 export {getGL}
```

542 Recall that the function that is bounded to `window.onload` will be invoked after
543 `index1.html` has been loaded by the web browser. For this reason, WebGL will be
544 initialized, the canvas cleared to light green, and a white square will be drawn. You can
545 refer to the source code in the `chapter2/2.3.draw_one_square` project for the entire
546 system described.

Observations

547

Run the project and you will see a white rectangle on a green canvas. What happened to the square? Remember that the vertex position of your 1×1 square was defined at locations $(\pm 0.5, \pm 0.5)$. Now observe the project output: the white rectangle is located in the middle of the green canvas covering exactly half of the canvas's width and height. As it turns out, WebGL draws vertices within the ± 1.0 range onto the entire defined drawing area. In this case, the ± 1.0 in the x dimension is mapped to 640 pixels, while the ± 1.0 in the y dimension is mapped to 480 pixels (the created canvas dimension is 640×480). The 1×1 square is drawn onto a 640×480 area, or an area with an aspect ratio of 4:3. Since the 1:1 aspect ratio of the square does not match the 4:3 aspect ratio of the display area, the square shows up as a 4:3 rectangle. This problem will be resolved later in the next chapter.

548
549
550
551
552
553
554
555
556
557

You can try editing the fragment shader in `index.html` by changing the color set in the `gl_FragColor` function to alter the color of the white square. Notice that a value of less than 1 in the alpha channel does not result in the white square becoming transparent. Transparency of drawn primitives will be discussed in later chapters.

558
559
560
561

Finally, note that this project defines three separate files and hides information with the JavaScript import/export statements. The functionality defined in these files with the corresponding import and export statements is referred to as JavaScript modules. A module can be considered as a global singleton object and is excellent for hiding implementation details. The `loadAndCompileShader()` function in the `shader_support` module serves as a great example of this concept. However, modules are not well suited for supporting abstraction and specialization. In the next sections, you will begin to work with JavaScript classes to further encapsulate portions of this example to form the basis of the game engine framework.

562
563
564
565
566
567
568
569
570

Abstraction with JavaScript Classes

571

The previous project decomposed the drawing of a square into logical modules and implemented the modules as files containing global functions. In software engineering, this process is referred to as functional decomposition, and the implementation is referred to as procedural programming. Procedural programming often results in solutions that are well structured and easy to understand. This is why functional decomposition and procedural programming are often used to prototype concepts or to learn new techniques.

572
573
574
575
576
577
578

579 This project enhances the Draw One Square project with object-oriented
580 analysis and programming to introduce data abstraction. As additional concepts are
581 introduced and as the game engine complexity grows, proper data abstraction supports
582 straightforward design, behavior specialization, and code reuse through inheritance.

583 **The JavaScript Objects Project**

584 This project demonstrates how to abstract the global functions from the Draw One
585 Square project into JavaScript classes and objects. This object-oriented abstraction
586 will result in a framework that offers manageability and expandability for subsequent
587 projects. As illustrated in Figure 2-7, when running, this project displays a white
588 rectangle in a greenish canvas, identical to that from the Draw One Square project. The
589 source code to this project can be found in the chapter2/2.4.javascript_objects
590 folder.

this figure will be printed in b/w

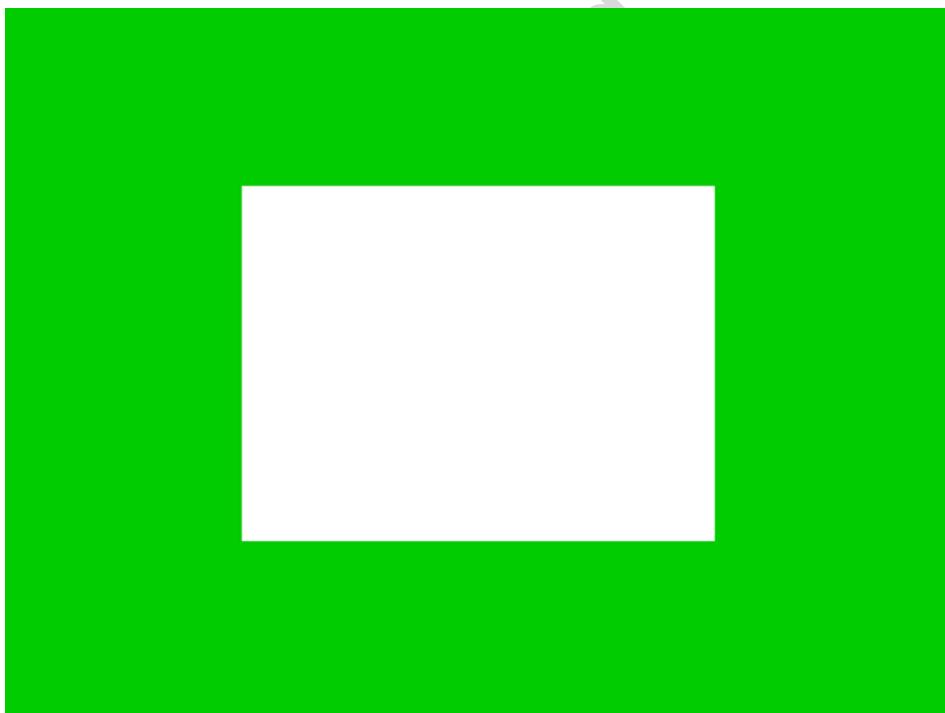


Figure 2-7. Running the JavaScript Objects project

The goals of the project are as follows:

- To separate the code for the game engine from the code for the game logic 591
- To understand how to build abstractions with JavaScript classes and objects 593
- To understand how to build abstractions with JavaScript classes and objects 594
- To understand how to build abstractions with JavaScript classes and objects 595

The steps for creating this project are as follows:

1. Create separate folders to organize the source code for the game engine and the logic of the game. 597
2. Define a JavaScript class to abstract the `simple_shader` and work with an instance of this class. 598
3. Define a JavaScript class to implement the drawing of one square, which is the logic of your simple game for now. 600
3. Define a JavaScript class to implement the drawing of one square, which is the logic of your simple game for now. 601
3. Define a JavaScript class to implement the drawing of one square, which is the logic of your simple game for now. 602

Source Code Organization

Create a new HTML5 project with VS Code by creating a new folder and adding a source code folder named `src`. Within `src`, create `engine` and `my_game` as subfolders, as illustrated in Figure 2-8.

this figure will be printed in b/w

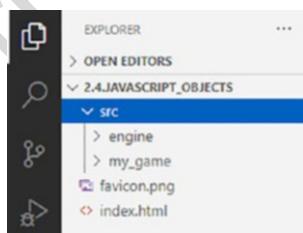


Figure 2-8. Creating `engine` and `my_game` under the `src` folder

The `src/engine` folder will contain all the source code to the game engine, and the `src/my_game` folder will contain the source for the logic of your game. It is important to organize source code diligently because the complexity of the system and the number of files will increase rapidly as more concepts are introduced. A well-organized source code structure facilitates understanding and expansion.

612 **Tip** The source code in the `my_game` folder implements the game by relying on
613 the functionality provided by the game engine defined in the `engine` folder. For
614 this reason, in this book, the source code in the `my_game` folder is often referred to
615 as the *client* of the game engine.

616

Abstracting the Game Engine

617 A completed game engine would include many self-contained subsystems to fulfill
618 different responsibilities. For example, you may be familiar with or have heard of
619 the geometry subsystem for managing the geometries to be drawn, the resource
620 management subsystem for managing images and audio clips, the physics subsystem for
621 managing object interactions, and so on. In most cases, the game engine would include
622 one unique instance of each of these subsystems, that is, one instance of the geometry
623 subsystem, of the resource management subsystem, of the physics subsystem, and so on.

624 These subsystems will be covered in later chapters of this book. This section focuses
625 on establishing the mechanism and organization for implementing this single-instance
626 or singleton-like functionality based on the JavaScript module you have worked with in
627 the previous project.

628 **Note** All module and instance variable names begin with an m and are followed
629 by a capital letter, as in `mVariable`. Though not enforced by JavaScript, you
630 should never access a module or instance variable from outside the module/class.
631 For example, you should never access `core.mGL` directly; instead, call the `core.getGL()` function to access the variable.

633

The Shader Class

634 Although the code in the `shader_support.js` file from the previous project properly
635 implements the required functionality, the variables and functions do not lend
636 themselves well to behavior specialization and code reuse. For example, in the cases
637 when different types of shaders are required, it can be challenging to modify the
638 implementation while achieving behavior and code reuse. This section follows the
639 object-oriented design principles and defines a `SimpleShader` class to abstract the

behaviors and hide the internal representations of shaders. Besides the ability to create multiple instances of the SimpleShader object, the basic functionality remains largely unchanged.

Note Module identifiers begin with lower case, for example, core or vertexBuffer. Class names begin with upper case, for example, SimpleShader or MyGame.

1. Create a new source file in the src/engine folder and name the file simple_shader.js to implement the SimpleShader class.

646
647

2. Import both the core and vertex_buffer modules:

import * as core from "./core.js";
import * as vertexBuffer from "./vertex_buffer.js";

3. Declare the SimpleShader as a JavaScript class:

```
class SimpleShader {  
    ... implementation to follow  
}
```

4. Define the constructor within the SimpleShader class to load, compile, and link the vertex and fragment shaders into a program and to create a reference to the aVertexPosition attribute in the vertex shader for loading the square vertex positions from the WebGL vertex buffer for drawing:

```
constructor(vertexShaderID, fragmentShaderID) {  
    // instance variables  
    // Convention: all instance variables: mVariables  
    this.mCompiledShader = null; // ref to compiled shader in webgl  
    this.mVertexPositionRef = null; // ref to VertexPosition in shader  
  
    let gl = core.getGL();  
    // Step A: load and compile vertex and fragment shaders  
    this.mVertexShader = loadAndCompileShader(vertexShaderID,  
                                              gl.VERTEX_SHADER);
```

```

669     this.mFragmentShader = loadAndCompileShader(fragmentShaderID,
670                                         gl.FRAGMENT_SHADER);
671
672     // Step B: Create and link the shaders into a program.
673     this.mCompiledShader = gl.createProgram();
674     gl.attachShader(this.mCompiledShader, this.mVertexShader);
675     gl.attachShader(this.mCompiledShader, this.mFragmentShader);
676     gl.linkProgram(this.mCompiledShader);
677
678     // Step C: check for error
679     if (!gl.getProgramParameter(this.mCompiledShader, gl.LINK_STATUS)) {
680         throw new Error("Error linking shader");
681         return null;
682     }
683
684     // Step D: reference to aVertexPosition attribute in the shaders
685     this.mVertexPositionRef = gl.getAttribLocation(
686                                         this.mCompiledShader, "aVertexPosition");
687 }
```

Notice that this constructor is essentially the same as the `init()` function in the `shader_support.js` module from the previous project.

687 Note The JavaScript `constructor` keyword defines the constructor of a class.

- 688 5. Add a method to the `SimpleShader` class to activate the shader
689 for drawing. Once again, similar to your `activate()` function in
690 `shader_support.js` from previous project.

```

691     activate() {
692         let gl = core.getGL();
693         gl.useProgram(this.mCompiledShader);
694
695         // bind vertex buffer
696         gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer.get());
697         gl.vertexAttribPointer(this.mVertexPositionRef,
698             3,           // each element is a 3-float (x,y,z)
699             gl.FLOAT,    // data type is FLOAT
```

```

        false,           // if the content is normalized vectors      699
        0,                // number of bytes to skip in between elements 700
        0);              // offsets to the first element                 701
    gl.enableVertexAttribArray(this.mVertexPositionRef);          702
}

6. Add a private method, which cannot be accessed from outside      704
   the simple_shader.js file, by creating a function outside the      705
   SimpleShader class to perform the actual loading and compiling      706
   functionality:                                                 707

function loadAndCompileShader(id, shaderType) {                      708
    let shaderSource = null, compiledShader = null;                  709
    let gl = core.getGL();                                         710

    // Step A: Get the shader source from index.html             711
    let shaderText = document.getElementById(id);                  712
    shaderSource = shaderText.firstChild.textContent;            713

    // Step B: Create shader based on type: vertex or fragment 714
    compiledShader = gl.createShader(shaderType);                 715

    // Step C: Compile the created shader                         716
    gl.shaderSource(compiledShader, shaderSource);               717
    gl.compileShader(compiledShader);                            718

    // Step D: check for errors and return results (null if error) 719
    // The log info is how shader compilation errors are displayed 720
    // This is useful for debugging the shaders.                   721
    if (!gl.getShaderParameter(compiledShader, gl.COMPILE_STATUS)) { 722
        throw new Error("A shader compiling error occurred: " + 723
                        gl.getShaderInfoLog(compiledShader));          724
    }

    return compiledShader;                                       726
}                                                               727

```

728 Notice that this function is identical to the one you created in `shader_support.js`.

729 **Note** The JavaScript `#` prefix that defines private members is not used in this
730 book because the lack of visibility from subclasses complicates specialization of
731 behaviors in inheritance.

- 732 7. Finally, add an export for the `SimpleShader` class such that it can
733 be accessed and instantiated outside of this file:

734 `export default SimpleShader;`

735 **Note** The `default` keyword signifies that the name `SimpleShader` cannot be
736 changed by `import` statements.

737 The Core of the Game Engine: `core.js`

738 The core contains common functionality shared by the entire game engine. This
739 can include one-time initialization of the WebGL (or GPU), shared resources, utility
740 functions, and so on.

- 741 1. Create a copy of your `core.js` under the new folder `src/engine`.
742 2. Define a function to create a new instance of the `SimpleShader` object:

```
743 // The shader
744 let mShader = null;
745 function createShader() {
746     mShader = new SimpleShader(
747         "VertexShader",      // IDs of the script tag in the index.html
748         "FragmentShader"); // 
749 }
```

3. Modify the `initWebGL()` function to focus on only initializing the WebGL as follows:

```
// initialize the WebGL  
function initWebGL(htmlCanvasID) {  
    let canvas = document.getElementById(htmlCanvasID);  
  
    // Get standard or experimental webgl and binds to the Canvas area  
    // store the results to the instance variable mGL  
    mGL = canvas.getContext("webgl2") ||  
        canvas.getContext("experimental-webgl2");  
  
    if (mGL === null) {  
        document.write("<br><b>WebGL 2 is not supported!</b>");  
        return;  
    }  
}
```

4. Create an `init()` function to perform engine-wide system initialization, which includes initializing of WebGL and the vertex buffer and creating an instance of the simple shader:

```
function init(htmlCanvasID) {  
    initWebGL(htmlCanvasID); // setup mGL  
    vertexBuffer.init(); // setup mGLVertexBuffer  
    createShader(); // create the shader  
}
```

5. Modify the clear canvas function to parameterize the color to be cleared to:

```
function clearCanvas(color) {  
    mGL.clearColor(color[0], color[1], color[2], color[3]);  
    mGL.clear(mGL.COLOR_BUFFER_BIT); // clear to the color set  
}
```

- 778 6. Export the relevant functions for access by the rest of the game
 779 engine:

780 export { getGL, init, clearCanvas, drawSquare }

- 781 7. Finally, remove the `window.onload` function as the behavior of the
 782 actual game should be defined by the client of the game engine or,
 783 in this case, the `MyGame` class.

784 The `src/engine` folder now contains the basic source code for the entire game
 785 engine. Due to these structural changes to your source code, the game engine can now
 786 function as a simple library that provides functionality for creating games or a simple
 787 application programming interface (API). For now, your game engine consists of three
 788 files that support the initialization of WebGL and the drawing of a unit square, the core
 789 module, the `vertex_buffer` module, and the `SimpleShader` class. New source files and
 790 functionality will continue to be added to this folder throughout the remaining projects.
 791 Eventually, this folder will contain a complete and sophisticated game engine. However,
 792 the core library-like framework defined here will persist.

793 The Client Source Code

794 The `src/my_game` folder will contain the actual source code for the game. As mentioned,
 795 the code in this folder will be referred to as the *client* of the game engine. For now, the
 796 source code in the `my_game` folder will focus on drawing a simple square by utilizing the
 797 functionality of the simple game engine you defined.

- 798 1. Create a new source file in the `src/my_game` folder, or the *client*
 799 folder, and name the file `my_game.js`.

- 800 2. Import the `core` module as follows:

801 import * as engine from "../engine/core.js";

- 802 3. Define `MyGame` as a JavaScript class and add a constructor to
 803 initialize the game engine, clear the canvas, and draw the square:

```
804     class MyGame {
  805        constructor(htmlCanvasID) {
  806           // Step A: Initialize the game engine
  807           engine.init(htmlCanvasID);
```

```

// Step B: Clear the canvas          808
engine.clearCanvas([0, 0.8, 0, 1]); 809

// Step C: Draw the square          810
engine.drawSquare();               811

}

}

4. Bind the creation of a new instance of the MyGame object to the      814
   window.onload function:          815

window.onload = function() {           816
    new MyGame('GLCanvas');          817
}
}

5. Finally, modify the index.html to load the game client rather than      819
   the engine core.js within the head element:                          820

<script type="module" src="./src/my_game/my_game.js"></script>      821

```

Observations

Although you're accomplishing the same tasks as with the previous project, with this project, you have created an infrastructure that supports subsequent modifications and expansions of your game engine. You have organized your source code into separate and logical folders, organized the singleton-like modules to implement core functionality of the engine, and gained experience with abstracting the SimpleShader class that will support future design and code reuse. With the engine now comprised of well-defined modules and objects with clean interface methods, you can now focus on learning new concepts, abstracting the concepts, and integrating new implementation source code into your engine.

Separating GLSL from HTML

Thus far in your projects, the GLSL shader code is embedded in the HTML source code of index.html. This organization means that new shaders must be added through the editing of the index.html file. Logically, GLSL shaders should be organized separately

836 from HTML source files; logically, continuously adding to `index.html` will result in
837 a cluttered and unmanageable file that would become difficult to work with. For these
838 reasons, the GLSL shaders should be stored in separate source files.

839 The Shader Source File Project

840 This project demonstrates how to separate the GLSL shaders into separate files. As
841 illustrated in Figure 2-9, when running this project, a white rectangle is displayed on
842 a greenish canvas, identical to the previous projects. The source code to this project is
843 defined in the `chapter2/2.5.shader_source_files` folder.

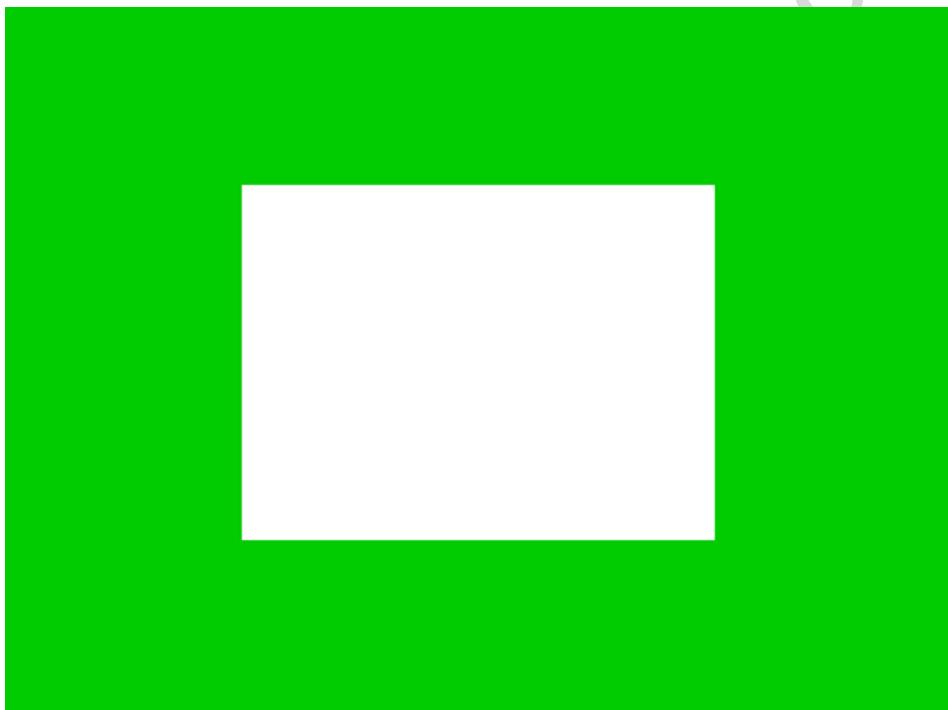


Figure 2-9. Running the Shader Source File project

844 The goals of the project are as follows:

- 845 • To separate the GLSL shaders from the HTML source code
846 • To demonstrate how to load the shader source code files during
847 runtime

Loading Shaders in SimpleShader

Instead of loading the GLSL shaders as part of the HTML document, the `loadAndCompileShader()` in `SimpleShader` can be modified to load the GLSL shaders as separate files:

1. Continue from the previous project, open the `simple_shader.js` file, and edit the `loadAndCompileShader()` function, to receive a file path instead of an HTML ID:

```
function loadAndCompileShader(filePath, shaderType)
```

2. Within the `loadAndCompileShader()` function, replace the HTML element retrieval code in step A with the following XMLHttpRequest to load a file:

```
let xmlReq, shaderSource = null, compiledShader = null;
```

```
let gl = core.getGL();
```

```
// Step A: Request the text from the given file location.
```

```
xmlReq = new XMLHttpRequest();
```

```
xmlReq.open('GET', filePath, false);
```

```
try {
```

```
    xmlReq.send();
```

```
} catch (error) {
```

```
    throw new Error("Failed to load shader: "
```

```
        + filePath
```

```
        + " [Hint: you cannot double click to run this project. ]"
```

```
        + "The index.html file must be loaded by a web-server.]");
```

```
    return null;
```

```
}
```

```
shaderSource = xmlReq.responseText;
```

```
if (shaderSource === null) {
```

```
    throw new Error("WARNING: Loading of:" + filePath + " Failed!");
```

```
    return null;
```

```
}
```

878 Notice that the file loading will occur synchronously where the web browser will
 879 actually stop and wait for the completion of the `xmlReq.open()` function to return with
 880 the content of the opened file. If the file should be missing, the opening operation will
 881 fail, and the response text will be null.

882 The synchronized “stop and wait” for the completion of `xmlReq.open()` function
 883 is inefficient and may result in slow loading of the web page. This shortcoming will
 884 be addressed in Chapter 4 when you learn about the asynchronous loading of game
 885 resources.

886 **Note** The `XMLHttpRequest()` object requires a running web server to fulfill
 887 the HTTP get request. This means you will be able to test this project from within
 888 the VS Code with the installed “Go Live” extension. However, unless there is a web
 889 server running on your machine, you will not be able to run this project by
 890 double-clicking the `index.html` file directly. This is because there is no server to
 891 fulfill the HTTP get requests and the GLSL shader loading will fail.

892 With this modification, the `SimpleShader` constructor can now be modified to
 893 receive and forward file paths to the `loadAndCompileShader()` function instead of the
 894 HTML element IDs.

895 Extracting Shaders into Their Own Files

896 The following steps retrieve the source code of the vertex and fragment shaders from the
 897 `index.html` file and create separate files for storing them:

- 898 1. Create a new folder that will contain all of the GLSL shader
 899 source code files in the `src` folder, and name it `glsl_shaders`, as
 900 illustrated in Figure 2-10.

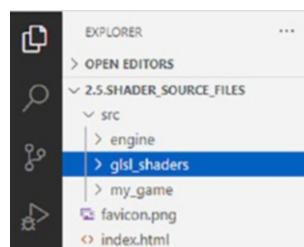


Figure 2-10. Creating the `glsl_shaders` folder

2. Create two new text files within the `glsl_shaders` folder, and
 name them `simple_vs.glsl` and `white_fs.glsl` for simple vertex
 shader and white fragment shader. 901
 902
 903

Note All GLSL shader source code files will end with the `.glsl` extension. The
 vs in the shader file names signifies that the file contains a vertex shader, while fs
 signifies a fragment shader. 904
 905
 906

3. Create the GLSL vertex shader source code by editing `simple_`
`vs.glsl` and pasting the vertex shader code in the `index.html` file
 from the previous project: 907
 908
 909

```
attribute vec3 aVertexPosition; // Vertex shader expects one position 910
void main(void) { 911
    // Convert the vec3 into vec4 for scan conversion and 912
    // assign to gl_Position to pass the vertex to the fragment shader 913
    gl_Position = vec4(aVertexPosition, 1.0); 914
}
```

915

4. Create the GLSL fragment shader source code by editing `white_`
`fs.glsl` and pasting the fragment shader code in the `index.html`
 file from the previous project: 916
 917
 918

```
precision mediump float; // precision for float computation 919
void main(void) { 920
    // for every pixel called (within the square) sets 921
    // constant color white with alpha-channel value of 1.0 922
    gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0); 923
}
```

924

Cleaning Up HTML Code

925

With vertex and fragment shaders being stored in separate files, it is now possible to
 clean up the `index.html` file such that it contains only HTML code: 926
 927

- 928 1. Remove all the GLSL shader code from `index.html`, such that this
 929 file becomes as follows:

```

930 <!DOCTYPE html>
931 <html>
932   <head>
933     <title>Example 2.5: The Shader Source File Project</title>
934     <link rel = "icon" type = "image/x-icon" href = "./favicon.png">
935     <!-- there are javascript source code contained in
936         the external source files
937         -->
938     <!-- Client game code -->
939     <script type="module" src = "./src/my_game/my_game.js"></script>
940   </head>
941
942   <body>
943     <canvas id="GLCanvas" width="640" height="480">
944       <!-- GLCanvas is the area we will draw in: a 640x480 area -->
945       Your browser does not support the HTML5 canvas.
946       <!-- this message will show only if WebGL clearing failed -->
947     </canvas>
948   </body>
949 </html>
```

949 Notice that `index.html` no longer contains any GLSL shader code and only a single
 950 reference to JavaScript code. With this organization, the `index.html` file can properly
 951 be considered as representing the web page where you do not need to edit this file to
 952 modify the shaders from now on.

- 953 2. Modify the `createShader()` function in `core.js` to load the
 954 shader files instead of HTML element IDs:

```

955 function createShader() {
956   mShader = new SimpleShader(
957     "src/glsl_shaders/simple_vs.glsl", // Path to VertexShader
958     "src/glsl_shaders/white_fs.glsl"); // Path to FragmentShader
959 }
```

Source Code Organization

The separation of logical components in the engine source code has progressed to the following state:

- `index.html`: This is the file that contains the HTML code that defines the canvas on the web page for the game and loads the source code for your game.
- `src/glsl_shaders`: This is the folder that contains all the GLSL shader source code files that draws the elements of your game.
- `src/engine`: This is the folder that contains all the source code files for your game engine.
- `src/my_game`: This is the client folder that contains the source code for the actual game.

Changing the Shader and Controlling the Color

With GLSL shaders being stored in separate source code files, it is now possible to edit or replace the shaders with relatively minor changes to the rest of the source code. The next project demonstrates this convenience by replacing the restrictive constant white color fragment shader, `white_fs.glsl`, with a shader that can be parameterized to draw with any color.

The Parameterized Fragment Shader Project

This project replaces `white_fs.glsl` with a `simple_fs.glsl` that supports the drawing with any color. Figure 2-11 shows the output of running the Parameterized Fragment Shader project; notice that a red square replaces the white square from the previous projects. The source code for this project is defined in the `chapter2/2.6.parameterized_fragment_shader` folder.

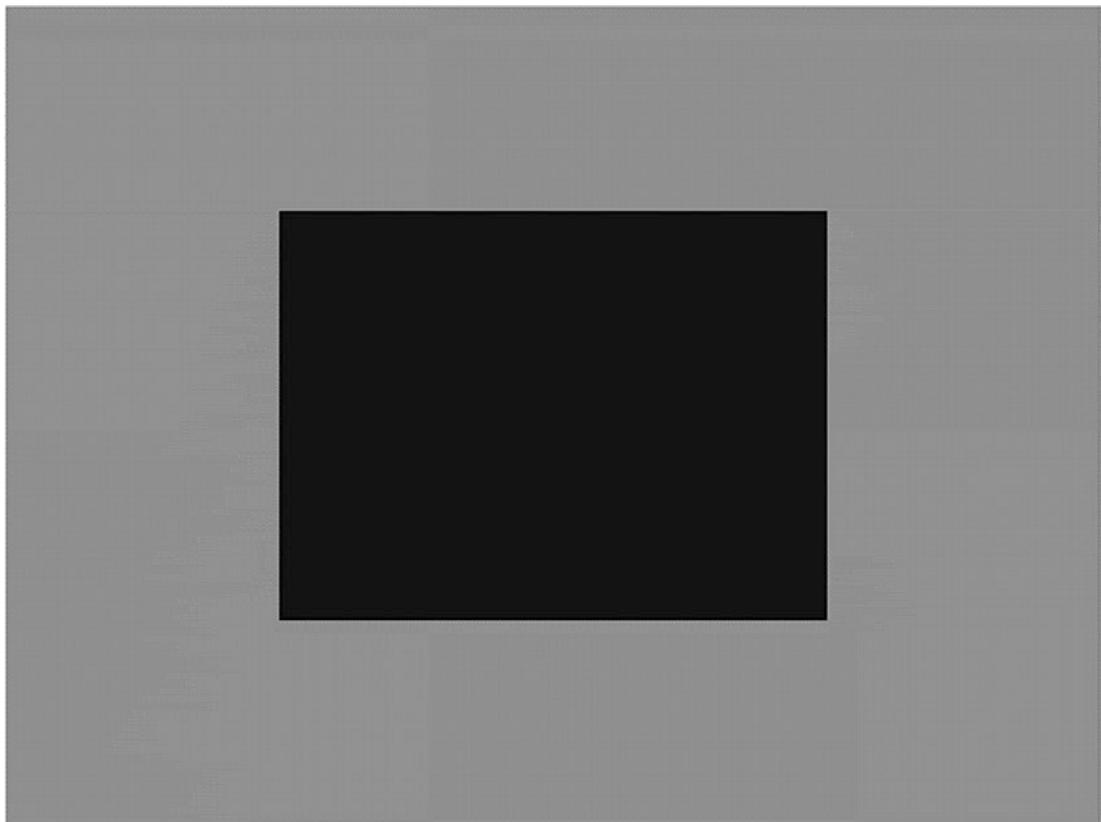


Figure 2-11. Running the Parameterized Fragment Shader project

984 The goals of the project are as follows:

- 985 • To gain experience with creating a GLSL shader in the source code
986 structure
- 987 • To learn about the `uniform` variable and define a fragment shader
988 with the color parameter

Defining the simple_fs.glsl Fragment Shader

989

A new fragment shader needs to be created to support changing the pixel color for each draw operation. This can be accomplished by creating a new GLSL fragment shader in the src/glsl_shaders folder and name it `simple_fs.glsl`. Edit this file to add the following:

```
precision mediump float; // precision for float computation
// Color of pixel
uniform vec4 uPixelColor;
void main(void) {
    // for every pixel called sets to the user specified color
    gl_FragColor = uPixelColor;
}
```

Recall that the `GLSL` attribute keyword identifies data that changes for every vertex position. In this case, the `uniform` keyword denotes that a variable is constant for all the vertices. The `uPixelColor` variable can be set from JavaScript to control the eventual pixel color. The `precision mediump` keywords define the floating precisions for computations.

1000

1001

1002

1003

1004

Note Floating-point precision trades the accuracy of computation for performance. Please follow the references in Chapter 1 for more information on WebGL.

1005

1006

Modify the SimpleShader to Support the Color Parameter

1007

The `SimpleShader` class can now be modified to gain access to the new `uPixelColor` variable:

1. Edit `simple_shader.js` and add a new instance variable for referencing the `uPixelColor` in the constructor:

```
this.mPixelColorRef = null; // pixelColor uniform in fragment shader
```

1011

2. Add code to the end of the constructor to create the reference to the `uPixelColor`:

```
// Step E: Gets uniform variable uPixelColor in fragment shader
this.mPixelColorRef = gl.getUniformLocation(
    this.mCompiledShader, "uPixelColor");
```

1014

1015

1016

```

1017     3. Modify the shader activation to allow the setting of the pixel color
1018         via the uniform4fv() function:

1019 activate(pixelColor) {
1020     let gl = core.getGL();
1021     gl.useProgram(this.mCompiledShader);

1022     // bind vertex buffer
1023     gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer.get());
1024     gl.vertexAttribPointer(this.mVertexPositionRef,
1025         3,           // each element is a 3-float (x,y,z)
1026         gl.FLOAT,    // data type is FLOAT
1027         false,       // if the content is normalized vectors
1028         0,           // number of bytes to skip in between elements
1029         0);          // offsets to the first element
1030     gl.enableVertexAttribArray(this.mVertexPositionRef);

1031     // load uniforms
1032     gl.uniform4fv(this.mPixelColorRef, pixelColor);
1033 }

```

1034 The `gl.uniform4fv()` function copies four floating-point values from the
 1035 `pixelColor` float array to the WebGL location referenced by `mPixelColorRef` or the
 1036 `uPixelColor` in the `simple_fs.gls1` fragment shader.

1037 Drawing with the New Shader

1038 To test `simple_fs.gls1`, modify the `core.js` module to create a `SimpleShader` with the
 1039 new `simple_fs` and use the parameterized color when drawing with the new shader:

```

1040 function createShader() {
1041     mShader = new SimpleShader(
1042         "src/gls1_shaders/simple_vs.gls1", // Path to the VertexShader
1043         "src/gls1_shaders/simple_fs.gls1"); // Path to the FragmentShader
1044 }

1045 function drawSquare(color) {
1046     // Step A: Activate the shader
1047     mShader.activate(color);

```

```
// Step B: Draw with currently activated geometry and shader          1048  
mGL.drawArrays(mGL.TRIANGLE_STRIP, 0, 4);                          1049  
}                                                               1050
```

Lastly, edit the constructor of the `MyGame` class to include a color when drawing the square, in this case, red: 1051
1052

```
// Step C: Draw the square in red                                     1053  
engine.drawSquare([1, 0, 0, 1]);                                     1054
```

Notice that a color value, an array of four floats, is now required with the new `simple_fs.glsl` (instead of `white_fs`) shader and that it is important to pass in the drawing color when activating the shader. With the new `simple_fs`, you can now experiment with drawing the squares with any desired color. 1055
1056
1057
1058

As you have experienced in this project, the source code structure supports simple and localized changes when the game engine is expanded or modified. In this case, only changes to the `simple_shader.js` file and minor modifications to `core.js` and the `my_game.js` were required. This demonstrates the benefit of proper encapsulation and source code organization. 1059
1060
1061
1062
1063

Summary

By this point, the game engine is simple and supports only the initialization of WebGL and the drawing of one colored square. However, through the projects in this chapter, you have gained experience with the techniques required to build an excellent foundation for the game engine. You have also structured the source code to support further complexity with limited modification to the existing code base, and you are now ready to further encapsulate the functionality of the game engine to facilitate additional features. The next chapter will focus on building a proper framework in the game engine to support more flexible and configurable drawings. 1064
1065
1066
1067
1068
1069
1070
1071
1072

CHAPTER 3

1

Drawing Objects in the World

2

3

After completing this chapter, you will be able to

- Create and draw multiple rectangular objects
- Control the position, size, rotation, and color of the created rectangular objects
- Define a coordinate system to draw from
- Define a target subarea on the canvas to draw to
- Work with abstract representations of Renderable objects, transformation operators, and cameras

4

5

6

7

8

9

10

11

Introduction

12

Ideally, a video game engine should provide proper abstractions to support designing and building games in meaningful contexts. For example, when designing a soccer game, instead of a single square with a fixed ± 1.0 drawing range, a game engine should provide proper utilities to support designs in the context of players running on a soccer field. This high-level abstraction requires the encapsulation of basic operations with data hiding and meaningful functions for setting and receiving the desired results.

13

14

15

16

17

18

While this book is about building abstractions for a game engine, this chapter focuses on creating the fundamental abstractions to support drawing. Based on the soccer game example, the support for drawing in an effective game engine would likely include the ability to easily create the soccer players, control their size and orientations, and allow them to be moved and drawn on the soccer field. Additionally, to support

19

20

21

22

23

proper presentation, the game engine must allow drawing to specific subregions on the canvas so that a distinct game status can be displayed at different subregions, such as the soccer field in one subregion and player statistics and scores in another subregion.

This chapter identifies proper abstraction entities for the basic drawing operations, introduces operators that are based on foundational mathematics to control the drawing, overviews the WebGL tools for configuring the canvas to support subregion drawing, defines JavaScript classes to implement these concepts, and integrates these implementations into the game engine while maintaining the organized structure of the source code.

Encapsulating Drawing

Although the ability to draw is one of the most fundamental functionalities of a game engine, the details of how drawings are implemented are generally a distraction to gameplay programming. For example, it is important to create, control the locations of, and draw soccer players in a soccer game. However, exposing the details of how each player is actually defined (by a collection of vertices that form triangles) can quickly overwhelm and complicate the game development process. Thus, it is important for a game engine to provide a well-defined abstraction interface for drawing operations.

With a well-organized source code structure, it is possible to gradually and systematically increase the complexity of the game engine by implementing new concepts with localized changes to the corresponding folders. The first task is to expand the engine to support the encapsulation of drawing such that it becomes possible to manipulate drawing operations as a logical entity or as an object that can be rendered.

Note In the context of computer graphics and video games, the word *render* refers to the process of changing the color of pixels corresponding to an abstract representation. For example, in the previous chapter, you learned how to render a square.

The Renderable Objects Project

50

This project introduces the `Renderable` class to encapsulate the drawing operation.
Over the next few projects, you will learn more supporting concepts to refine the
implementation of the `Renderable` class such that multiple instances can be created and
manipulated. Figure 3-1 shows the output of running the `Renderable Objects` project. The
source code to this project is defined in the `chapter3/3.1.renderable_objects` folder.

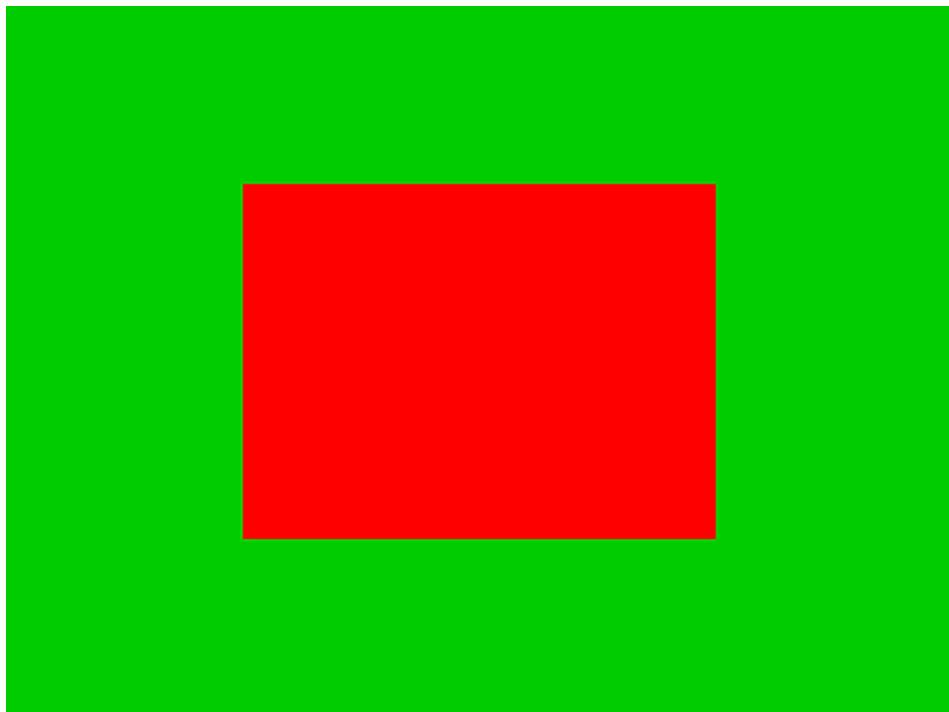
51

52

53

54

55



this figure will be printed in b/w

Figure 3-1. Running the `Renderable Objects` project

The goals of the project are as follows:

- To reorganize the source code structure in anticipation for functionality increases
- To support game engine internal resource sharing
- To introduce a systematic interface for the game developer via the `index.js` file

- 56 • To begin the process of building a class to encapsulate drawing
57 operations by first abstracting the related drawing functionality
58 • To demonstrate the ability to create multiple Renderable objects

59 **Source Code Structure Reorganization**

60 Before introducing additional functionality to the game engine, it is important to
61 recognize some shortfalls of the engine source code organization from the previous
62 project. In particular, take note of the following:

- 63 1. The `core.js` source code file contains the WebGL interface,
64 engine initialization, and drawing functionalities. These should
65 be modularized to support the anticipated increase in system
66 complexity.
- 67 2. A system should be defined to support the sharing of game engine
68 internal resources. For example, `SimpleShader` is responsible for
69 interfacing from the game engine to the GLSL shader compiled
70 from the `simple_vs.glsl` and `simple_fs.glsl` source code files.
71 Since there is only one copy of the compiled shader, there only
72 needs to be a single instance of the `SimpleShader` object. The
73 game engine should facilitate this by allowing the convenient
74 creation and sharing of the object.
- 75 3. As you have experienced, the JavaScript `export` statement can be
76 an excellent tool for hiding detailed implementations. However, it
77 is also true that determining which classes or modules to import
78 from a number of files can be confusing and overwhelming in a
79 large and complex system, such as the game engine you are about
80 to develop. An easy to work with and systematic interface should
81 be provided such that the game developer, users of the game
82 engine, can be insulated from these details.

83 In the following section, the game engine source code will be reorganized to address
84 these issues.

Define a WebGL-Specific Module

The first step in source code reorganization is to recognize and isolate functionality that is internal and should not be accessible by the clients of the game engine:

1. In your project, under the `src/engine` folder, create a new folder and name it `core`. From this point forward, this folder will contain all functionality that is internal to the game engine and will not be exported to the game developers.
2. You can cut and paste the `vertex_buffer.js` source code file from the previous project into the `src/engine/core` folder. The details of the primitive vertices are internal to the game engine and should not be visible or accessible by the clients of the game engine.
3. Create a new source code file in the `src/engine/core` folder, name it `gl.js`, and define WebGL's initialization and access methods:

```
"use strict"

let mCanvas = null;
let mGL = null;

function get() { return mGL; }

function init(htmlCanvasID) {
    mCanvas = document.getElementById(htmlCanvasID);
    if (mCanvas == null)
        throw new Error("Engine init [" +
                        htmlCanvasID + "] HTML element id not found");

    // Get standard or experimental webgl and binds to the Canvas area
    // store the results to the instance variable mGL
    mGL = mCanvas.getContext("webgl2") ||
          mCanvas.getContext("experimental-webgl2");
}
```

```

113     if (mGL === null) {
114         document.write("<br><b>WebGL 2 is not supported!</b>");
115         return;
116     }
117 }

118 export {init, get}

```

119 Notice that the `init()` function is identical to the `initWebGL()` function in `core.js`
120 from the previous project. Unlike the previous `core.js` source code file, the `gl.js` file
121 contains only WebGL-specific functionality.

122 Define a System for Internal Shader Resource Sharing

123 Since only a single copy of the GLSL shader is created and compiled from the
124 `simple_vs.gls1` and `simple_fs.gls1` source code files, only a single copy of
125 `SimpleShader` object is required within the game engine to interface with the compiled
126 shader. You will now create a simple resource sharing system to support future additions
127 of different types of shaders.

128 Create a new source code file in the `src/engine/core` folder, name it
129 `shader_resources.js`, and define the creation and accessing methods for
130 `SimpleShader`.

131 **Note** Recall from the previous chapter that the `SimpleShader` class is defined
132 in the `simple_shader.js` file which is located in the `src/engine` folder.
133 Remember to copy all relevant source code files from the previous project.

```

134 "use strict";

135 import SimpleShader from "../simple_shader.js";

136 // Simple Shader
137 let kSimpleVS = "src/gls1_shaders/simple_vs.gls1"; // to VertexShader
138 let kSimpleFS = "src/gls1_shaders/simple_fs.gls1"; // to FragmentShader
139 let mConstColorShader = null;

```

```

function createShaders() { 140
    mConstColorShader = new SimpleShader(kSimpleVS, kSimpleFS); 141
}
} 142

function init() { 143
    createShaders(); 144
}
} 145
function getConstColorShader() { return mConstColorShader; } 146
export {init, getConstColorShader} 147

```

Note Variables referencing constant values have names that begin with lowercase “k,” as in kSimpleVS.

Since the shader_resources module is located in the src/engine/core folder, the defined shaders are shared within and cannot be accessed from the clients of the game engine.

Define an Access File for the Game Developer

You will define an engine access file, index.js, to implement the fundamental functions of the game engine and to serve a similar purpose as a C++ header file, the import statement in Java, or the using statement in C#, where functionality can be readily accessed without in-depth knowledge of the engine source code structure. That is, by importing index.js, the client can access all the components and functionality from the engine to build their game.

1. Create index.js file in the src/engine folder; import from gl.js, vertex_buffer.js, and shader_resources.js; and define the init() function to initialize the game engine by calling the corresponding init() functions of the three imported modules:

```

// local to this file only
import * as glSys from "./core/gl.js";
import * as vertexBuffer from "./core/vertex_buffer.js";
import * as shaderResources from "./core/shader_resources.js";

```

CHAPTER 3 DRAWING OBJECTS IN THE WORLD

```
168 // general engine utilities  
169 function init(htmlCanvasID) {  
170     glSys.init(htmlCanvasID);  
171     vertexBuffer.init();  
172     shaderResources.init();  
173 }
```

174 2. Define the `clearCanvas()` function to clear the drawing canvas:

```
175 function clearCanvas(color) {  
176     let gl = glSys.get();  
177     gl.clearColor(color[0], color[1], color[2], color[3]);  
178     gl.clear(gl.COLOR_BUFFER_BIT); // clear to the color set  
179 }
```

180 3. Now, to properly expose the `Renderable` symbol to the clients of
181 the game engine, make sure to import such that the class can be
182 properly exported. The `Renderable` class will be introduced in
183 details in the next section.

```
184 // general utilities  
185 import Renderable from "./renderable.js";
```

186 4. Finally, remember to export the proper symbols and functionality
187 for the clients of the game engine:

```
188 export default {  
189     // Util classes  
190     Renderable,  
191     // functions  
192     init, clearCanvas  
193 }
```

194 With proper maintenance and update of this `index.js` file, the clients of your game
195 engine, the game developers, can simply import from the `index.js` file to gain access
196 to the entire game engine functionality without any knowledge of the source code
197 structure. Lastly, notice that the `glSys`, `vertexBuffer`, and `shaderResources` internal
198 functionality defined in the `engine/src/core` folder are not exported by `index.js` and
199 thus are not accessible to the game developers.

The Renderable Class

At last, you are ready to define the Renderable class to encapsulate the drawing process: 201

1. Define the Renderable class in the game engine by creating a new source code file in the `src/engine` folder, and name the file `renderable.js`. 202
2. Open `renderable.js`, import from `gl.js` and `shader_resources.js`, and define the Renderable class with a constructor to initialize a reference to a shader and a color instance variable. Notice that the shader is a reference to the shared `SimpleShader` instance defined in `shader_resources`. 203

```
import * as glSys from "./core/gl.js"; 210
import * as shaderResources from "./core/shader_resources.js"; 211
```

```
class Renderable { 212
  constructor() { 213
    this.mShader = shaderResources.getConstColorShader(); 214
    this.mColor = [1, 1, 1, 1]; // color of pixel 215
  } 216
  ... implementation to follow ... 217
}
```

3. Define a `draw()` function for `Renderable`: 218

```
draw() { 220
  let gl = glSys.get(); 221
  this.mShader.activate(this.mColor); 222
  gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4); 223
}
```

Notice that it is important to activate the proper GLSL shader in the GPU by calling the `activate()` function before sending the vertices with the `gl.drawArrays()` function. 225

4. Define the getter and setter functions for the color instance variable: 226

```
setColor(color) {this.mColor = color; } 228
getColor() { return this.mColor; } 229
```

- 230 5. Export the Renderable symbol as default to ensure this identifier
231 cannot be renamed:

232 `export default Renderable;`

233 Though this example is simple, it is now possible to create and draw multiple
234 instances of the Renderable objects with different colors.

235 **Testing the Renderable Object**

236 To test Renderable objects in MyGame, white and red instances are created and drawn as
237 follows:

```
238 // import from engine/index.js for all engine symbols
239 import engine from "../engine/index.js";

240 class MyGame {
241     constructor(htmlCanvasID) {
242         // Step A: Initialize the webGL Context
243         engine.init(htmlCanvasID);
244
245         // Step B: Create the Renderable objects:
246         this.mWhiteSq = new engine.Renderable();
247         this.mWhiteSq.setColor([1, 1, 1, 1]);
248         this.mRedSq = new engine.Renderable();
249         this.mRedSq.setColor([1, 0, 0, 1]);
250
251         // Step C: Draw!
252         engine.clearCanvas([0, 0.8, 0, 1]); // Clear the canvas
253
254         // Step C1: Draw Renderable objects with the white shader
255         this.mWhiteSq.draw();
256
257     }
258 }
```

Notice that the <code>import</code> statement is modified to import from the engine access file, <code>index.js</code> . Additionally, the <code>MyGame</code> constructor is modified to include the following steps:	257
1. Step A initializes the engine.	259
2. Step B creates two instances of <code>Renderable</code> and sets the colors of the objects accordingly.	260
3. Step C clears the canvas; steps C1 and C2 simply call the respective <code>draw()</code> functions of the white and red squares.	262
Although both of the squares are drawn, for now, you are only able to see the last of the drawn squares in the canvas. Please refer to the following discussion for the details.	264
	266

Observations

Run the project and you will notice that only the red square is visible! What happens is that both of the squares are drawn to the same location. Being the same size, the two squares simply overlap perfectly. Since the red square is drawn last, it overwrites all the pixels of the white square. You can verify this by commenting out the drawing of the red square (comment out the line `mRedSq.draw()`) and rerunning the project. An interesting observation to make is that objects that appear in the front are drawn last (the red square). You will take advantage of this observation much later when working with transparency.

This simple observation leads to your next task—to allow multiple instances of `Renderable` to be visible at the same time. Each instance of `Renderable` object needs to support the ability to be drawn at different locations, with different sizes and orientations so that they do not overlap one another.

Transforming a Renderable Object

A mechanism is required to manipulate the position, size, and orientation of a `Renderable` object. Over the next few projects, you will learn about how matrix transformations can be used to translate or move an object's position, scale the size of an object, and change the orientation or rotate an object on the canvas. These operations are the most intuitive ones for object manipulations. However, before the implementation of transformation matrices, a quick review of the operations and capabilities of matrices is required.

287

Matrices as Transform Operators

288 Before we begin, it is important to recognize that matrices and transformations are
 289 general topic areas in mathematics. The following discussion does not attempt to
 290 include a comprehensive coverage of these subjects. Instead, the focus is on a small
 291 collection of relevant concepts and operators from the perspective of what the game
 292 engine requires. In this way, the coverage is on how to utilize the operators and not
 293 the theories. If you are interested in the specifics of matrices and how they relate to
 294 computer graphics, please refer to the discussion in Chapter 1 where you can learn
 295 more about these topics by delving into relevant books on linear algebra and computer
 296 graphics.

297 A matrix is an m rows by n columns 2D array of numbers. For the purposes of this
 298 game engine, you will be working exclusively with 4×4 matrices. While a 2D game engine
 299 could get by with 3×3 matrices, a 4×4 matrix is used to support features that will be
 300 introduced in the later chapters. Among the many powerful applications, 4×4 matrices
 301 can be constructed as transform operators for vertex positions. The most important and
 302 intuitive of these operators are the translation, scaling, rotation, and identity operators.

- 303 • The translation operator $T(tx, ty)$, as illustrated in Figure 3-2,
 304 translates or moves a given vertex position from (x, y) to $(x+tx, y+ty)$.
 305 Notice that $T(0, 0)$ does not change the value of a given vertex
 306 position and is a convenient initial value for accumulating translation
 307 operations.

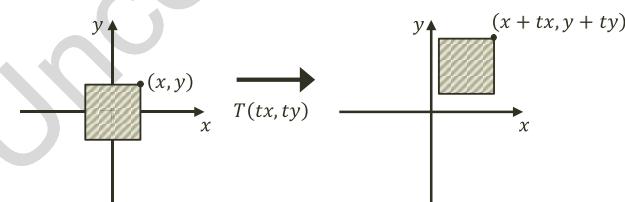


Figure 3-2. Translating a square by $T(tx, ty)$

- 308 • The scaling operator $S(sx, sy)$, as illustrated by Figure 3-3, scales or
 309 resizes a given vertex position from (x, y) to $(x \times sx, y \times sy)$. Notice that
 310 $S(1, 1)$ does not change the value of a given vertex position and is a
 311 convenient initial value for accumulating scaling operations.

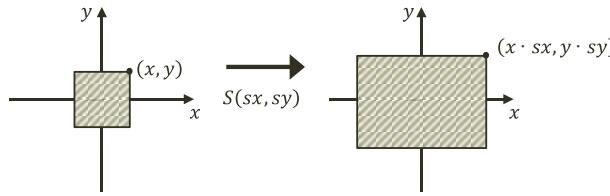


Figure 3-3. Scaling a square by $S(sx, sy)$

- The rotation operator $R(\theta)$, as illustrated in Figure 3-4, rotates a given vertex position with respect to the origin. 312
- 313

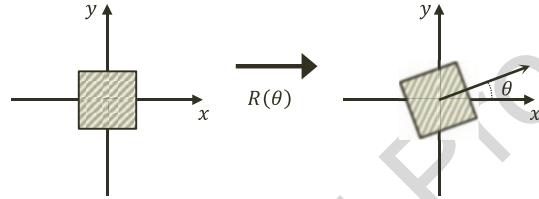


Figure 3-4. Rotating a square by $R(\theta)$

In the case of rotation, $R(0)$ does not change the value of a given vertex and is the convenient initial value for accumulating rotation operations. The values for θ are typically expressed in radians (and not degrees).

- The identity operator I does not affect a given vertex position. This operator is mostly used for initialization.

As an example, a 4×4 identity matrix looks like the following:

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

AU1 Mathematically, a matrix transform operator operates on a vertex through a matrix-vector multiplication. To support this operation, a vertex position $p = (x, y, z)$ must be represented as a 4×1 vector as follows:

$$p = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

314

315 **Note** The z component is the third dimension, or the depth information, of a
 316 vertex position. In most cases, you should leave the z component to be 0.

317 For example, if position p' is the result of a translation operator T operating on the
 318 vertex position p , mathematically, p' would be computed by the following:

$$319 \quad p' = T \times p = Tp$$

320 Concatenation of Matrix Operators

321 Multiple matrix operators can be *concatenated*, or combined, into a single operator while
 322 retaining the same transformation characteristics as the original operators. For example,
 323 you may want to apply the scaling operator S , followed by the rotation operator R , and finally
 324 the translation operator T , on a given vertex position, or to compute p' with the following:

$$325 \quad p' = TRSp$$

326 Alternatively, you can compute a new operator M by concatenating all the transform
 327 operators, as follows:

$$328 \quad M = TRS$$

329 And then operate M on vertex position p , as follows, to produce identical results:

$$330 \quad p' = Mp$$

331 The M operator is a convenient and efficient way to record and reapply the results of
 332 multiple operators.

Finally, notice that when working with transformation operators, the order of operation is important. For example, a scaling operation followed by a translation operation is in general different from a translation followed by a scaling or, in general

$$ST \neq TS$$

333
334
335

336

The glMatrix Library

337

The details of matrix operators and operations are nontrivial to say the least. Developing a complete matrix library is time-consuming and not the focus of this book. Fortunately, there are many well-developed and well-documented matrix libraries available in the public domain. The glMatrix library is one such example. To integrate this library into your source code structure, follow these steps:

1. Create a new folder under the `src` folder, and name the new folder `lib`.
2. Go to <http://glMatrix.net>, as shown in Figure 3-5, and download, unzip, and store the resulting `glMatrix.js` source file into the new `lib` folder.

this figure will be printed in b/w

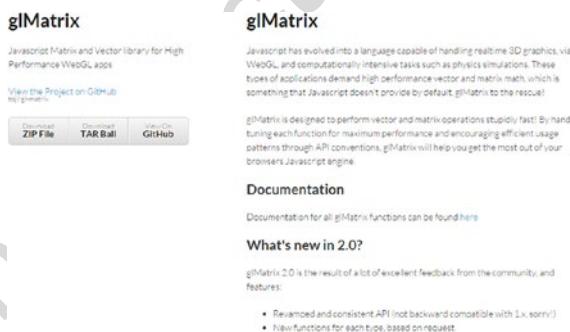


Figure 3-5. Downloading the glMatrix library

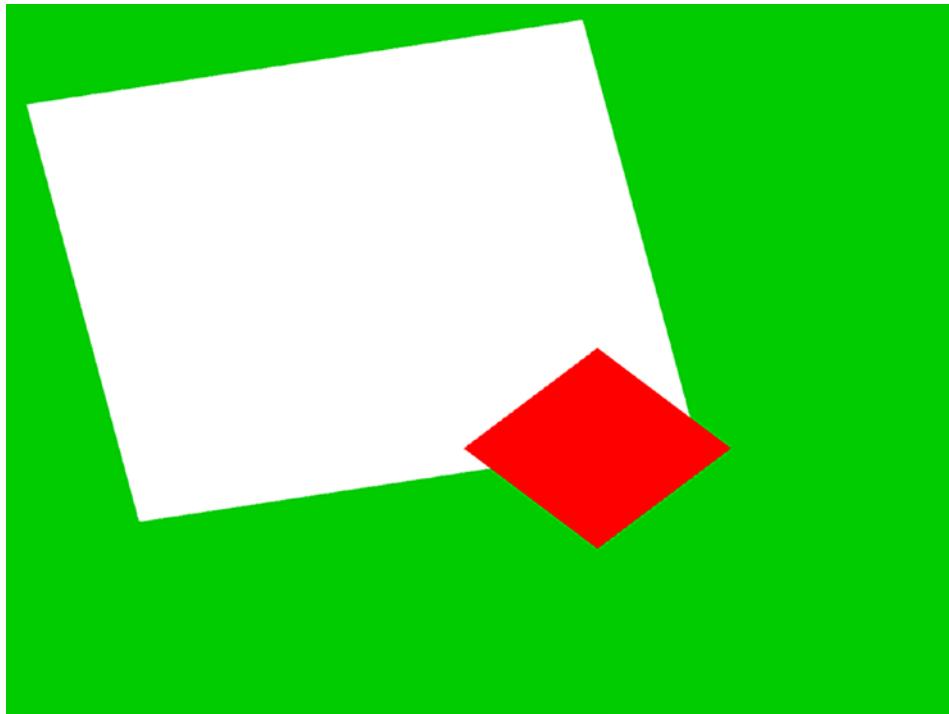
347 All projects in this book are based on `glMatrix` version 2.2.2.

- 348 3. As a library that must be accessible by both the game engine and
349 the client game developer, you will load the source file in the
350 main `index.html` by adding the following before the loading of
351 `my_game.js`:

```
352 <!-- external library -->  
353 <script type="text/javascript" src="src/lib/gl-matrix.js"></script>  
  
354 <!-- our game -->  
355 <script type="module" src=".src/my_game/my_game.js"></script>
```

356 The Matrix Transform Project

357 This project introduces and demonstrates how to use transformation matrices as
358 operators to manipulate the position, size, and orientation of Renderable objects drawn
359 on the canvas. In this way, a Renderable can now be drawn to any location, with any
360 size and any orientation. Figure 3-6 shows the output of running the Matrix Transform
361 project. The source code to this project is defined in the `chapter3/3.2.matrix_`
362 `transform` folder.



this figure will be printed in b/w

Figure 3-6. Running the Matrix Transform project

The goals of the project are as follows:

- To introduce transformation matrices as operators for drawing a Renderable 363
- To understand how to work with the transform operators to 364
- manipulate a Renderable 365
- 366
- 367

Modify the Vertex Shader to Support Transforms

368

As discussed, matrix transform operators operate on vertices of geometries. The vertex shader is where all vertices are passed in from the WebGL context and is the most convenient location to apply the transform operations.

369

370

371

372 You will continue working with the previous project to support the transformation
373 operator in the vertex shader:

- 374 1. Edit `simple_vs.glsl` to declare a uniform 4×4 matrix:

375 **Note** Recall from the discussion in Chapter 2 that `glsl` files contain OpenGL
376 Shading Language (GLSL) instructions that will be loaded to and executed by the
377 GPU. You can find out more about GLSL by referring to the WebGL and OpenGL
378 references provided at the end of Chapter 1.

379 // to transform the vertex position
380 uniform mat4 uModelXformMatrix;

381 Recall that the `uniform` keyword in a GLSL shader declares a variable with values that
382 do not change for all the vertices within that shader. In this case, the `uModelXformMatrix`
383 variable is the transform operator for all the vertices.

384 **Note** GLSL uniform variable names always begin with lowercase “u,” as in
385 `uModelXformMatrix`.

- 386 2. In the `main()` function, apply the `uModelXformMatrix` to the
387 currently referenced vertex position:

388 `gl_Position = uModelXformMatrix * vec4(aVertexPosition, 1.0);`

389 Notice that the operation follows directly from the discussion on matrix
390 transformation operators. The reason for converting `aVertexPosition` to a `vec4` is to
391 support the matrix-vector multiplication.

392 With this simple modification, the vertex positions of the unit square will be
393 operated on by the `uModelXformMatrix` operator, and thus the square can be drawn
394 to different locations. The task now is to set up `SimpleShader` to load the appropriate
395 transformation operator into `uModelXformMatrix`.

Modify SimpleShader to Load the Transform Operator

Follow these steps:

1. Edit simple_shader.js and add an instance variable to hold the reference to the `uModelXformMatrix` matrix in the vertex shader:

```
this.mModelMatrixRef = null;
```

2. At the end of the `SimpleShader` constructor under step E, after setting the reference to `uPixelColor`, add the following code to initialize this reference:

```
// Step E: Gets a reference to uniform variables in fragment shader
this.mPixelColorRef = gl.getUniformLocation(
    this.mCompiledShader, "uPixelColor");
this.mModelMatrixRef = gl.getUniformLocation(
    this.mCompiledShader, "uModelXformMatrix");
```

3. Modify the `activate()` function to receive a second parameter, and load the value to `uModelXformMatrix` via `mModelMatrixRef`:

```
activate(pixelColor, trsMatrix) {
    let gl = glSys.get();
    gl.useProgram(this.mCompiledShader);

    ... identical to previous code ...

    // load uniforms
    gl.uniform4fv(this.mPixelColorRef, pixelColor);
    gl.uniformMatrix4fv(this.mModelMatrixRef, false, trsMatrix);
}
```

The `gl.uniformMatrix4fv()` function copies the values from `trsMatrix` to the vertex shader location identified by `this.mModelMatrixRef` or the `uModelXfromMatrix` operator in the vertex shader. The name of the variable, `trsMatrix`, signifies that it should be a matrix operator containing the concatenated result of translation (T), rotation (R), and scaling (S) or TRS.

424 Modify Renderable Class to Set the Transform Operator

```
425 Edit renderable.js to modify the draw() function to receive and to forward a transform
426 operator to the mShader.activate() function to be loaded to the GLSL shader:
```

```
427 draw(trsMatrix) {
428     let gl = glSys.get();
429     this.mShader.activate(this.mColor, trsMatrix);
430     gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);
431 }
```

432 In this way, when the vertices of the unit square are processed by the vertex shader,
 433 the uModelXformMatrix will contain the proper operator for transforming the vertices
 434 and thus drawing the square at the desired location, size, and rotation.

435 Testing the Transforms

436 Now that the game engine supports transformation, you need to modify the client code
 437 to draw with it:

- 438 1. Edit my_game.js; after step C, instead of activating and drawing
 439 the two squares, replace steps C1 and C2 to create a new identity
 440 transform operator, trsMatrix:

```
441 // create a new identify transform operator
442 let trsMatrix = mat4.create();

443 2. Compute the concatenation of matrices to a single transform
444 operator that implements translation (T), rotation (R), and
445 scaling (S) or TRS:

446 // Step D: compute the white square transform
447 mat4.translate(trsMatrix, trsMatrix, vec3.fromValues(-0.25, 0.25, 0.0));
448 mat4.rotateZ(trsMatrix, trsMatrix, 0.2);      // rotation is in radian
449 mat4.scale(trsMatrix, trsMatrix, vec3.fromValues(1.2, 1.2, 1.0));

450 // Step E: draw the white square with the computed transform
451 this.mWhiteSq.draw(trsMatrix);
```

Step D concatenates $T(-0.25, 0.25)$, moving to the left and up; with $R(0.2)$, rotating clockwise by 0.2 radians; and $S(1.2, 1.2)$, increasing size by 1.2 times. 452
 The concatenation order applies the scaling operator first, followed by rotation, with 453
 translation being the last operation, or $\text{trsMatrix}=TRS$. In step E, the Renderable object 454
 is drawn with the trsMatrix operator or a 1.2×1.2 white rectangle slightly rotated and 455
 located somewhat to the upper left from the center. 456
 457

3. Finally, step F defines the trsMatrix operator that to draw a 458
 0.4×0.4 square that is rotated by 45 degrees and located slightly 459
 toward the lower right from the center of the canvas, and step G 460
 draws the red square: 461

```
// Step F: compute the red square transform 462
mat4.identity(trsMatrix); // restart 463
mat4.translate(trsMatrix, trsMatrix, vec3.fromValues(0.25, -0.25, 0.0)); 464
mat4.rotateZ(trsMatrix, trsMatrix, -0.785); // about -45-degrees 465
mat4.scale(trsMatrix, trsMatrix, vec3.fromValues(0.4, 0.4, 1.0)); 466

// Step G: draw the red square with the computed transform 467
this.mRedSq.draw(trsMatrix); 468
```

Observations

Run the project, and you should see the corresponding white and red rectangles drawn 470 on the canvas. You can gain some intuition of the operators by changing the values; 471
 for example, move and scale the squares to different locations with different sizes. You 472
 can try changing the order of concatenation by moving the corresponding line of code; 473
 for example, move `mat4.scale()` to before `mat4.translate()`. You will notice that, in 474
 general, the transformed results do not correspond to your intuition. In this book, you 475
 will always apply the transformation operators in the fixed TRS order. This ordering of 476
 transformation operators corresponds to typical human intuition. The TRS operation 477
 order is followed by most, if not all, graphical APIs and applications that support 478
 transformation operations. 479

Now that you understand how to work with the matrix transformation operators, it is 480
 time to abstract them and hide their details. 481

482

Encapsulating the Transform Operator

483 In the previous project, the transformation operators were computed directly based on
484 the matrices. While the results were important, the computation involves distracting
485 details and repetitive code. This project guides you to follow good coding practices to
486 encapsulate the transformation operators by hiding the detailed computations with a
487 class. In this way, you can maintain the modularity and accessibility of the game engine
488 by supporting further expansion while maintaining programmability.

489

The Transform Objects Project

490 This project defines the `Transform` class to provide a logical interface for manipulating
491 and hiding the details of working with the matrix transformation operators. Figure 3-7
492 shows the output of running the Matrix Transform project. Notice that the output of this
493 project is identical to that from the previous project. The source code to this project is
494 defined in the `chapter3/3.3.transform_objects` folder.

this figure will be printed in b/w

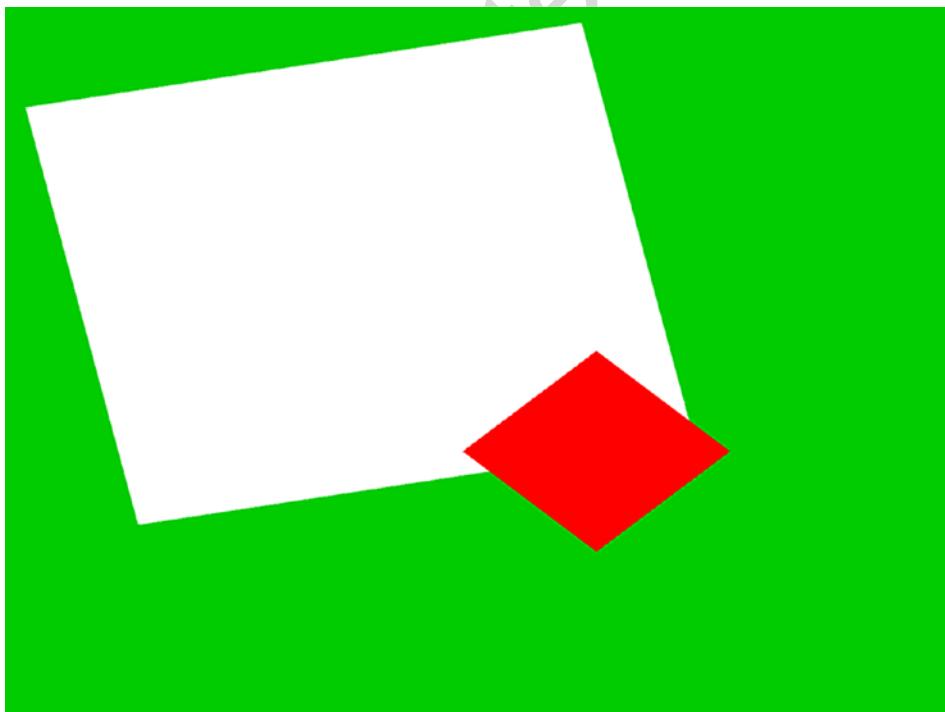


Figure 3-7. Running the Transform Objects project

The goals of the project are as follows:	495
• To create the <code>Transform</code> class to encapsulate the matrix transformation functionality	496
	497
• To integrate the <code>Transform</code> class into the game engine	498
• To demonstrate how to work with <code>Transform</code> objects	499

The Transform Class

Continue working with the previous project: 501

1. Define the `Transform` class in the game engine by creating a new source code file in the `src/engine` folder, and name the file `transform.js`. 502
2. Define the constructor to initialize instance variables that correspond to the operators: `mPosition` for translation, `mScale` for scaling, and `mRotationInRad` for rotation. 503

```
class Transform { 508
    constructor() { 509
        this.mPosition = vec2.fromValues(0, 0); // translation 510
        this.mScale = vec2.fromValues(1, 1); // width (x), height (y) 511
        this.mRotationInRad = 0.0; // in radians! 512
    } 513
    ... implementation to follow ... 514
}
```

3. Add getters and setters for the values of each operator: 516

```
// Position getters and setters 517
setPosition(xPos, yPos) { this.setXPos(xPos); this.setYPos(yPos); } 518
getPosition() { return this.mPosition; } 519
// ... additional get and set functions for position not shown 520
// Size setters and getters 521
setSize(width, height) { 522
    this.setWidth(width); 523
    this.setHeight(height); 524
}
```

CHAPTER 3 DRAWING OBJECTS IN THE WORLD

```
526 getSize() { return this.mScale; }
527 // ... additional get and set functions for size not shown
528 // Rotation getters and setters
529 setRotationInRad(rotationInRadians) {
530     this.mRotationInRad = rotationInRadians;
531     while (this.mRotationInRad > (2 * Math.PI)) {
532         this.mRotationInRad -= (2 * Math.PI);
533     }
534 }
535 setRotationInDegree(rotationInDegree) {
536     this.setRotationInRad(rotationInDegree * Math.PI / 180.0);
537 }
538 // ... additional get and set functions for rotation not shown
```

- 539 4. Define the `getTRSMatrix()` function to compute and return the concatenated transform operator, TRS:

```
541 getTRSMatrix() {
542     // Creates a blank identity matrix
543     let matrix = mat4.create();
544
545     // Step A: compute translation, for now z is always at 0.0
546     mat4.translate(matrix, matrix,
547                   vec3.fromValues(this.getXPos(), this.getYPos(), 0.0));
548
549     // Step B: concatenate with rotation.
550     mat4.rotateZ(matrix, matrix, this.getRotationInRad());
551
552     // Step C: concatenate with scaling
553     mat4.scale(matrix, matrix,
554                vec3.fromValues(this.getWidth(), this.getHeight(), 1.0));
555
556     return matrix;
557 }
```

554 This code is similar to steps D and F in `my_game.js` from the previous project. The
555 concatenated operator TRS performs scaling first, followed by rotation, and lastly by
556 translation.

5. Finally, remember to export the newly defined Transform class:

```
export default Transform;
```

557

558

The Transformable Renderable Class

559

By integrating the Transform class, a Renderable object can now have a position, size (scale), and orientation (rotation). This integration can be easily accomplished through the following steps:

1. Edit renderable.js and add a new instance variable to reference a Transform object in the constructor:

```
this.mXform = new Transform(); // transform operator for the object
```

560

2. Define an accessor for the transform operator:

```
getXform() { return this.mXform; }
```

561

3. Modify the draw() function to pass the trsMatrix operator of the mXform object to activate the shader before drawing the unit square:

```
draw() {
  let gl = glSys.get();
  this.mShader.activate(this.mColor, this.mXform.getTRSMatrix());
  gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);
}
```

562

563

564

With this simple modification, Renderable objects will be drawn with characteristics defined by the values of its own transformation operators.

565

566

Modify the Engine Access File to Export Transform

567

It is important to maintain the engine access file, index.js, up to date such that the newly defined Transform class can be accessed by the game developer:

568

569

570

571

572

573

574

575

576

577

578

579

580

1. Edit index.js; import from the newly define transform.js file:

```
// general utilities
import Transform from "./transform.js";
import Renderable from "./renderable.js";
```

2. Export Transform for client's access:

```
export default {
  // Util classes
  Transform, Renderable,
  // functions
  init, clearCanvas
}
```

Modify Drawing to Support Transform Object

To test the Transform and the improved Renderable classes, the MyGame constructor can be modified to set the transform operators in each of the Renderable objects accordingly:

```
// Step D: sets the white Renderable object's transform
this.mWhiteSq.getXform().setPosition(-0.25, 0.25);
this.mWhiteSq.getXform().setRotationInRad(0.2); // In Radians
this.mWhiteSq.getXform().setSize(1.2, 1.2);
// Step E: draws the white square (transform behavior in the object)
this.mWhiteSq.draw();

// Step F: sets the red square transform
this.mRedSq.getXform().setXPos(0.25); // alternative to setPosition
this.mRedSq.getXform().setYPos(-0.25); // setX/Y separately
this.mRedSq.getXform().setRotationInDegree(45); // this is in Degree
this.mRedSq.getXform().setWidth(0.4); // alternative to setSize
this.mRedSq.getXform().setHeight(0.4); // set width/height separately
// Step G: draw the red square (transform in the object)
this.mRedSq.draw();
```

Run the project to observe identical output as from the previous project. You can now create and draw a Renderable at any location in the canvas, and the transform operator has now been properly encapsulated.

The Camera Transform and Viewports

When designing and building a video game, the game designers and programmers must be able to focus on the intrinsic logic and presentation. To facilitate these aspects, it is important that the designers and programmers can formulate solutions in a convenient dimension and space.

For example, continuing with the soccer game idea, consider the task of creating a soccer field. How big is the field? What is the unit of measurement? In general, when building a game world, it is often easier to design a solution by referring to the real world. In the real world, soccer fields are around 100 meters long. However, in the game or graphics world, units are arbitrary. So, a simple solution may be to create a field that is 100 units in meters and a coordinate space where the origin is located at the center of the soccer field. In this way, opposing sides of the fields can simply be determined by the sign of the x value, and drawing a player at location (0, 1) would mean drawing the player 1 meter to the right from the center of the soccer field.

A contrasting example would be when building a chess-like board game. It may be more convenient to design the solution based on a unitless $n \times n$ grid with the origin located at the lower-left corner of the board. In this scenario, drawing a piece at location (0, 1) would mean drawing the piece at the location one cell or unit toward the right from the lower-left corner of the board. As will be discussed, the ability to define specific coordinate systems is often accomplished by computing and working with a matrix representing the view from a camera.

In all cases, to support a proper presentation of the game, it is important to allow the programmer to control the drawing of the contents to any location on the canvas. For example, you may want to draw the soccer field and players to one subregion and draw a mini-map into another subregion. These axis-aligned rectangular drawing areas or subregions of the canvas are referred to as viewports.

In this section, you will learn about coordinate systems and how to use the matrix transformation as a tool to define a drawing area that conforms to the fixed ± 1 drawing range of the WebGL.

613 Coordinate Systems and Transformations

614 A 2D coordinate system uniquely identifies every position on a 2D plane. All projects
 615 in this book follow the Cartesian coordinate system where positions are defined
 616 according to perpendicular distances from a reference point known as the *origin*, as
 617 illustrated in Figure 3-8. The perpendicular directions for measuring the distances are
 618 known as the *major axes*. In 2D space, these are the familiar x and y axes.

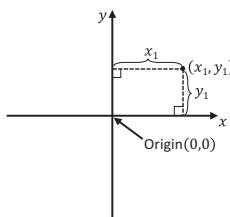


Figure 3-8. Working with a 2D Cartesian coordinate system

619 Modeling and Normalized Device Coordinate Systems

620 So far in this book, you have experience with two distinct coordinate systems. The first
 621 is the coordinate system that defines the vertices for the 1×1 square in the vertex buffer.
 622 This is referred to as the Modeling Coordinate System, which defines the Model Space.
 623 The Model Space is unique for each geometric object, as in the case of the unit square.
 624 The Model Space is defined to describe the geometry of a single model. The second
 625 coordinate system that you have worked with is the one that WebGL draws to, where
 626 the x-/y-axis ranges are bounded to ± 1.0 . This is known as the Normalized Device
 627 Coordinate (NDC) System. As you have experienced, WebGL always draws to the NDC
 628 space and that the contents in the ± 1.0 range cover all the pixels in the canvas.

629 The Modeling transform, typically defined by a matrix transformation operator, is
 630 the operation that transforms geometries from its Model Space into another coordinate
 631 space that is convenient for drawing. In the previous project, the `uModelXformMatrix`
 632 variable in `simple_vs.glsl` is the Modeling transform. As illustrated in Figure 3-9, in
 633 that case, the Modeling transform transformed the unit square into the WebGL's NDC
 634 space. The rightmost arrow annotated with the *Fixed Mapping* label in Figure 3-9 that
 635 points from *WebGL NDC* to *Canvas Coordinates* signifies that WebGL always displays the
 636 entire content of the NDC space in the canvas.

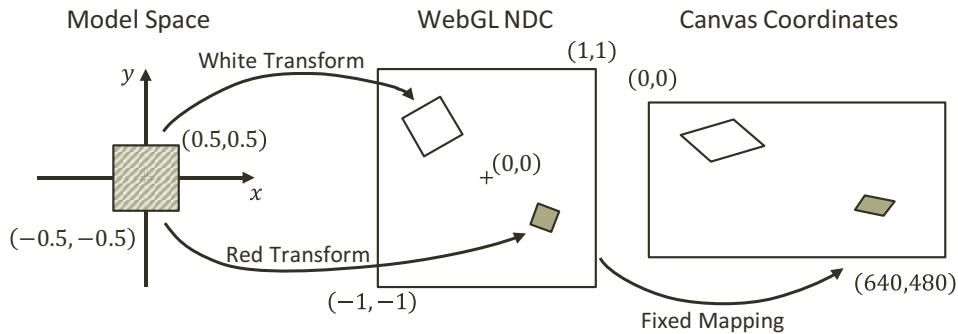


Figure 3-9. Transforming the square from Model to NDC space

this figure will be printed in b/w

The World Coordinate System

637
638
639
640
641P
642
643
644

Although it is possible to draw to any location with the Modeling transform, the disproportional scaling that draws squares as rectangles is still a problem. In addition, the fixed -1.0 and 1.0 NDC space is not a convenient coordinate space for designing games. The World Coordinate (WC) System describes a convenient World Space that resolves these issues. For convenience and readability, in the rest of this book, WC will also be used to refer to the World Space that is defined by a specific World Coordinate System.

645
646
647
648
649
650

As illustrated in Figure 3-10, with a WC instead of the fixed NDC space, Modeling transforms can transform models into a convenient coordinate system that lends itself to game designs. For the soccer game example, the World Space dimension can be the size of the soccer field. As in any Cartesian coordinate system, the WC system is defined by a reference position and its width and height. The reference position can either be the lower-left corner or the center of the WC.

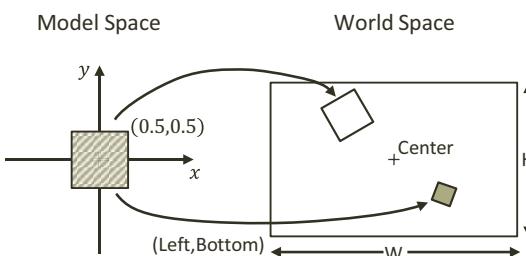


Figure 3-10. Working with a World Coordinate (WC) System

this figure will be printed in b/w

651 The WC is a convenient coordinate system for designing games. However, it is not
 652 the space that WebGL draws to. For this reason, it is important to transform from WC to
 653 NDC. In this book, this transform is referred to as the Camera transform. To accomplish
 654 this transform, you will have to construct an operator to align WC center with that of the
 655 NDC (the origin) and then to scale the WC WxH dimension to match the NDC width
 656 and height. Note that the NDC space has a constant range of -1 to +1 and thus a fixed
 657 dimension of 2x2. In this way, the Camera transform is simply a translation followed by a
 658 scaling operation:

$$659 \quad M = S\left(\frac{2}{W}, \frac{2}{H}\right)T(-center.x, -center.y)$$

660 In this case, (*center.x*, *center.y*) and *WxH* are the center and the dimension of the
 661 WC system.

662 The Viewport

663 A viewport is an area to be drawn to. As you have experienced, by default, WebGL
 664 defines the entire canvas to be the viewport for drawing. Conveniently, WebGL provides
 665 a function to override this default behavior:

```
666        gl.viewport(  

  667            x,        // x position of bottom-left corner of the area to be drawn  

  668            y,        // y position of bottom-left corner of the area to be drawn  

  669            width,    // width of the area to be drawn  

  670            height) // height of the area to be drawn  

  671        );
```

672 The `gl.viewport()` function defines a viewport for all subsequent drawings.

673 Figure 3-11 illustrates the Camera transform and drawing with a viewport.

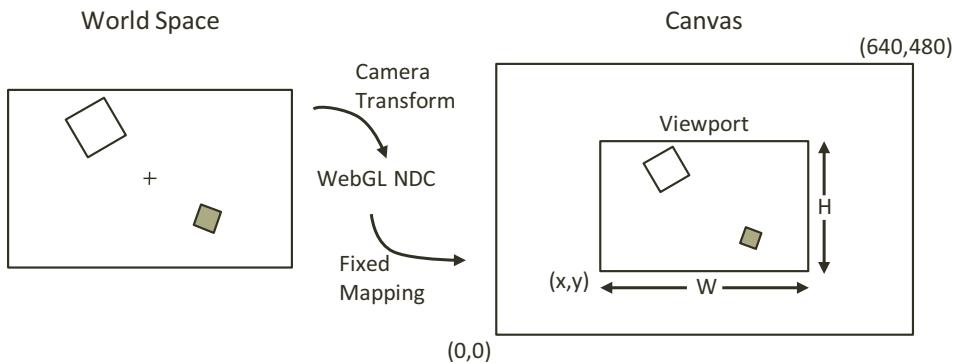


Figure 3-11. Working with the WebGL viewport

The Camera Transform and Viewport Project

this figure will be printed in b/w

674
675
676
677
678

This project demonstrates how to use the Camera transform to draw from any desired coordinate location to any subregion of the canvas or a viewport. Figure 3-12 shows the output of running the Camera Transform and Viewport project. The source code to this project is defined in the chapter3/3.4.camera_transform_and_viewport folder.

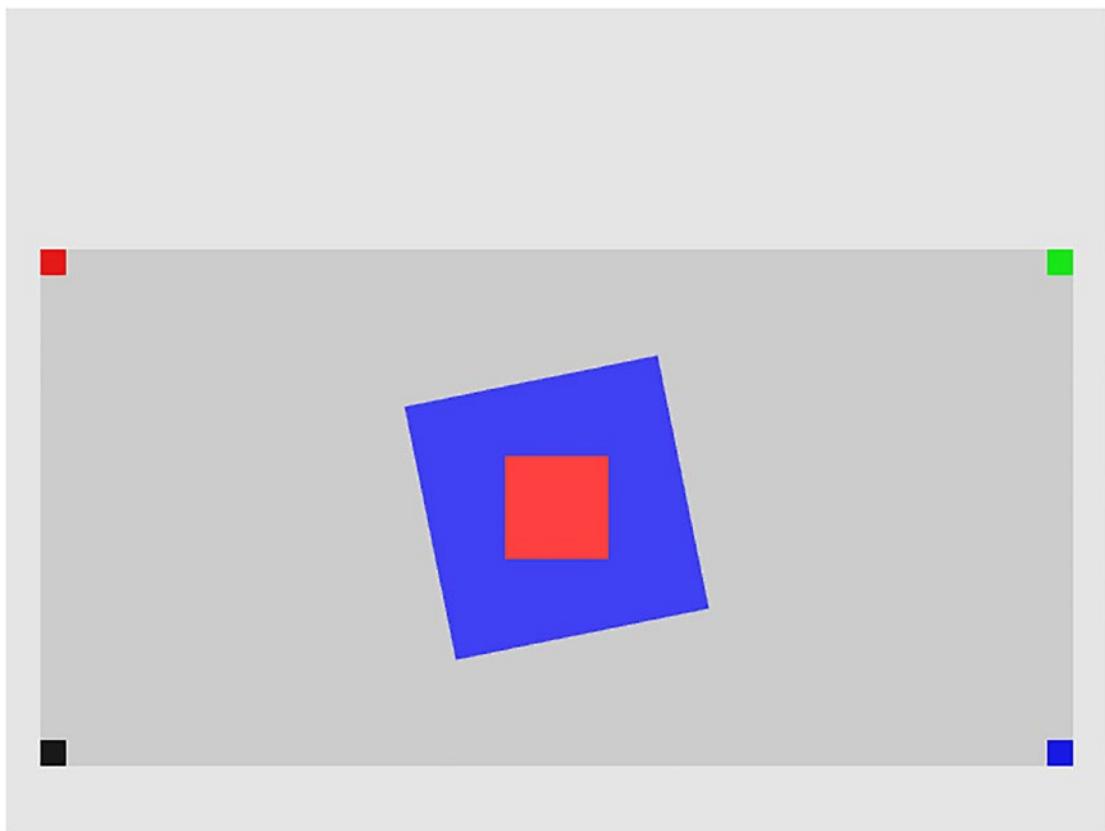


Figure 3-12. Running the Camera Transform and Viewport project

679 The goals of the project are as follows:

- 680 • To understand the different coordinate systems
681 • To experience working with a WebGL viewport to define and draw to
682 different subregions within the canvas
683 • To understand the Camera transform
684 • To begin drawing to the user-defined World Coordinate System

685 You are now ready to modify the game engine to support the Camera transform to
686 define your own WC and the corresponding viewport for drawing. The first step is to
687 modify the shaders to support a new transform operator.

Modify the Vertex Shader to Support the Camera Transform

Relatively, minor changes are required to add the support for the Camera transform:

1. Edit `simple_vs.gls1` to add a new uniform matrix operator to represent the Camera transform:

```
uniform mat4 uCameraXformMatrix;
```

2. Make sure to apply the operator on the vertex positions in the vertex shader program:

```
gl_Position = uCameraXformMatrix *
    uModelXformMatrix *
    vec4(aVertexPosition, 1.0)
```

Recall that the order of matrix operations is important. In this case, the `uModelXformMatrix` first transforms the vertex positions from Model Space to WC, and then the `uCameraXformMatrix` transforms from WC to NDC. The order of `uModelXformMatrix` and `uCameraXformMatrix` cannot be switched.

Modify SimpleShader to Support the Camera Transform

The `SimpleShader` object must be modified to access and pass the Camera transform matrix to the vertex shader:

1. Edit `simple_shader.js` and, in the constructor, add an instance variable for storing the reference to the Camera transform operator in `simple_vs.gls1`:

```
this.mCameraMatrixRef = null;
```

2. At the end of the `SimpleShader` constructor, retrieve the reference to the Camera transform operator, `uCameraXformMatrix`, after retrieving those for the `uModelXformMatrix` and `uPixelColor`:

```
// Step E: Gets reference to uniform variables in fragment shader
this.mPixelColorRef = gl.getUniformLocation(
    this.mCompiledShader, "uPixelColor");
```

```

715 this.mModelMatrixRef = gl.getUniformLocation(
716                         this.mCompiledShader, "uModelXformMatrix");
717 this.mCameraMatrixRef = gl.getUniformLocation(
718                         this.mCompiledShader, "uCameraXformMatrix");

719     3. Modify the activate function to receive a Camera matrix and pass
720         it to the shader:

721 activate(pixelColor, trsMatrix, cameraMatrix) {
722     let gl = glSys.get();
723     gl.useProgram(this.mCompiledShader);
724
725     ... identical to previous code ...
726
727     // load uniforms
728     gl.uniform4fv(this.mPixelColorRef, pixelColor);
729     gl.uniformMatrix4fv(this.mModelMatrixRef, false, trsMatrix);
730     gl.uniformMatrix4fv(this.mCameraMatrixRef, false, cameraMatrix);
731 }

```

As you have seen previously, the `gl.uniformMatrix4fv()` function copies the content of `cameraMatrix` to the `uCameraXformMatrix` operator.

732 **Modify Renderable to Support the Camera Transform**

733 Recall that shaders are activated in the `draw()` function of the `Renderable` class; as such,
734 `Renderable` must also be modified to receive and pass `cameraMatrix` to activate the
735 shader:

```

736 draw(cameraMatrix) {
737     let gl = glSys.get();
738     this.mShader.activate(this.mColor,
739                           this.mXform.getTRSMatrix(), cameraMatrix);
740     gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);
741 }

```

742 It is now possible to set up a WC for drawing and define a subarea in the
743 canvas to draw to.

Design the Scene

744

As illustrated in Figure 3-13, for testing purposes, a World Space (WC) will be defined to be centered at (20, 60) with a dimension of 20×10 . Two rotated squares, a 5x5 blue square and a 2x2 red square, will be drawn at the center of the WC. To verify the coordinate bounds, a 1x1 square with a distinct color will be drawn at each of the WC corners.

this figure will be printed in b/w

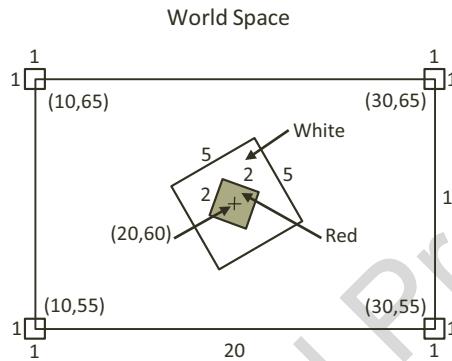


Figure 3-13. Designing a WC to support drawing

As illustrated in Figure 3-14, the WC will be drawn into a viewport with the lower-left corner located at (20, 40) and a dimension of 600×300 pixels. It is important to note that in order for squares to show up proportionally, the width-to-height aspect ratio of the WC must match that of the viewport. In this case, the WC has a 20:10 aspect ratio, and this 2:1 ratio matches that of the 600:300 of the viewport.

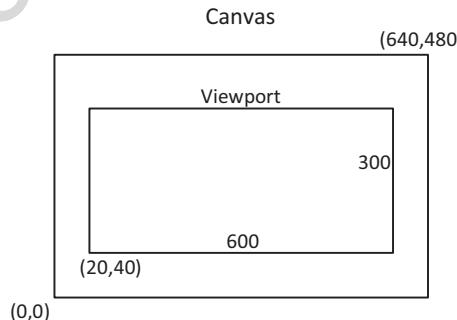
749
750
751
752
753

Figure 3-14. Drawing the WC to the viewport

this figure will be printed in b/w

754 Note that the details of the WC, centered at (20, 60) with dimension 20x10, and the
755 viewport, lower-left corner at (20, 40) and dimension of 600x300, are chosen rather
756 randomly. These are simply reasonable values that can demonstrate the correctness of
757 the implementation.

758 **Implement the Design**

759 The MyGame class will be modified to implement the design:

- 760 1. Edit `my_game.js`. In the constructor, perform step A to initialize
761 the game engine and step B to create six Renderable objects (two
762 to be drawn at the center, with four at each corner of the WC) with
763 corresponding colors.

```
764 constructor(htmlCanvasID) {  
765     // Step A: Initialize the game engine  
766     engine.init(htmlCanvasID);  
  
767     // Step B: Create the Renderable objects:  
768     this.mBlueSq = new engine.Renderable();  
769     this.mBlueSq.setColor([0.25, 0.25, 0.95, 1]);  
770     this.mRedSq = new engine.Renderable();  
771     this.mRedSq.setColor([1, 0.25, 0.25, 1]);  
772     this.mTLSq = new engine.Renderable();  
773     this.mTLSq.setColor([0.9, 0.1, 0.1, 1]);  
774     this.mTRSq = new engine.Renderable();  
775     this.mTRSq.setColor([0.1, 0.9, 0.1, 1]);  
776     this.mBRSq = new engine.Renderable();  
777     this.mBRSq.setColor([0.1, 0.1, 0.9, 1]);  
778     this.mBLSq = new engine.Renderable();  
779     this.mBLSq.setColor([0.1, 0.1, 0.1, 1]);  
780     ... implementation to follow ...  
781 }
```

- 782 2. Steps C and D clear the entire canvas, set up the viewport, and
783 clear the viewport to a different color:

```
784 // Step C: Clear the entire canvas first  
785 engine.clearCanvas([0.9, 0.9, 0.9, 1]);
```

```

// get access to the gl connection to the GPU 786
let gl = glSys.get(); 787

// Step D: Setting up Viewport 788
// Step D1: Set up the viewport: area on canvas to be drawn 789
gl.viewport( 790
    20, // x position of bottom-left corner of the area to be drawn 791
    40, // y position of bottom-left corner of the area to be drawn 792
    600, // width of the area to be drawn 793
    300); // height of the area to be drawn 794

// Step D2: set up the corresponding scissor area to limit clear area 795
gl.scissor( 796
    20, // x position of bottom-left corner of the area to be drawn 797
    40, // y position of bottom-left corner of the area to be drawn 798
    600, // width of the area to be drawn 799
    300); // height of the area to be drawn 800

// Step D3: enable scissor area, clear and then disable the scissor area 801
gl.enable(gl.SCISSOR_TEST); 802
engine.clearCanvas([0.8, 0.8, 0.8, 1.0]); // clear the scissor area 803
gl.disable(gl.SCISSOR_TEST); 804

```

Step D1 defines the viewport, and step D2 defines a corresponding scissor area. The scissor area tests and limits the area to be cleared. Since the testing involved in `gl.scissor()` is computationally expensive, it is disabled immediately after use.

3. Step E defines the WC with the Camera transform by concatenating the proper scaling and translation operators:

```

// Step E: Set up camera transform matrix 810
// assume camera position and dimension 811
let cameraCenter = vec2.fromValues(20, 60); 812
let wcSize = vec2.fromValues(20, 10); 813
let cameraMatrix = mat4.create(); 814

// Step E1: after translation, scale to: -1 to 1: a 2x2 square at origin 815
mat4.scale(cameraMatrix, mat4.create(), 816
    vec3.fromValues(2.0/wcSize[0], 2.0/wcSize[1], 1.0)); 817

```

CHAPTER 3 DRAWING OBJECTS IN THE WORLD

```
818 // Step E2: first to perform is to translate camera center to origin  
819 mat4.translate(cameraMatrix, cameraMatrix,  
820                 vec3.fromValues(-cameraCenter[0], -cameraCenter[1], 0));  
  
821     Step E1 defines the scaling operator,  $S(2/W, 2/H)$ , to scale the WC WxH to the NDC  
822 2x2 dimension, and step E2 defines the translation operator,  $T(-center.x, -center.y)$ ,  
823 to align the WC with the NDC center. Note that the concatenation order implements the  
824 translation first followed by the scaling operator. This is precisely the Camera transform  
825 described earlier that defines the WC as follows:
```

- 826 a. **Center:** (20,60)
- 827 b. **Top-left corner:** (10, 65)
- 828 c. **Top-right corner:** (30, 65)
- 829 d. **Bottom-right corner:** (30, 55)
- 830 e. **Bottom-left corner:** (10, 55)

831 Recall that the order of multiplication is important and that the order of scaling and
832 translation operators cannot be swapped.

- 833 4. Set up the slightly rotated 5x5 blue square at the center of WC, and
834 draw with the Camera transform operator, `cameraMatrix`:

```
835 // Step F: Draw the blue square  
836 // Center Blue, slightly rotated square  
837 this.mBlueSq.getXform().setPosition(20, 60);  
838 this.mBlueSq.getXform().setRotationInRad(0.2); // In Radians  
839 this.mBlueSq.getXform().setSize(5, 5);  
840 this.mBlueSq.draw(cameraMatrix);
```

- 841 5. Now draw the other five squares, first the 2x2 in the center and
842 one each at a corner of the WC:

```
843 // Step G: Draw the center and the corner squares  
844 // center red square  
845 this.mRedSq.getXform().setPosition(20, 60);  
846 this.mRedSq.getXform().setSize(2, 2);  
847 this.mRedSq.draw(cameraMatrix);
```

```

// top left 848
this.mTLSq.getXform().setPosition(10, 65); 849
this.mTLSq.draw(cameraMatrix); 850

// top right 851
this.mTRSq.getXform().setPosition(30, 65); 852
this.mTRSq.draw(cameraMatrix); 853

// bottom right 854
this.mBRSq.getXform().setPosition(30, 55); 855
this.mBRSq.draw(cameraMatrix); 856

// bottom left 857
this.mBLSq.getXform().setPosition(10, 55); 858
this.mBLSq.draw(cameraMatrix); 859

```

Run this project and observe the distinct colors at the four corners: the top left (`mTLSq`) in red, the top right (`mTRSq`) in green, the bottom right (`mBRSq`) in blue, and the bottom left (`mBLSq`) in dark gray. Change the locations of the corner squares to verify that the center positions of these squares are located in the bounds of the WC, and thus, only one quarter of the squares are actually visible. For example, set `mBLSq` to (12, 57) to observe the dark-gray square is actually four times the size. This observation verifies that the areas of the squares outside of the viewport/scissor area are clipped by WebGL.

Although lacking proper abstraction, it is now possible to define any convenient WC system and any rectangular subregions of the canvas for drawing. With the Modeling and Camera transformations, a game programmer can now design a game solution based on the semantic needs of the game and ignore the irrelevant WebGL NDC drawing range. However, the code in the `MyGame` class is complicated and can be distracting. As you have seen so far, the important next step is to define an abstraction to hide the details of Camera transform matrix computation.

The Camera

The Camera transform allows the definition of a WC. In the physical world, this is analogous to taking a photograph with the camera. The center of the viewfinder of your camera is the center of the WC, and the width and height of the world visible through the

878 viewfinder are the dimensions of WC. With this analogy, the act of taking the photograph
879 is equivalent to computing the drawing of each object in the WC. Lastly, the viewport
880 describes the location to display the computed image.

881 **The Camera Objects Project**

882 This project demonstrates how to abstract the Camera transform and the viewport
883 hide the details of matrix computation and WebGL configurations. Figure 3-15 shows
884 the output of running the Camera Objects project; notice the output of this project is
885 identical to that from the previous project. The source code to this project is defined in
886 the chapter3/3.5.camera_objects folder.

this figure will be printed in b/w

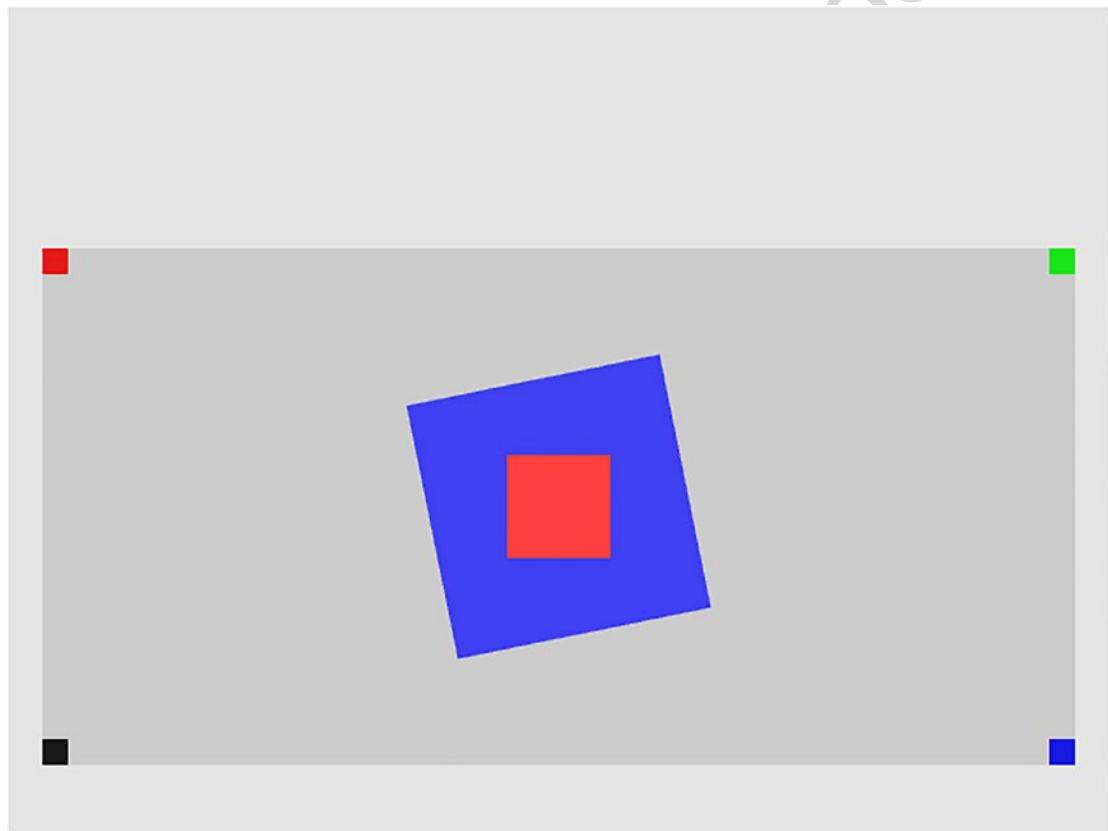


Figure 3-15. Running the Camera Objects project

The goals of the project are as follows:	887
• To define the Camera class to encapsulate the definition of WC and the viewport functionality	888
• To integrate the Camera class into the game engine	889
• To demonstrate how to work with a Camera object	890
The Camera Class	892
The Camera class must encapsulate the functionality defined by the scaling and translation operators in the MyGame constructor from the previous example. A clean and reusable class design should be completed with appropriate getter and setter functions.	893
1. Define the Camera class in the game engine by creating a new source file in the src/engine folder, and name the file camera.js.	896
2. Add the constructor for Camera:	897
class Camera {	899
constructor(wcCenter, wcWidth, viewportArray) {	900
// WC and viewport position and size	901
this.mWCCenter = wcCenter;	902
this.mWCWidth = wcWidth;	903
this.mViewport = viewportArray; // [x, y, width, height]	904
// Camera transform operator	905
this.mCameraMatrix = mat4.create();	906
// background color	907
this.mBGColor = [0.8, 0.8, 0.8, 1]; // RGB and Alpha	908
}	909
... implementation to follow ...	910
}	911

CHAPTER 3 DRAWING OBJECTS IN THE WORLD

912 The Camera defines the WC center and width, the viewport, the Camera transform
913 operator, and a background color. Take note of the following:

- 914 a. The `mWCCenter` is a `vec2` (`vec2` is defined in the `glMatrix` library).
915 It is a float array of two elements. The first element, index position
916 0, of `vec2` is the x, and the second element, index position 1, is the
917 y position.
- 918 b. The four elements of the `viewportArray` are the x and y positions
919 of the lower-left corner and the width and height of the viewport,
920 in that order. This compact representation of the viewport keeps
921 the number of instance variables to a minimum and helps keep
922 the Camera class manageable.
- 923 c. The `mWCWidth` is the width of the WC. To guarantee a matching
924 aspect ratio between WC and the viewport, the height of the WC
925 is always computed from the aspect ratio of the viewport and
926 `mWCWidth`.
- 927 d. `mBgColor` is an array of four floats representing the red, green,
928 blue, and alpha components of a color.
- 929 3. Outside of the Camera class definition, define enumerated indices
930 for accessing the `viewportArray`:

```
931 const eViewport = Object.freeze({  
932     eOrgX: 0,  
933     eOrgY: 1,  
934     eWidth: 2,  
935     eHeight: 3  
936 });
```

937 **Note** Enumerated elements have names that begin with lowercase “e,” as in
938 `eViewport` and `eOrgX`.

4. Define the function to compute the WC height based on the aspect ratio of the viewport:

```
getWCHeight() { 941
    // viewportH/viewportW 942
    let ratio = this.mViewport[eViewport.eHeight] / 943
        this.mViewport[eViewport.eWidth]; 944
    return this.getWCWidth() * ratio; 945
}
} 946
```

5. Add getters and setters for the instance variables:

```
setWCCenter(xPos, yPos) { 948
    this.mWCCenter[0] = xPos; 949
    this.mWCCenter[1] = yPos; 950
}
getWCCenter() { return this.mWCCenter; } 952
setWCWidth(width) { this.mWCWidth = width; } 953
setViewport(viewportArray) { this.mViewport = viewportArray; } 954
getViewport() { return this.mViewport; } 955
setBackgroundColor(newColor) { this.mBGColor = newColor; } 956
getBackgroundColor() { return this.mBGColor; } 957
```

6. Create a function to set the viewport and compute the Camera transform operator for this Camera:

```
// Initializes the camera to begin drawing 960
setViewAndCameraMatrix() { 961
    let gl = glSys.get(); 962
    // Step A: Configure the viewport 963
    ... implementation to follow ...
    // Step B: compute the Camera Matrix 965
    ... implementation to follow ...
}
} 967
```

CHAPTER 3 DRAWING OBJECTS IN THE WORLD

968 Note that this function is called `setViewAndCameraMatrix()` because it configures
969 WebGL to draw to the desire viewport and sets up the Camera transform operator. The
970 following explains the details of steps A and B.

971 7. The code to configure the viewport under step A is as follows:

```
972 // Step A1: Set up the viewport: area on canvas to be drawn
973 gl.viewport(this.mViewport[0], // x of bottom-left of area to be drawn
974             this.mViewport[1], // y of bottom-left of area to be drawn
975             this.mViewport[2], // width of the area to be drawn
976             this.mViewport[3]); // height of the area to be drawn
977 // Step A2: set up the corresponding scissor area to limit the clear area
978 gl.scissor(this.mViewport[0], // x of bottom-left of area to be drawn
979             this.mViewport[1], // y of bottom-left of area to be drawn
980             this.mViewport[2], // width of the area to be drawn
981             this.mViewport[3]); // height of the area to be drawn
982 // Step A3: set the color to be clear
983 gl.clearColor(this.mBGColor[0], this.mBGColor[1],
984                 this.mBGColor[2], this.mBGColor[3]);
985 // set the color to be cleared
986 // Step A4: enable scissor area, clear and then disable the scissor area
987 gl.enable(gl.SCISSOR_TEST);
988 gl.clear(gl.COLOR_BUFFER_BIT);
989 gl.disable(gl.SCISSOR_TEST);
```

990 Notice the similarity of these steps to the viewport setup code in `MyGame` of the previous
991 example. The only difference is the proper references to the instance variables via `this`.

992 8. The code to set up the Camera transform operator under step B is
993 as follows:

```
994 // Step B: Compute the Camera Matrix
995 let center = this.getWCCenter();
996 // Step B1: after translation, scale to -1 to 1: 2x2 square at origin
997 mat4.scale(this.mCameraMatrix, mat4.create(),
998             vec3.fromValues(2.0 / this.getWCWidth(),
999                             2.0 / this.getWCHeight(), 1.0));
```

```
// Step B2: first translate camera center to the origin          1000
mat4.translate(this.mCameraMatrix, this.mCameraMatrix,           1001
    vec3.fromValues(-center[0], -center[1], 0));                1002
```

Once again, this code is similar to the MyGame constructor from the previous example. 1003

9. Define a function to access the computed camera matrix: 1004

```
getCameraMatrix() { return this.mCameraMatrix; } 1005
```

10. Finally, remember to export the newly defined Camera class: 1006

```
export default Camera. 1007
```

Modify Renderable to Support the Camera Class 1008

The draw() function of the Renderable class must be modified to receive the newly 1009
defined Camera in order to access the computed camera matrix: 1010

```
draw(camera) { 1011
    let gl = glSys.get(); 1012
    this.mShader.activate(this.mColor, this.mXform.getTRSMatrix(), 1013
        camera.getCameraMatrix()); 1014
    gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4); 1015
} 1016
```

Modify the Engine Access File to Export Camera 1017

It is important to maintain the engine access file, index.js, up to date such that the 1018
newly defined Camera class can be accessed by the game developer: 1019

1. Edit index.js; import from the newly define camera.js file: 1020

```
// general utilities 1021
import Camera from "./camera.js"; 1022
import Transform from "./transform.js"; 1023
import Renderable from "./renderable.js"; 1024
```

```

1025      2. Export Camera for client's access:
1026 export default {
1027   // Util classes
1028   Camera, Transform, Renderable,
1029   // functions
1030   init, clearCanvas
1031 }
```

Test the Camera

1032 With the Camera class properly defined, testing it from `my_game.js` is straightforward:

1034 1. Edit `my_game.js`; after the initialization of the game engine in step
 1035 A, create an instance of the Camera object with settings that define
 1036 the WC and viewport from the previous project in step B:

```

1037 class MyGame {
1038   constructor(htmlCanvasID) {
1039     // Step A: Initialize the game engine
1040     engine.init(htmlCanvasID);
1041
1042     // Step B: Setup the camera
1043     this.mCamera = new engine.Camera(
1044       vec2.fromValues(20, 60),    // center of the WC
1045       20,                      // width of WC
1046       [20, 40, 600, 300]        // viewport:orgX, orgY, W, H
1047     );
1048   ...
1049 }
```

1049 2. Continue with the creation of the six Renderable objects and the
 1050 clearing of the canvas in steps C and D:

```

1051 // Step C: Create the Renderable objects:
1052 this.mBlueSq = new engine.Renderable();
1053 this.mBlueSq.setColor([0.25, 0.25, 0.95, 1]);
1054 this.mRedSq = new engine.Renderable();
1055 this.mRedSq.setColor([1, 0.25, 0.25, 1]);
```

```

this.mTLSq = new engine.Renderable();                                1056
this.mTLSq.setColor([0.9, 0.1, 0.1, 1]);                            1057
this.mTRSq = new engine.Renderable();                                1058
this.mTRSq.setColor([0.1, 0.9, 0.1, 1]);                            1059
this.mBRSq = new engine.Renderable();                                1060
this.mBRSq.setColor([0.1, 0.1, 0.9, 1]);                            1061
this.mBLSq = new engine.Renderable();                                1062
this.mBLSq.setColor([0.1, 0.1, 0.1, 1]);                            1063

// Step D: Clear the canvas                                         1064
engine.clearCanvas([0.9, 0.9, 0.9, 1]);           // Clear the canvas 1065

3. Now, call the setViewAndCameraMatrix() function of the Camera    1066
object in to configure the WebGL viewport and compute the        1067
camera matrix in step E, and draw all the Renderables using the   1068
Camera object in steps F and G.                                     1069

// Step E: Starts the drawing by activating the camera             1070
this.mCamera.setViewAndCameraMatrix();                           1071

// Step F: Draw the blue square                                    1072
// Center Blue, slightly rotated square                         1073
this.mBlueSq.getXform().setPosition(20, 60);                   1074
this.mBlueSq.getXform().setRotationInRad(0.2); // In Radians     1075
this.mBlueSq.getXform().setSize(5, 5);                          1076
this.mBlueSq.draw(this.mCamera);                                1077

// Step G: Draw the center and the corner squares                1078
// center red square                                           1079
this.mRedSq.getXform().setPosition(20, 60);                   1080
this.mRedSq.getXform().setSize(2, 2);                          1081
this.mRedSq.draw(this.mCamera);                                1082

// top left
this.mTLSq.getXform().setPosition(10, 65);                   1083
this.mTLSq.draw(this.mCamera);                                1084

```

```
1086 // top right  
1087 this.mTRSq.getXform().setPosition(30, 65);  
1088 this.mTRSq.draw(this.mCamera);  
  
1089 // bottom right  
1090 this.mBRSq.getXform().setPosition(30, 55);  
1091 this.mBRSq.draw(this.mCamera);  
  
1092 // bottom left  
1093 this.mBLSq.getXform().setPosition(10, 55);  
1094 this.mBLSq.draw(this.mCamera);
```

1095 The `mCamera` object is passed to the `draw()` function of the `Renderable` objects such
1096 that the `Camera` matrix operator can be retrieved and used to activate the shader.

1097 Summary

1098 In this chapter, you learned how to create a system that can support the drawing of many
1099 objects. The system is composed of three parts: the objects, the details of each object,
1100 and the display of the objects on the browser's canvas. The objects are encapsulated
1101 by the `Renderable`, which uses a `Transform` to capture its details—the position, size,
1102 and rotation. The particulars of displaying the objects are defined by the `Camera`, where
1103 objects at specific locations can be displayed at desirable subregions on the canvas.

1104 You also learned that objects are all drawn relative to a `World Space` or `WC`, a
1105 convenient coordinate system. A `WC` is defined for scene compositions based on
1106 coordinate transformations. Lastly, the `Camera` transform is used to select which portion
1107 of the `WC` to actually display on the canvas within a browser. This can be achieved by
1108 defining an area that is viewable by the `Camera` and using the `viewport` functionality
1109 provided by `WebGL`.

1110 As you built the drawing system, the game engine source code structure has been
1111 consistently refactored into abstracted and encapsulated components. In this way, the
1112 source code structure continues to support further expansion including additional
1113 functionality which will be discussed in the next chapter.

Author Queries

Chapter No.: 3 0005244096

Queries	Details Required	Author's Response
AU1	Both forms “matrix transform” and “Matrix Transform” have been used in the text and retained as given. Please check if okay.	
AU2	Please check if edit to sentence starting “The World Coordinate...” is okay.	
AU3	Both forms “Camera” and “camera” have been used in the text and retained as given. Please check if okay.	

Uncorrected Proof

CHAPTER 4

1

Implementing Common Components of Video Games

2

3

4

After completing this chapter, you will be able to

- Control the position, size, and rotation of Renderable objects to construct complex movements and animations
- Receive keyboard input from the player to control and animate Renderable objects
- Work with asynchronous loading and unloading of external assets
- Define, load, and execute a simple game level from a scene file
- Change game levels by loading a new scene
- Work with sound clips for background music and audio cues

5

6

7

8

9

10

11

12

13

Introduction

14

In the previous chapters, a skeletal game engine was constructed to support basic drawing operations. Drawing is the first step to constructing your game engine because it allows you to observe the output while continuing to expand the game engine functionality. In this chapter, the two important mechanisms, interactivity and resource support, will be examined and added to the game engine. Interactivity allows the engine to receive and interpret player input, while resource support refers to the functionality of working with external files like the GLSL shader source code files, audio clips, and images.

15

16

17

18

19

20

21

This chapter begins by introducing you to the game loop, a critical component that creates the sensation of real-time interaction and immediacy in nearly all video games. Based on the game loop foundation, player keyboard input will be supported via integrating the corresponding HTML5 functionality. A resource management infrastructure will be constructed from the ground up to support the efficient loading, storing, retrieving, and utilization of external files. Functionality for working with external text files (e.g., the GLSL shader source code files) and audio clips will be integrated with corresponding example projects. Additionally, game scene architecture will be derived to support the ability to work with multiple scenes and scene transitions, including scenes that are defined in external scene files. By the end of this chapter, your game engine will support player interaction via the keyboard, have the ability to provide audio feedback, and be able to transition between distinct game levels including loading a level from an external file.

The Game Loop

One of the most basic operations of any video game is the support of seemingly instantaneous interactions between the players' input and the graphical gaming elements. In reality, these interactions are implemented as a continuous running loop that receives and processes player input, updates the game state, and renders the game. This constantly running loop is referred to as the *game loop*.

To convey the proper sense of instantaneity, each cycle of the game loop must be completed within an average person's reaction time. This is often referred to as real time, which is the amount of time that is too short for humans to detect visually. Typically, real time can be achieved when the game loop is running at a rate of higher than 40–60 cycles in a second. Since there is usually one drawing operation in each game loop cycle, the rate of this cycle is also referred to as frames per second (FPS) or the frame rate. An FPS of 60 is a good target for performance. This is to say, your game engine must receive player input, update the game world, and then draw the game world all within 1/60th of a second!

The game loop itself, including the implementation details, is the most fundamental control structure for a game. With the main goal of maintaining real-time performance, the details of a game loop's operation are of no concern to the rest of the game engine. For this reason, the implementation of a game loop should be tightly encapsulated in the core of the game engine with its detailed operations hidden from other gaming elements.

Typical Game Loop Implementations

55

A game loop is the mechanism through which logic and drawing are continuously executed. A simple game loop consists of drawing all objects, processing the player input, and updating the state of those objects, as illustrated in the following pseudocode:

```
initialize();  
while(game running) {  
    draw();  
    input();  
    update();  
}  
}
```

56

57

58

59

60

61

62

63

64

As discussed, an FPS of 60 is required to maintain the sense of real-time interactivity. When the game complexity increases, one problem that may arise is when sometimes a single loop can take longer than 1/60th of a second to complete, causing the game to run at a reduced frame rate. When this happens, the entire game will appear to slow down. A common solution is to prioritize some operations over others. That is, the engine can be designed in such a way as to fixate the game loop on completing operations that the engine deems more vital while skipping others. Since correct input and updates are required for a game to function as designed, it is often the draw operation that is skipped when necessary. This is referred to as frame skipping, and the following pseudocode illustrates one such implementation:

```
elapsedTime = now;  
previousLoop = now;  
while(game running) {  
    elapsedTime += now - previousLoop;  
    previousLoop = now;  
  
    draw();  
    input();  
    while( elapsedTime >= UPDATE_TIME_RATE ) {  
        update();  
        elapsedTime -= UPDATE_TIME_RATE;  
    }  
}
```

75

76

77

78

79

80

81

82

83

84

85

86

In the previous pseudocode listing, UPDATE_TIME_RATE is the required real-time update rate. When the elapsed time between the game loop cycle is greater than the UPDATE_TIME_RATE, update() will be called until it has caught up. This means that the draw() operation is essentially skipped when the game loop is running too slowly. When this happens, the entire game will appear to run slowly, with lagging gameplay input responses and skipped frames. However, the game logic will continue to function correctly.

Notice that the while loop that encompasses the update() function call simulates a fixed update time step of UPDATE_TIME_RATE. This fixed time step update allows for a straightforward implementation in maintaining a deterministic game state. This is an important component to make sure your game engine functions as expected whether running optimally or slowly.

To ensure the focus is solely on the understanding of the core game loop's draw and update operations, input will be ignored until the next project.

The Game Loop Project

This project demonstrates how to incorporate a game loop into your game engine and to support real-time animation by drawing and updating Renderable objects. You can see an example of this project running in Figure 4-1. The source code to this project is defined in the chapter4/4.1.game_loop folder.

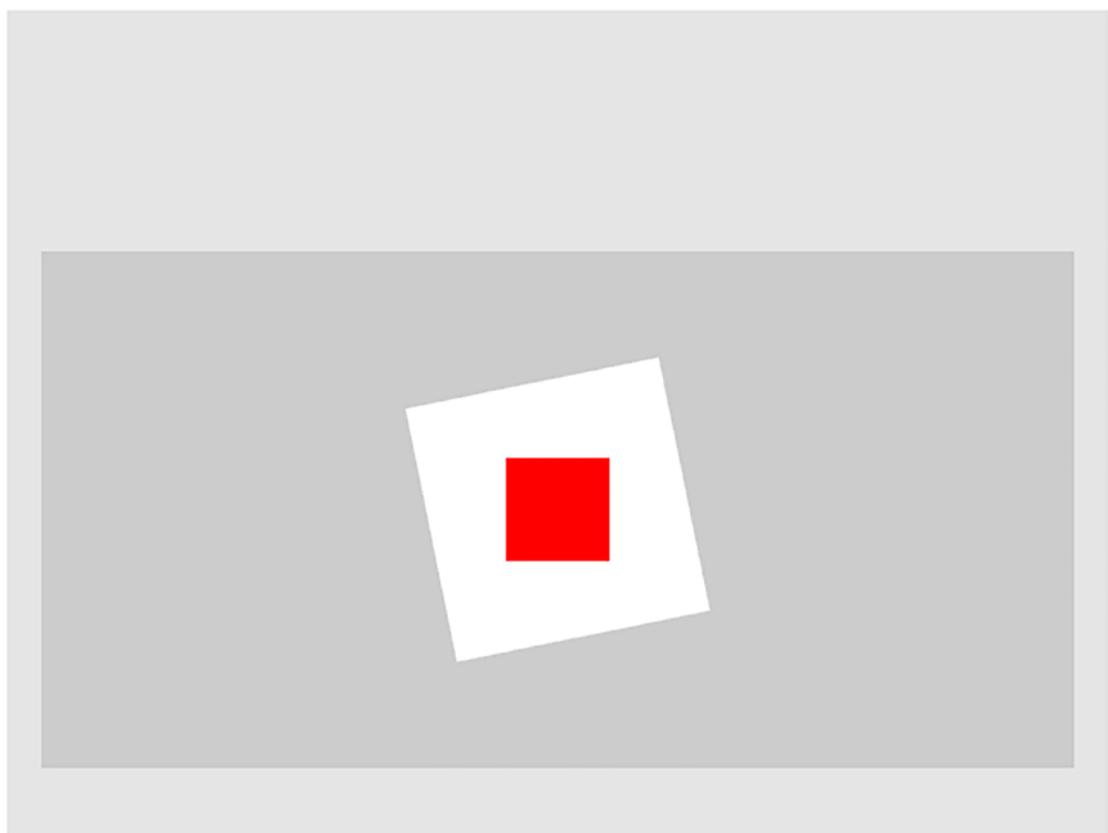


Figure 4-1. Running the Game Loop project

The goals of the project are as follows:

- To understand the internal operations of a game loop 106
- To implement and encapsulate the operations of a game loop 107
- To gain experience with continuous draw and update to create 108
- animation 109
- 110

Implement the Game Loop Component

The game loop component is core to the game engine's functionality and thus should be located similar to that of `vertex_buffer`, as a file defined in the `src/engine/core` folder: 112
113

1. Create a new file for the loop module in the `src/engine/core` 114
folder and name the file `loop.js`. 115

- 116 2. Define the following instance variables to keep track of frame
 117 rate, processing time in milliseconds per frame, the game loop's
 118 current run state, and a reference to the current scene as follows:

```
119 "use strict"
120 const kUPS = 60; // Updates per second
121 const kMPF = 1000 / kUPS; // Milliseconds per update.
122 // Variables for timing gameloop.
123 let mPrevTime;
124 let mLagTime;
125 // The current loop state (running or should stop)
126 let mLoopRunning = false;
127 let mCurrentScene = null;
128 let mFrameID = -1;
```

129 Notice that kUPS is the updates per second similar to the FPS discussed, and it is set
 130 to 60 or 60 updates per second. The time available for each update is simply 1/60 of a
 131 second. Since there are 1000 milliseconds in a second, the available time for each update
 132 in milliseconds is 1000 * (1/60), or kMPF.

133 **Note** When the game is running optimally, frame drawing and updates are both
 134 maintained at the same rate; FPS and kUPS can be thought of interchangeably.
 135 However, when lag occurs, the loop skips frame drawing and prioritizes updates.
 136 In this case, FPS will decrease, while kUPS will be maintained.

- 137 3. Add a function to run the core loop as follows:

```
138 function loopOnce() {
139   if (mLoopRunning) {
140     // Step A: set up for next call to LoopOnce
141     mFrameID = requestAnimationFrame(loopOnce);
142
143     // Step B: now let's draw
144     //           draw() MUST be called before update()
145     //           as update() may stop the loop!
146     mCurrentScene.draw();
```

```

// Step C: compute time elapsed since last loopOnce was executed      146
let currentTime = performance.now();                                147
let elapsedTime = currentTime - mPrevTime;                            148
mPrevTime = currentTime;                                              149
mLagTime += elapsedTime;                                            150

// Step D: update the game the appropriate number of times.          151
//           Update only every Milliseconds per frame.                152
//           If lag larger then update frames, update until caught up. 153
while ((mLagTime >= kMPF) && mLoopRunning) {                         154
    mCurrentScene.update();                                         155
    mLagTime -= kMPF;                                               156
}
}                                                               157
}                                                               158
159

```

Note The `performance.now()` is a JavaScript function that returns a timestamp in milliseconds.

Notice the similarity between the pseudocode examined previously and the steps B, C, and D of the `loopOnce()` function, that is, the drawing of the scene or game in step B, the calculation of the elapsed time since last update in step C, and the prioritization of update if the engine is lagging behind.

The main difference is that the outermost while loop is implemented based on the HTML5 `requestAnimationFrame()` function call at step A. The `requestAnimationFrame()` function will, at an approximated rate of 60 times per second, invoke the function pointer that is passed in as its parameter. In this case, the `loopOnce()` function will be called continuously at approximately 60 times per second. Notice that each call to the `requestAnimationFrame()` function will result in exactly one execution of the corresponding `loopOnce()` function and thus draw only once. However, if the system is lagging, multiple updates can occur during this single frame.

174 **Note** The `requestAnimationFrame()` function is an HTML5 utility provided by
175 the browser that hosts your game. The precise behavior of this function is browser
176 implementation dependent.

177 The `mLoopRunning` condition of the while loop in step D is a redundant check
178 for now. This condition will become important in later sections when `update()`
179 can call `stop()` to stop the loop (e.g., for level transitions or the end of the game).

180 4. Declare a function to start the game loop. This function
181 initializes the game or scene, the frame time variables,
182 and the loop running flag before calling the first
183 `requestAnimationFrame()` with the `loopOnce` function as its
184 parameter to begin the game loop.

```
185 function start(scene) {  
186     if (mLoopRunning) {  
187         throw new Error("loop already running")  
188     }  
  
189     mCurrentScene = scene;  
190     mCurrentScene.init();  
  
191     mPrevTime = performance.now();  
192     mLagTime = 0.0;  
193     mLoopRunning = true;  
194     mFrameID = requestAnimationFrame(loopOnce);  
195 }
```

196 5. Declare a function to stop the game loop. This function simply
197 stops the loop by setting `mLoopRunning` to `false` and cancels the
198 last requested animation frame.

```
199 function stop() {  
200     mLoopRunning = false;  
201     // make sure no more animation frames  
202     cancelAnimationFrame(mFrameID);  
203 }
```

6. Lastly, remember to export the desired functionality to the rest of
the game engine, in this case just the start and stop functions: 204
205

```
export {start, stop}
```

206

Working with the Game Loop

207

To test the game loop implementation, your game class must now implement draw(),
update(), and init() functions. This is because to coordinate the beginning and the
continual operation of your game, these functions are being called from the core of the
game loop—the init() function is called from loop.start(), while the draw() and
update() functions are called from loop.loopOnce(). 208
209
210
211
212

1. Edit your my_game.js file to provide access to the loop by
importing from the module. Allowing game developer access
to the game loop module is a temporary measure and will be
corrected in later sections. 213
214
215
216

```
// Accessing engine internal is not ideal,  
//      this must be resolved! (later)  
import * as loop from "../engine/core/loop.js";
```

217

218

219

2. Replace the MyGame constructor with the following: 220

```
constructor() {  
    // variables for the squares  
    this.mWhiteSq = null;          // these are the Renderable objects  
    this.mRedSq = null;  
  
    // The camera to view the scene  
    this.mCamera = null;  
}
```

221

222

223

224

225

226

227

3. Add an initialization function to set up a camera and two
Renderable objects: 228
229

```
init() {  
    // Step A: set up the cameras  
    this.mCamera = new engine.Camera(  
        vec2.fromValues(20, 60),    // position of the camera
```

230

231

232

233

CHAPTER 4 IMPLEMENTING COMMON COMPONENTS OF VIDEO GAMES

```
234     20,                      // width of camera
235     [20, 40, 600, 300]        // viewport (orgX, orgY, width, height)
236   );
237   this.mCamera.setBackgroundColor([0.8, 0.8, 0.8, 1]);
238   // sets the background to gray

239   // Step B: Create the Renderable objects:
240   this.mWhiteSq = new engine.Renderable();
241   this.mWhiteSq.setColor([1, 1, 1, 1]);
242   this.mRedSq = new engine.Renderable();
243   this.mRedSq.setColor([1, 0, 0, 1]);

244   // Step C: Init the white Renderable: centered, 5x5, rotated
245   this.mWhiteSq.getXform().setPosition(20, 60);
246   this.mWhiteSq.getXform().setRotationInRad(0.2); // In Radians
247   this.mWhiteSq.getXform().setSize(5, 5);

248   // Step D: Initialize the red Renderable object: centered 2x2
249   this.mRedSq.getXform().setPosition(20, 60);
250   this.mRedSq.getXform().setSize(2, 2);
251 }
```

- 252 4. Draw the scene as before by clearing the canvas, setting up the
- 253 camera, and drawing each square:

```
254 draw() {
255   // Step A: clear the canvas
256   engine.clearCanvas([0.9, 0.9, 0.9, 1.0]); // clear to light gray

257   // Step B: Activate the drawing Camera
258   this.mCamera.setViewAndCameraMatrix();

259   // Step C: Activate the white shader to draw
260   this.mWhiteSq.draw(this.mCamera);

261   // Step D: Activate the red shader to draw
262   this.mRedSq.draw(this.mCamera);
263 }
```

5. Add an update() function to animate a moving white square and
a pulsing red square: 264
265

```
update() { 266
    // Simple game: move the white square and pulse the red 267
    let whiteXform = this.mWhiteSq.getXform(); 268
    let deltaX = 0.05; 269

    // Step A: Rotate the white square 270
    if (whiteXform.getXPos() > 30) // the right-bound of the window 271
        whiteXform.setPosition(10, 60); 272
    whiteXform.incXPosBy(deltaX); 273
    whiteXform.incRotationByDegree(1); 274

    // Step B: pulse the red square 275
    let redXform = this.mRedSq.getXform(); 276
    if (redXform.getWidth() > 5) 277
        redXform.setSize(2, 2); 278
    redXform.incSizeBy(0.05); 279
}
```

280

Recall that the update() function is called at about 60 times per second, and each time the following happens: 281
282

- Step A for the white square: Increase the rotation by 1 degree,
increase the x position by 0.05, and reset to 10 if the resulting
x position is greater than 30. 283
284
285
 - Step B for the red square: Increase the size by 0.05 and reset it to 2 if
the resulting size is greater than 5. 286
287
 - Since the previous operations are performed continuously at about
60 times a second, you can expect to see the following: 288
289
- a. A white square rotating while moving toward the right and
upon reaching the right boundary wrapping around to the left
boundary 290
291
292
 - b. A red square increasing in size and reducing to a size of 2
when the size reaches 5, thus appearing to be pulsing 293
294

- 295 6. Start the game loop from the `window.onload` function. Notice that
296 a reference to an instance of `MyGame` is passed to the loop.

```
297 window.onload = function () {  
298     engine.init("GLCanvas");  
299     let myGame = new MyGame();  
300     // now begins the game  
301     loop.start(myGame);  
302 }
```

303 You can now run the project to observe the rightward-moving, rotating white
304 square and the pulsing red square. You can control the rate of the movement,
305 rotation, and pulsing by changing the corresponding values of the `incXPosBy()`,
306 `incRotationByDegree()`, and `incSizeBy()` functions. In these cases, the positional,
307 rotational, and size values are changed by a constant amount in a fixed time interval.
308 In effect, the parameters to these functions are the rate of change, or the speed,
309 `incXPosBy(0.05)`, is the rightward speed of 0.05 units per 1/60th of a second or 3 units
310 per second. In this project, the width of the world is 20 units, and with the white square
311 traveling at 3 units per second, you can verify that it takes slightly more than 6 seconds
312 for the white square to travel from the left to the right boundary.

313 Notice that in the core of the `loop` module, it is entirely possible for the
314 `requestAnimationFrame()` function to invoke the `loopOnce()` function multiple times
315 within a single kMPF interval. When this happens, the `draw()` function will be called
316 multiples times without any `update()` function calls. In this way, the game loop can end
317 up drawing the same game state multiple times. Please refer to the following references
318 for discussions of supporting extrapolations in the `draw()` function to take advantage of
319 efficient game loops:

- <http://gameprogrammingpatterns.com/game-loop.html#play-catch-up>
- <http://gafferongames.com/game-physics/fix-your-timestep/>

323 To clearly describe each component of the game engine and illustrate how these
324 components interact, this book does not support extrapolation of the `draw()` function.

Keyboard Input

325

It is obvious that proper support to receive player input is important to interactive video games. For a typical personal computing device such as a PC or a Mac, the two common input devices are the keyboard and the mouse. While keyboard input is received in the form of a stream of characters, mouse input is packaged with positional information and is related to camera views. For this reason, keyboard input is more straightforward to support at this point in the development of the engine. This section will introduce and integrate keyboard support into your game engine. Mouse input will be examined in the “Mouse Input project” of Chapter 7, after the coverage of supporting multiple cameras in the same game.

326

327

328

329

330

331

332

333

334

The Keyboard Support Project

335

This project examines keyboard input support and integrates the functionality into the game engine. The position, rotation, and size of the game objects in this project are under your input control. You can see an example of this project running in Figure 4-2. The source code to this project is defined in the chapter4/4.2.keyboard_support folder.

336

337

338

339

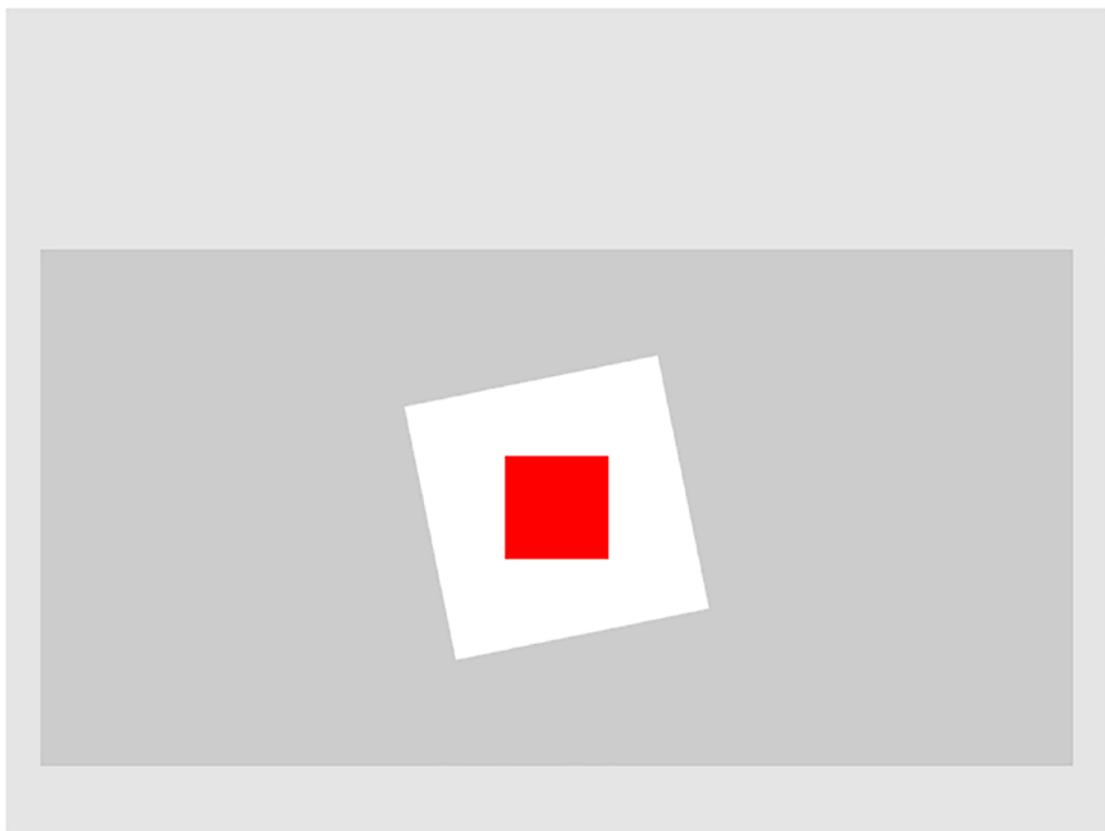


Figure 4-2. Running the Keyboard Support project

340 The controls of the project are as follows:

- 341 • **Right-arrow key:** Moves the white square toward the right and wraps
342 it to the left of the game window
- 343 • **Up-arrow key:** Rotates the white square
- 344 • **Down-arrow key:** Increases the size of the red square and then resets
345 the size at a threshold

346 The goals of the project are as follows:

- 347 • To implement an engine component to receive keyboard input
- 348 • To understand the difference between key state (if a key is released or
349 pressed) and key event (when the key state changes)
- 350 • To understand how to integrate the input component in the game loop

Add an Input Component to the Engine

351

Recall that the loop component is part of the core of the game engine and should not be accessed by the client game developer. In contrast, a well-defined input module should support the client game developer to query keyboard states without being distracted by any details. For this reason, the input module will be defined in the `src/engine` folder.

352

354

355

1. Create a new file in the `src/engine` folder and name it `input.js`. 356
2. Define a JavaScript dictionary to capture the key code mapping: 357

```
"use strict"  
// Key code constants  
const keys = {  
    // arrows  
    Left: 37,  
    Up: 38,  
    Right: 39,  
    Down: 40,  
  
    // space bar  
    Space: 32,  
  
    // numbers  
    Zero: 48,  
    One: 49,  
    Two: 50,  
    Three: 51,  
    Four: 52,  
    Five : 53,  
    Six : 54,  
    Seven : 55,  
    Eight : 56,  
    Nine : 57,  
  
    // Alphabets  
    A : 65,  
    D : 68,  
    E : 69,
```

358

359

360

361

362

363

364

365

366

367

368

369

370

371

372

373

374

375

376

377

378

379

380

381

382

```

383     F : 70,
384     G : 71,
385     I : 73,
386     J : 74,
387     K : 75,
388     L : 76,
389     Q : 81,
390     R : 82,
391     S : 83,
392     W : 87,
393     LastKeyCode: 222
394 }

```

Key codes are unique numbers representing each keyboard character. Note that there are up to 222 unique keys. In the listing, only a small subset of the keys, those that are relevant to this project, are defined in the dictionary.

Note Key codes for the alphabets are continuous, starting from 65 for A and ending with 90 for Z. You should feel free to add any characters for your own game engine. For a complete list of key codes, see www.cambiaresearch.com/articles/15/javascript-char-codes-key-codes.

3. Create array instance variables for tracking the states of every key:

```

402
403 // Previous key state
404 let mKeyPreviousState = []; // a new array
405 // The pressed keys.
406 let mIsKeyPressed = [];
407 // Click events: once an event is set, it will remain there until polled
408 let mIsKeyClicked = [];

```

All three arrays define the state of every key as a boolean. The `mKeyPreviousState` records the key states from the previous update cycle, and the `mIsKeyPressed` records the current state of the keys. The key code entries of these two arrays are true when the corresponding keyboard keys are pressed, and false otherwise. The `mIsKeyClicked`

array captures key click events. The key code entries of this array are true only when
the corresponding keyboard key goes from being released to being pressed in two
consecutive update cycles.

It is important to note that KeyPress is the state of a key, while KeyClicked is an
event. For example, if a player presses the *A* key for one second before releasing it, then
the duration of that entire second KeyPress for *A* is true, while KeyClick for *A* is true only
once—the update cycle right after the key is pressed.

4. Define functions to capture the actual keyboard state changes:

```
// Event handler functions
function onKeyDown(event) {
    mIsKeyPressed[event.keyCode] = true;
}

function onKeyUp(event) {
    mIsKeyPressed[event.keyCode] = false;
}
```

When these functions are called, the key code from the parameter is used to
record the corresponding keyboard state changes. It is expected that the caller of these
functions will pass the appropriate key code in the argument.

5. Add a function to initialize all the key states, and register the key event handlers with the browser. The `window`.

`addEventListener()` function registers the `onKeyUp/Down()` event
handlers with the browser such that the corresponding functions
will be called when the player presses or releases keys on the
keyboard.

```
function init() {
    let i;
    for (i = 0; i < keys.LastKeyCode; i++) {
        mIsKeyPressed[i] = false;
        mKeyPreviousState[i] = false;
        mIsKeyClicked[i] = false;
    }
}
```

CHAPTER 4 IMPLEMENTING COMMON COMPONENTS OF VIDEO GAMES

```
444     // register handlers
445     window.addEventListener('keyup', onKeyUp);
446     window.addEventListener('keydown', onKeyDown);
447 }
448
449     6. Add an update() function to derive the key click events. The
450         update() function uses mIsKeyPressed and mKeyPreviousState
451         to determine whether a key clicked event has occurred.
452
453     function update() {
454         let i;
455         for (i = 0; i < keys.LastKeyCode; i++) {
456             mIsKeyClicked[i] = (!mKeyPreviousState[i]) && mIsKeyPressed[i];
457             mKeyPreviousState[i] = mIsKeyPressed[i];
458         }
459     }
460
461     7. Add public functions for inquiries to current keyboard states to
462         support the client game developer:
463
464     // Function for GameEngine programmer to test if a key is pressed down
465     function isKeyPressed(keyCode) {
466         return mIsKeyPressed[keyCode];
467     }
468
469     function isKeyClicked(keyCode) {
470         return mIsKeyClicked[keyCode];
471     }
472
473     8. Finally, export the public functions and key constants:
474
475     export {keys, init,
476             update,
477             isKeyClicked,
478             isKeyPressed
479     }
```

Modify the Engine to Support Keyboard Input

473

To properly support input, before the game loop begins, the engine must initialize the mIsKeyPressed, mIsKeyClicked, and mKeyPreviousState arrays. To properly capture the player actions, during gameplay from within the core of the game loop, these arrays must be updated accordingly.

1. Input state initialization: Modify index.js by importing the input.js module, adding the initialization of the input to the engine init() function, and adding the input module to the exported list to allow access from the client game developer.

```
import * as input from "./input.js";                                     482

function init(htmlCanvasID) {                                              483
    glSys.init(htmlCanvasID);                                               484
    vertexBuffer.init();                                                 485
    shaderResources.init();                                              486
    input.init();                                                       487
}

export default {                                                       488
    // input support                                                 489
    input,                                                       490
    // Util classes                                                 492
    Camera, Transform, Renderable,                                         493
    // functions                                                 494
    init, clearCanvas                                                 495
}
```

2. To accurately capture keyboard state changes, the input component must be integrated with the core of the game loop. Include the input's update() function in the core game loop by adding the following lines to loop.js. Notice the rest of the code is identical.

```
import * as input from "../input.js";                                     501

function loopOnce() {
```

```

503     if (mLoopRunning) {
504         ... identical to previous code ...
505
506         // Step D: update the game the appropriate number of times.
507         //           Update only every Milliseconds per frame.
508         //           If lag larger then update frames, update until caught up.
509         while ((mLagTime >= kMPF) && mLoopRunning) {
510             input.update();
511             mCurrentScene.update();
512             mLagTime -= kMPF;
513         }
514     }

```

Test Keyboard Input

515 You can test the input functionality by modifying the Renderable objects in your MyGame class. Replace the code in the MyGame update() function with the following:

```

518 update() {
519     // Simple game: move the white square and pulse the red
520     let whiteXform = this.mWhiteSq.getXform();
521     let deltaX = 0.05;
522
523     // Step A: test for white square movement
524     if (engine.input.isKeyPressed(engine.input.keys.Right)) {
525         if (whiteXform.getXPos() > 30) { // right-bound of the window
526             whiteXform.setPosition(10, 60);
527         }
528         whiteXform.incXPosBy(deltaX);
529     }
530
531     // Step B: test for white square rotation
532     if (engine.input.isKeyClicked(engine.input.keys.Up)) {
533         whiteXform.incRotationByDegree(1);
534     }

```

```

let redXform = this.mRedSq.getXform();          533
// Step C: test for pulsing the red square      534
if (engine.input.isKeyPressed(engine.input.keys.Down)) { 535
    if (redXform.getWidth() > 5) {            536
        redXform.setSize(2, 2);                537
    }
    redXform.incSizeBy(0.05);                 538
}
}                                              540
}                                              541

```

In the previous code, step A ensures that pressing and holding the right-arrow key will move the white square toward the right. Step B checks for the pressing and then the releasing of the up-arrow key event. The white square is rotated when such an event is detected. Notice that pressing and holding the up-arrow key will not generate continuously key press events and thus will not cause the white square to continuously rotate. Step C tests for the pressing and holding of the down-arrow key to pulse the red square.

You can run the project and include additional controls for manipulating the squares. For example, include support for the **WASD** keys to control the location of the red square. Notice once again that by increasing/decreasing the position change amount, you are effectively controlling the speed of the object's movement.

Note The term “**WASD** keys” is used to refer to the key binding of the popular game controls: key W to move upward, A leftward, S downward, and D rightward.

Resource Management and Asynchronous Loading

Video games typically utilize a multitude of artistic assets, or resources, including audio clips and images. The required resources to support a game can be large. Additionally, it is important to maintain the independence between the resources and the actual game such that they can be updated independently, for example, changing the background audio without changing the game itself. For these reasons, game resources are typically stored externally on a system hard drive or a server across the network. Being stored external to the game, the resources are sometimes referred to as *external resources* or *assets*.

563 After a game begins, external resources must be explicitly loaded. For efficient
564 memory utilization, a game should load and unload resources dynamically based
565 on necessity. However, loading external resources may involve input/output device
566 operations or network packet latencies and thus can be time intensive and potentially
567 affect real-time interactivity. For these reasons, at any instance in a game, only a portion
568 of resources are kept in memory, where the loading operations are strategically executed
569 to avoid interrupting the game. In most cases, resources required in each level are kept
570 in memory during the gameplay of that level. With this approach, external resource
571 loading can occur during level transitions where players are expecting a new game
572 environment and are more likely to tolerate slight delays for loading.

573 Once loaded, a resource must be readily accessible to support interactivity. The
574 efficient and effective management of resources is essential to any game engine.
575 Take note of the clear differentiation between resource management, which is
576 the responsibility of a game engine, and the actual ownerships of the resources.
577 For example, a game engine must support the efficient loading and playing of the
578 background music for a game, and it is the game (or client of the game engine) that
579 actually owns and supplies the audio file for the background music. When implementing
580 support for external resource management, it is important to remember that the actual
581 resources are not part of the game engine.

582 At this point, the game engine you have been building handles only one type
583 of resource—the GLSL shader files. Recall that the `SimpleShader` object loads and
584 compiles the `simple_vs.glsl` and `simple_fs.glsl` files in its constructor. So far, the
585 shader file loading has been accomplished via synchronous `XMLHttpRequest.open()`.
586 This synchronous loading is an example of inefficient resource management because
587 no operations can occur while the browser attempts to open and load a shader file.
588 An efficient alternative would be to issue an asynchronous load command and allow
589 additional operations to continue while the file is being opened and loaded.

590 This section builds an infrastructure to support asynchronous loading and efficient
591 accessing of the loaded resources. Based on this infrastructure, over the next few
592 projects, the game engine will be expanded to support batch resource loading during
593 scene transitions.

The Resource Map and Shader Loader Project

594

This project guides you to develop the resource_map component, an infrastructural module for resource management, and demonstrates how to work with this module to load shader files asynchronously. You can see an example of this project running in Figure 4-3. This project appears to be identical to the previous project, with the only difference being how the GLSL shaders are loaded. The source code to this project is defined in the chapter4/4.3.resource_map_and_shader_loader folder.

595

596

597

598

599

600

this figure will be printed in b/w

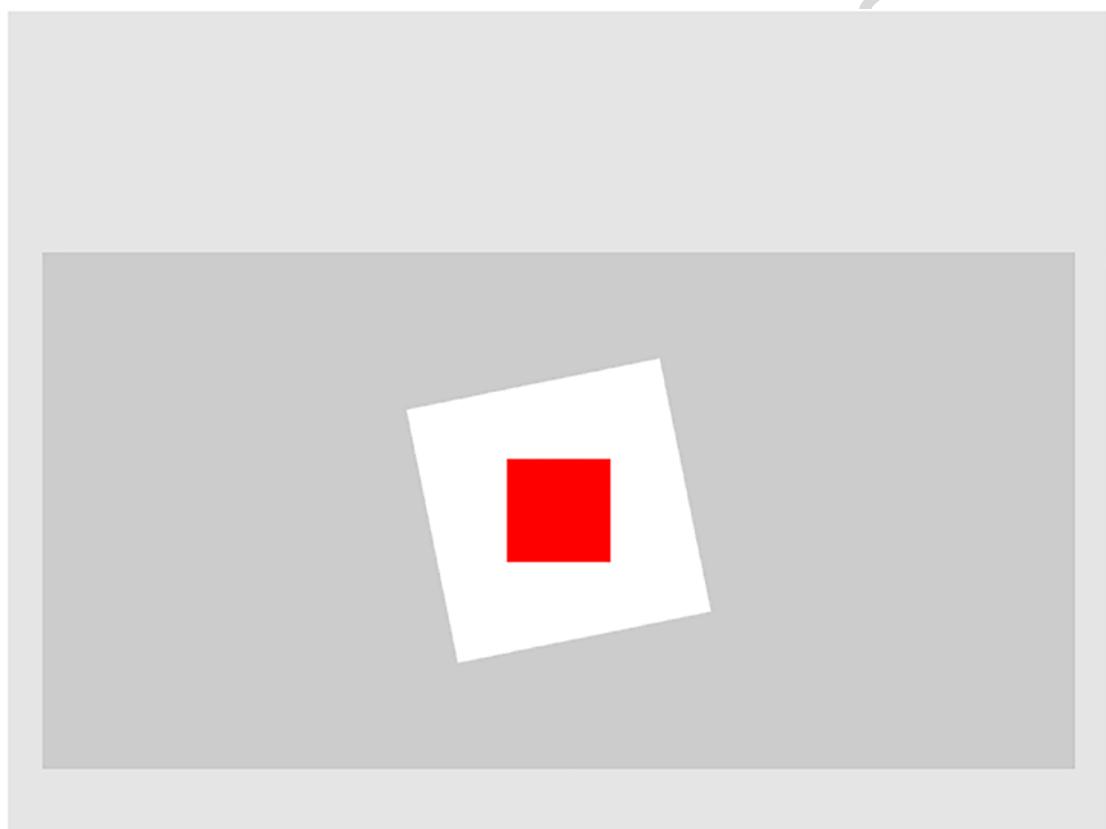


Figure 4-3. Running the Resource Map and Shader Loader project

The controls of the project are identical to the previous project as follows:

601

- **Right-arrow key:** Moves the white square toward the right and wraps it to the left of the game window
- **Up-arrow key:** Rotates the white square

602

603

604

- 605 • **Down-arrow key:** Increases the size of the red square and then resets
 606 the size at a threshold

607 The goals of the project are as follows:

- 608 • To understand the handling of asynchronous loading
 609 • To build an infrastructure that supports future resource loading and
 610 accessing
 611 • To experience asynchronous resource loading via loading of the
 612 GLSL shader files

613 **Note** For more information about asynchronous JavaScript operations, you can
 614 refer to many excellent resources online, for example, <https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous>.

616 Add a Resource Map Component to the Engine

617 The `resource_map` engine component manages resource loading, storage, and retrieval
 618 after the resources are loaded. These operations are internal to the game engine and
 619 should not be accessed by the game engine client. As in the case of all core engine
 620 components, for example, the game loop, the source code file is created in the `src/`
 621 `engine/core` folder. The details are as follows.

- 622 1. Create a new file in the `src/engine/core` folder and name it
 623 `resource_map.js`.
- 624 2. Define the `MapEntry` class to support reference counting of loaded
 625 resources. Reference counting is essential to avoid multiple
 626 loading or premature unloading of a resource.

```
627 class MapEntry {
 628   constructor(data) {
 629     this.mData = data;
 630     this.mRefCount = 1;
 631   }
 632   decRef() { this.mRefCount--; }
```

```

incRef() { this.mRefCount++; } 633

set(data) { this.mData = data; } 634

data() { return this.mData; } 635

canRemove() { return (this.mRefCount == 0); } 636

} 637

```

3. Define a key-value pair map, `mMap`, for storing and retrieving of resources and an array, `mOutstandingPromises`, to capture all outstanding asynchronous loading operations: 638

```

let mMap = new Map(); 641

let mOutstandingPromises = []; 642

```

Note A JavaScript Map object holds a collection of key-value pairs. 643

4. Define functions for querying the existence of, retrieving, and setting a resource. Notice that as suggested by the variable name of the parameter, `path`, it is expected that the full path to the external resource file will be used as the key for accessing the corresponding resource, for example, using the path to the `src/gls1_shaders/simple_vs.gls1` file as the key for accessing the content of the file. 644

```

function has(path) { return mMap.has(path) } 651

function get(path) { 652
    if (!has(path)) { 653
        throw new Error("Error [" + path + "]: not loaded"); 654
    } 655
    return mMap.get(path).data(); 656
}
function set(key, value) { mMap.get(key).set(value); } 658

```

- 659 5. Define functions to indicate that loading has been requested,
660 increase the reference count of a loaded resource, and to properly
661 unload a resource. Due to the asynchronous nature of the loading
662 operation, a load request will result in an empty MapEntry which
663 will be updated when the load operation is completed sometime
664 in the future. Note that each unload request will decrease the
665 reference count and may or may not result in the resource being
666 unloaded.

```
667 function loadRequested(path) {  
668     mMap.set(path, new MapEntry(null));  
669 }  
670 function incRef(path) {  
671     mMap.get(path).incRef();  
672 }  
673 function unload(path) {  
674     let entry = mMap.get(path);  
675     entry.decRef();  
676     if (entry.canRemove())  
677         mMap.delete(path)  
678     return entry.canRemove();  
679 }
```

- 680 6. Define a function to append an ongoing asynchronous loading
681 operation to the mOutstandingPromises array

```
682 function pushPromise(p) { mOutstandingPromises.push(p); }
```

- 683 7. Define a loading function, loadDecodeParse(). If the resource
684 is already loaded, the corresponding reference count is
685 incremented. Otherwise, the function first issues a loadRequest()
686 to create an empty MapEntry in mMap. The function then creates an
687 HTML5 fetch promise, using the path to the resource as key, to
688 asynchronously fetch the external resource, decode the network
689 packaging, parse the results into a proper format, and update the
690 results into the created MapEntry. This created promise is then
691 pushed into the mOutstandingPromises array.

```
// generic loading function, 692
// Step 1: fetch from server 693
// Step 2: decodeResource on the loaded package 694
// Step 3: parseResource on the decodedResource 695
// Step 4: store result into the map 696
// Push the promised operation into an array 697
function loadDecodeParse(path, decodeResource, parseResource) { 698
    let fetchPromise = null; 699
    if (!has(path)) { 700
        loadRequested(path);
        fetchPromise = fetch(path)
            .then(res => decodeResource(res) )
            .then(data => parseResource(data) )
            .then(data => { return set(path, data) } )
            .catch(err => { throw err } );
        pushPromise(fetchPromise); 707
    } else { 708
        incRef(path); // increase reference count 709
    }
    return fetchPromise;
} 711
} 712
```

Notice that the decoding and parsing functions are passed in as parameters and thus are dependent upon the actual resource type that is being fetched. For example, the decoding and parsing of simple text, XML (Extensible Markup Language)-formatted text, audio clips, and images all have distinct requirements. It is the responsibility of the actual resource loader to define these functions.

The HTML5 `fetch()` function returns a JavaScript promise object. A typical JavaScript promise object contains operations that will be completed in the future. A promise is fulfilled when the operations are completed. In this case, the `fetchPromise` is fulfilled when the path is properly fetched, decoded, parsed, and updated into the corresponding `MapEntry`. This promise is being kept in the `mOutstandingPromises` array. Note that by the end of the `loadDecodeParse()` function, the asynchronous `fetch()` loading operation is issued and ongoing but not guaranteed to be completed. In this way, the `mOutstandingPromises` is an array of ongoing and unfulfilled, or outstanding, promises.

- 727 8. Define a JavaScript `async` function to block the execution and wait
728 for all outstanding promises to be fulfilled, or wait for all ongoing
729 asynchronous loading operations to be completed:

```
730 // will block, wait for all outstanding promises complete
731 // before continue
732 async function waitOnPromises() {
733     await Promise.all(mOutstandingPromises);
734     mOutstandingPromises = []; // remove all
735 }
```

736 **Note** The JavaScript `async/await` keywords are paired where only `async`
737 functions can await for a promise. The `await` statement blocks and returns
738 the execution back to the caller of the `async` function. When the promise being
739 waited on is fulfilled, execution will continue to the end of the `async` function.

- 740 9. Finally, export functionality to the rest of the game engine:

```
741 export {has, get, set,
742         loadRequested, incRef, loadDecodeParse,
743         unload,
744         pushPromise, waitOnPromises}
```

745 Notice that although the storage-specific functionalities—`query`, `get`, and `set`—are
746 well defined, `resource_map` is actually not capable of loading any specific resources. This
747 module is designed to be utilized by resource type-specific modules where the decoding
748 and parsing functions can be properly defined. In the next subsection, a text resource
749 loader is defined to demonstrate this idea.

750 Define a Text Resource Module

751 This section will define a text module that utilizes the `resource_map` module to load
752 your text files asynchronously. This module serves as an excellent example of how to
753 take advantage of the `resource_map` facility and allows you to replace the synchronous
754 loading of GLSL shader files. Replacing synchronous with asynchronous loading support
755 is a significant upgrade to the game engine.

1. Create a new folder in `src/engine/` and name it `resources`. This new folder is created in anticipation of the necessary support for many resource types and to maintain a clean source code organization. 756
757
758
759
2. Create a new file in the `src/engine/resources` folder and name it `text.js`. 760
761
3. Import the core resource management and reuse the relevant functionality from `resource_map`: 762
763

```
"use strict"  
import * as map from "../core/resource_map.js";  
  
// functions from resource_map  
let unload = map.unload; 764  
765  
let has = map.has; 766  
let get = map.get; 767  
768  
769
```

4. Define the text decoding and parsing functions for `loadDecodeParse()`. Notice that there are no requirements for parsing the loaded text, and thus, the text parsing function does not perform any useful operation. 770
771
772
773

```
function decodeText(data) { 774  
    return data.text(); 775  
}  
function parseText(text) { 776  
    return text; 777  
} 778  
779
```

5. Define the `load()` function to call the `resource_map.loadDecodeParse()` function to trigger the asynchronous `fetch()` operation: 780
781
782

```
function load(path) { 783  
    return map.loadDecodeParse(path, decodeText, parseText); 784  
}  
785
```

- 786 6. Export the functionality to provide access to the rest of the game
 787 engine:

788 `export {has, get, load, unload}`

- 789 7. Lastly, remember to update the defined functionality for the client
 790 in the `index.js`:

791 `import * as text from "./resources/text.js";`

792 ... identical to previous code ...

793 `export default {`

794 `// resource support`

795 `text,`

796 ... identical to previous code ...

797 `}`

798 Load Shaders Asynchronously

799 The text resource module can now be used to assist the loading of the shader
 800 files asynchronously as plain-text files. Since it is impossible to predict when an
 801 asynchronous loading operation will be completed, it is important to issue the load
 802 commands *before* the resources are needed and to ensure that the loading operations are
 803 *completed* before proceeding to retrieve the resources.

804 Modify Shader Resources for Asynchronous Support

805 To avoid loading the GLSL shader files synchronously, the files must be loaded before
 806 the creation of a `SimpleShader` object. Recall that a single instance of `SimpleShader`
 807 object is created in the `shader_resources` module and shared among all `Renderables`.
 808 You can now asynchronously load the GLSL shader files before the creation of the
 809 `SimpleShader` object.

- 810 1. Edit `shader_resources.js` and import functionality from the `text`
 811 and `resource_map` modules:

812 `import * as text from "../resources/text.js";`

813 `import * as map from "./resource_map.js";`

2. Replace the content of the `init()` function. Define a JavaScript promise, `loadPromise`, to load the two GLSL shader files asynchronously, and when the loading is completed, trigger the calling of the `createShaders()` function. Store the `loadPromise` in the `mOutstandingPromises` array of the `resource_map` by calling the `map.pushPromise()` function:

```

function init() {
    let loadPromise = new Promise(
        async function(resolve) {
            await Promise.all([
                text.load(kSimpleFS),
                text.load(kSimpleVS)
            ]);
            resolve();
        }).then(
            function resolve() { createShaders(); }
        );
    map.pushPromise(loadPromise);
}

```

Notice that after the `shader_resources init()` function, the loading of the two GLSL shader files would have begun. At that point, it is not guaranteed that the loading operations are completed and the `SimpleShader` object may not have been created. However, the promise that is based on the completion of these operations is stored in the `resource_map mOutstandingPromises` array. For this reason, it is guaranteed that these operations must have completed by the end of the `resource_map waitOnPromises()` function.

Modify SimpleShader to Retrieve the Shader Files

With the understanding that the GLSL shader files are already loaded, the changes to the `SimpleShader` object are straightforward. Instead of synchronously loading the shader files in the `loadAndCompileShader()` function, the contents to these files can simply be retrieved via the `text` resource.

- 845 1. Edit the `simple_shader.js` file and add an `import` from the `text`
846 module for retrieving the content of the GLSL shaders:

- 847 `import * as text from "./resources/text.js";`
- 848 2. Since no loading operations are required, you should change
849 the `loadAndCompileShader()` function name to simply
850 `compileShader()` and replace the file-loading commands by
851 text resource retrievals. Notice that the synchronous loading
852 operations are replaced by a single call to `text.get()` to retrieve
853 the file content based on the `filePath` or the unique resource
854 name for the shader file.

```
855 function compileShader(filePath, shaderType) {  
856     let shaderSource = null, compiledShader = null;  
857     let gl = glSys.get();  
  
858     // Step A: Access the shader textfile  
859     shaderSource = text.get(filePath);  
  
860     if (shaderSource === null) {  
861         throw new Error("WARNING:" + filePath + " not loaded!");  
862         return null;  
863     }  
  
864     ... identical to previous code ...  
865 }
```

- 866 3. Remember that in the `SimpleShader` constructor, the calls to
867 `loadAndCompileShader()` functions should be replaced by the
868 newly modified `compileShader()` functions, as follows:

```
869 constructor(vertexShaderPath, fragmentShaderPath) {  
870     ... identical to previous code ...  
  
871     // Step A: load and compile vertex and fragment shaders  
872     this.mVertexShader = compileShader(vertexShaderPath,  
873                                         gl.VERTEX_SHADER);
```

```

this.mFragmentShader = compileShader(fragmentShaderPath,
                                         gl.FRAGMENT_SHADER);          874
                                         875
... identical to previous code ...           876
}                                         877

```

Wait for Asynchronous Loading to Complete

With outstanding loading operations and incomplete shader creation, a client's game cannot be initialized because without SimpleShader, Renderable objects cannot be properly created. For this reason, the game engine must wait for all outstanding promises to be fulfilled before proceeding to initialize the client's game. Recall that client's game initialization is performed in the game loop start() function, right before the beginning of the first loop iteration.

1. Edit the loop.js file and import from the resource_map module:

```
import * as map from "./resource_map.js";          886
```

2. Modify the start() function to be an async function such that it is now possible to issue await and hold the execution by calling map.waitOnPromises() to wait for the fulfilment of all outstanding promises:

```

async function start(scene) {                      891
    if (mLoopRunning) {                            892
        throw new Error("loop already running")     893
    }                                              894
    // Wait for any async requests before game-load   895
    await map.waitOnPromises();                     896

    mCurrentScene = scene;                          897
    mCurrentScene.init();                         898

    mPrevTime = performance.now();                899
    mLagTime = 0.0;                                900
    mLoopRunning = true;                           901
    mFrameID = requestAnimationFrame(loopOnce);  902
}                                         903

```

904 **Test the Asynchronous Shader Loading**

905 You can now run the project with shaders being loaded asynchronously. Though the
906 output and interaction experience are identical to the previous project, you now have
907 a game engine that is much better equipped to manage the loading and accessing of
908 external resources.

909 The rest of this chapter further develops and formalizes the interface between the
910 client, MyGame, and the rest of the game engine. The goal is to define the interface to the
911 client such that multiple game-level instances can be created and interchanged during
912 runtime. With this new interface, you will be able to define what a game level is and
913 allow the game engine to load any level in any order.

914 **Game Level from a Scene File**

915 The operations involved in initiating a game level from a scene file can assist in the
916 derivation and refinement of the formal interface between the game engine and its
917 client. With a game level defined in a scene file, the game engine must first initiate
918 asynchronous loading, wait for the load completion, and then initialize the client for
919 the game loop. These steps present a complete functional interface between the game
920 engine and the client. By examining and deriving the proper support for these steps, the
921 interface between the game engine and its client can be refined.

922 **The Scene File Project**

923 This project uses the loading of a scene file as the vehicle to examine the necessary
924 public methods for a typical game level. You can see an example of this project running
925 in Figure 4-4. This project appears and interacts identically to the previous project with
926 the only difference being that the scene definition is asynchronously loaded from a file.
927 The source code to this project is defined in the chapter4/4.4.scene_file folder.

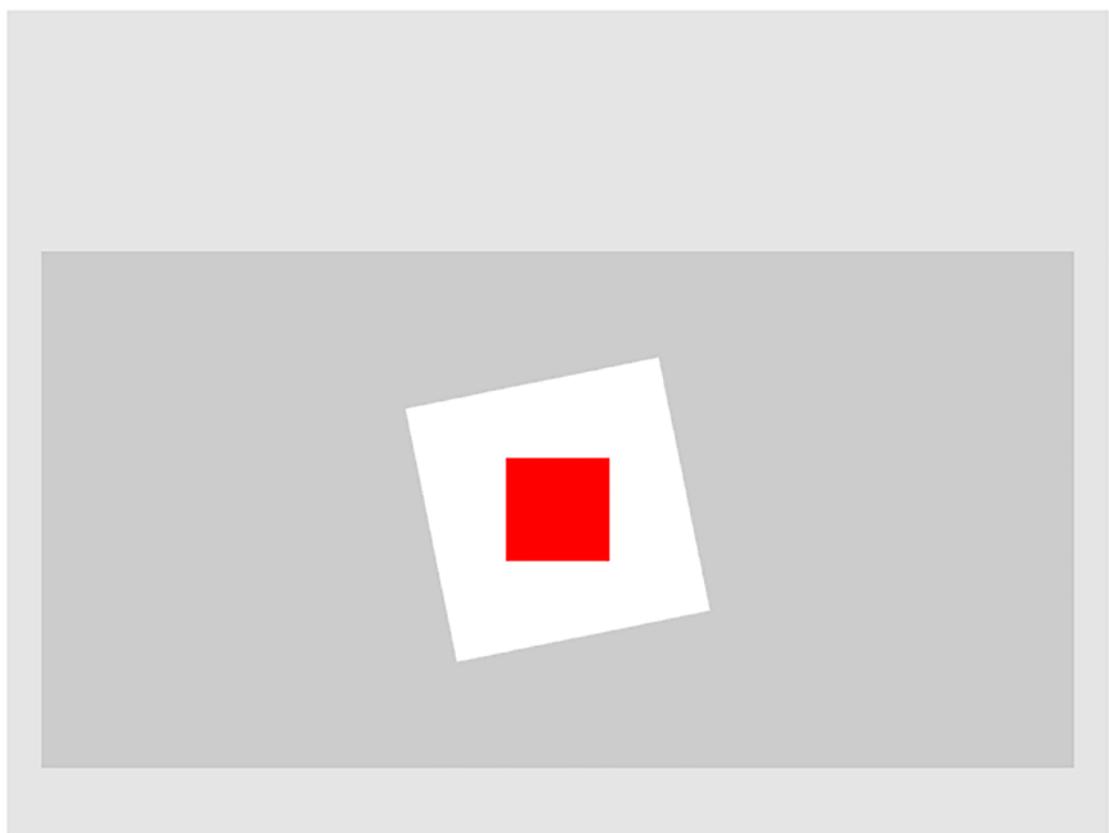


Figure 4-4. Running the Scene File project

The controls of the project are identical to the previous project, as follows: 928

- **Right-arrow key:** Moves the white square toward the right and wraps it to the left of the game window 929
- **Up-arrow key:** Rotates the white square 930
- **Down-arrow key:** Increases the size of the red square and then resets the size at a threshold 931
- **Space bar:** Toggles the rotation of the white square 932

The goals of the project are as follows: 934

- To introduce the protocol for supporting asynchronous loading of the resources of a game 935
- To develop the proper game engine support for the protocol 936
- To identify and define the public interface methods for a general game level 937
- To implement the logic for handling the user input 938
- To handle the rendering of the game objects 939

940 While the parsing and loading process of a scene file is interesting to a game engine
941 designer, the client should never need to concern themselves with these details. This
942 project aims at developing a well-defined interface between the engine and the client.
943 This interface will hide the complexity of the engine internal core from the client and
944 thus avoid situations such as requiring access to the loop module from MyGame in the first
945 project of this chapter.

946 The Scene File

947 Instead of hard-coding the creation of all objects to a game in the `init()` function,
948 the information can be encoded in a file, and the file can be loaded and parsed during
949 runtime. The advantage of such encoding in an external file is the flexibility to modify
950 a scene without the need to change the game source code, while the disadvantages are
951 the complexity and time required for loading and parsing. In general, the importance of
952 flexibility dictates that most game engines support the loading of game scenes from a file.

953 Objects in a game scene can be defined in many ways. The key decision factors are
954 that the format can properly describe the game objects and be easily parsed. Extensible
955 Markup Language (XML) is well suited to serve as the encoding scheme for scene files.

956 Define an XML Resource Module

957 In order to support an XML-encoded scene file, you first need to expand the engine to
958 support the asynchronous loading of an XML file resource. Similar to the `text` resource
959 module, an XML resource module should also be based on the `resource_map`: store the
960 loaded XML content in `mMap` of the `resource_map`, and define the specifics for decoding
961 and parsing for the calling of the `loadDecodeParse()` function of the `resource_map`.

- 962 1. Define a new file in the `src/engine/resources` folder and name
963 it `xml.js`. Edit this file and import the core resource management
964 functionality from the `resource_map`.

```
965 "use strict"  
966 import * as map from "../core/resource_map.js";  
967 // functions from resource_map  
968 let unload = map.unload;  
969 let has = map.has;  
970 let get = map.get;
```

2. Instantiate an XML DOMParser, define the decode and parsing functions, and call the loadDecodeParse() function of the resource_map with the corresponding parameters to initiate the loading of the XML file: 971
 972
 973
 974

```
let mParser = new DOMParser(); 975

function decodeXML(data) { 976
    return data.text();
}
977

function parseXML(text) { 979
    return mParser.parseFromString(text, "text/xml");
}
980
981

function load(path) { 982
    return map.loadDecodeParse(path, decodeXML, parseXML);
}
983
984
```

3. Remember to export the defined functionality: 985

```
export {has, get, load, unload} 986
```

4. Lastly, remember to export the defined functionality for the client 987
 in the index.js: 988

```
import * as xml from "./resources/xml.js"; 989

... identical to previous code ... 990

export default { 991
    // resource support
    text, xml,
    ...
    ... identical to previous code ...
}
994
995
```

The newly defined `xml` module can be conveniently accessed by the client and used 996
 in a similar fashion as the `text` module in loading external XML-encoded text files. 997

998 **Note** The JavaScript DOMParser provides the ability to parse XML or HTML text
999 strings.

1000 Modify the Engine to Integrate Client Resource Loading

1001 The scene file is an external resource that is being loaded by the client. With
1002 asynchronous operations, the game engine must stop and wait for the completion of the
1003 load process before it can initialize the game. This is because the game initialization will
1004 likely require the loaded resources.

1005 Coordinate Client Load and Engine Wait in the Loop Module

1006 Since all resource loading and storage are based on the same `resource_map`, the client
1007 issuing of the load requests and the engine waiting for the load completions can be
1008 coordinated in the `loop.start()` function as follows:

```
1009   async function start(scene) {  
1010       if (mLoopRunning) {  
1011           throw new Error("loop already running")  
1012       }  
1013       mCurrentScene = scene;  
1014       mCurrentScene.load();  
1015       // Wait for any async requests before game-load  
1016       await map.waitOnPromises();  
1017       mCurrentScene.init();  
1018       mPrevTime = performance.now();  
1019       mLagTime = 0.0;  
1020       mLoopRunning = true;  
1021       mFrameID = requestAnimationFrame(loopOnce);  
1022   }
```

1023 Note that this function is exactly two lines different from the previous
1024 project—`mCurrentScene` is assigned a reference to the parameter, and the client's `load()`
1025 function is called before the engine waits for the completion of all asynchronous loading
1026 operations.

Derive a Public Interface for the Client

1027

Though slightly involved, the details of XML-parsing specifics are less important than the fact that XML files can now be loaded. It is now possible to use the asynchronous loading of an external resource to examine the required public methods for interfacing a game level to the game engine.

1028

1029

1030

1031

Public Methods of MyGame

1032

While the game engine is designed to facilitate the building of games, the actual state of a game is specific to each individual client. In general, there is no way for the engine to anticipate the required operations to initialize, update, or draw any particular game. For this reason, such operations are defined to be part of the public interface between the game engine and the client. At this point, it is established that MyGame should define the following:

1033

1034

1035

1036

1037

1038

- `constructor()`: For declaring variables and defining constants.
- `init()`: For instantiating the variables and setting up the game scene. This is called from the `loop.start()` function before the first iteration of the game loop.
- `draw()/update()`: For interfacing to the game loop with these two functions being called continuously from within the core of the game loop, in the `loop.loopOnce()` function.

1039

1040

1041

1042

1043

1044

1045

With the requirement of loading a scene file, or any external resources, two additional public methods should be defined:

1046

1047

- `load()`: For initiating the asynchronous loading of external resources, in this case, the scene file. This is called from the `loop.start()` function before the engine waits for the completion of all asynchronous loading operations.
- `unload()`: For unloading of external resources when the game has ended. Currently, the engine does not attempt to free up resources. This will be rectified in the next project.

1048

1049

1050

1051

1052

1053

1054

1055 Implement the Client

1056 You are now ready to create an XML-encoded scene file to test external resource loading
 1057 by the client and to interface to the client with game engine based on the described
 1058 public methods.

1059 Define a Scene File

1060 Define a simple scene file to capture the game state from the previous project:

- 1061 1. Create a new folder at the same level as the `src` folder and name it
 1062 `assets`. This is the folder where all external resources, or assets, of
 1063 a game will be stored including the scene files, audio clips, texture
 1064 images, and fonts.

1065 **Tip** It is important to differentiate between the `src/engine/resources` folder
 1066 that is created for organizing game engine source code files and the `assets` folder
 1067 that you just created for storing client resources. Although GLSL shaders are also
 1068 loaded at runtime, they are considered as source code and will continue to be
 1069 stored in the `src/glsl_shaders` folder.

- 1070 2. Create a new file in the `assets` folder and name it `scene.xml`. This
 1071 file will store the client's game scene. Add the following content.
 1072 The listed XML content describes the same scene as defined in the
 1073 `init()` functions from the previous `MyGame` class.

```
1074 <MyGameLevel>
1075 <!-- *** be careful!! comma (,) is not a supported syntax!! -->
1076 <!-- make sure there are no comma in between attributes -->
1077 <!-- e.g., do NOT do: PosX="20", PosY="30" -->
1078 <!-- notice the "comma" between PosX and PosY: Syntax error! -->
1079   <!-- cameras -->
1080   <!-- Viewport: x, y, w, h -->
```

```

<Camera CenterX="20" CenterY="60" Width="20"          1081
    Viewport="20 40 600 300"                         1082
    BgColor="0.8 0.8 0.8 1.0"                        1083
/>                                                 1084

<!-- Squares Rotation is in degree -->           1085
<Square PosX="20" PosY="60" Width="5" Height="5"   1086
    Rotation="30" Color="1 1 1 1" />                1087
<Square PosX="20" PosY="60" Width="2" Height="2"   1088
    Rotation="0" Color="1 0 0 1" />                 1089
</MyGameLevel>                                    1090

```

Tip The JavaScript XML parser does not support delimiting attributes with commas. 1091

Parse the Scene File

A specific parser for the listed XML scene file must be defined to extract the scene information. Since the scene file is specific to a game, the parser should also be specific to the game and be created within the `my_game` folder. 1093
1094
1095

1. Create a new folder in the `src/my_game` folder and name it `util`.
Add a new file in the `util` folder and name it `scene_file_parser.js`.
This file will contain the specific parsing logic to decode the listed scene file. 1096
1097
1098
1099
2. Define a new class, name it `SceneFileParser`, and add a constructor with code as follows:
1100
1101

```

import engine from "../../engine/index.js";          1102

class SceneFileParser {                            1103
    constructor (xml) {                          1104
        this.xml = xml                           1105
    }                                         1106
    ... implementation to follow ...           1107
}

```

Note that the `xml` parameter is the actual content of the loaded XML file. 1109

1110 **Note** The following XML parsing is based on JavaScript XML API. Please refer to
1111 www.w3schools.com/dom for more details.

1112 3. Add a function to the SceneFileParser to parse the details of the
1113 Camera from the xml file you created:

```
1114 parseCamera() {  
1115     let camElm = getElm(this.xml, "Camera");  
1116     let cx = Number(camElm[0].getAttribute("CenterX"));  
1117     let cy = Number(camElm[0].getAttribute("CenterY"));  
1118     let w = Number(camElm[0].getAttribute("Width"));  
1119     let viewport = camElm[0].getAttribute("Viewport").split(" ");  
1120     let bgColor = camElm[0].getAttribute("BgColor").split(" ");  
1121     // make sure viewport and color are number  
1122     let j;  
1123     for (j = 0; j < 4; j++) {  
1124         bgColor[j] = Number(bgColor[j]);  
1125         viewport[j] = Number(viewport[j]);  
1126     }  
1127     let cam = new engine.Camera(  
1128         vec2.fromValues(cx, cy), // position of the camera  
1129         w, // width of camera  
1130         viewport // viewport (orgX, orgY, width, height)  
1131     );  
1132     cam.setBackgroundColor(bgColor);  
1133     return cam;  
1134 }
```

1135 The camera parser finds a camera element and constructs a Camera object with the
1136 retrieved information. Notice that the viewport and background color are arrays of four
1137 numbers. These are input as strings of four numbers delimited by spaces. Strings can be
1138 split into arrays, which is the case here with the space delimiter. The JavaScript Number()
1139 function ensures that all strings are converted into numbers.

4. Add a function to the SceneFileParser to parse the details of the squares from the xml file you created:

```

parseSquares(sqSet) {
    let elm = getElm(this.xml, "Square");
    let i, j, x, y, w, h, r, c, sq;
    for (i = 0; i < elm.length; i++) {
        x = Number(elm.item(i).attributes.getNamedItem("PosX").value);
        y = Number(elm.item(i).attributes.getNamedItem("PosY").value);
        w = Number(elm.item(i).attributes.getNamedItem("Width").value);
        h = Number(elm.item(i).attributes.getNamedItem("Height").value);
        r = Number(elm.item(i).attributes.getNamedItem("Rotation").value);
        c = elm.item(i).attributes.getNamedItem("Color").value.split(" ");
        sq = new engine.Renderable();
        // make sure color array contains numbers
        for (j = 0; j < 4; j++) {
            c[j] = Number(c[j]);
        }
        sq.setColor(c);
        sq.getXform().setPosition(x, y);
        sq.getXform().setRotationInDegree(r); // In Degree
        sq.getXform().setSize(w, h);
        sqSet.push(sq);
    }
}

```

This function parses the XML file to create Renderable objects to be placed in the array that is passed in as a parameter.

5. Add a function outside the SceneFileParser to parse for contents of an XML element:

```

function getElm(xmlContent, tagElm) {
    let theElm = xmlContent.getElementsByTagName(tagElm);
    if (theElm.length === 0) {
        console.error("Warning: Level element:[ " +

```

```

1172                     tagElm + "]": is not found!");
1173     }
1174     return theElm;
1175 }
```

1176 6. Finally, export the SceneFileParser:

```
1177 export default SceneFileParser;
```

1178 Implement MyGame

1179 The implementations of the described public functions for this project are as follows:

1180 1. Edit my_game.js file and import the SceneFileParser:

```
1181 import SceneFileParser from "./util/scene_file_parser.js";
```

1182 2. Modify the MyGame constructor to define the scene file path, the
1183 array mSqSet for storing the Renderable objects, and the camera:

```

1184 constructor() {
1185     // scene file name
1186     this.mSceneFile = "assets/scene.xml";
1187     // all squares
1188     this.mSqSet = [];  
        // these are the Renderable objects
1189     // The camera to view the scene
1190     this.mCamera = null;
1191 }
```

1192 3. Change the init() function to create objects based on
1193 the scene parser. Note the retrieval of the XML file content via the
1194 engine.xml.get() function where the file path to the scene file is
1195 used as the key.

```

1196 init() {
1197     let sceneParser = new SceneFileParser(
1198             engine.xml.get(this.mSceneFile));
1199     // Step A: Read in the camera
```

```

this.mCamera = sceneParser.parseCamera();          1200
// Step B: Read all the squares                  1201
sceneParser.parseSquares(this.mSqSet);            1202
}                                              1203

```

4. The draw and update functions are similar to the previous examples with the exception of referencing the corresponding array elements.

```

draw() {                                         1207
    // Step A: clear the canvas                 1208
    engine.clearCanvas([0.9, 0.9, 0.9, 1.0]);   1209
    this.mCamera.setViewAndCameraMatrix();        1210
    // Step B: draw all the squares             1211
    let i;
    for (i = 0; i < this.mSqSet.length; i++)
        this.mSqSet[i].draw(this.mCamera);       1213
}
update() {                                       1215
    // simple game: move the white square and pulse the red
    let xform = this.mSqSet[0].getXform();        1217
    let deltaX = 0.05;
    // Step A: test for white square movement
    ... identical to previous code ...
    xform = this.mSqSet[1].getXform();           1222
    // Step C: test for pulsing the red square
    ... identical to previous code ...
}

```

5. Lastly, define the functions to load and unload the scene file.

```

load() {                                         1227
    engine.xml.load(this.mSceneFile);            1228
}

```

```
1230 unload() {  
1231     // unload the scene file and loaded resources  
1232     engine.xml.unload(this.mSceneFile);  
1233 }
```

1234 You can now run the project and see that it behaves the same as the previous
1235 two projects. While this may not seem interesting, through this project, a simple and
1236 well-defined interface between the engine and the client has been derived where
1237 the complexities and details of each are hidden. Based on this interface, additional
1238 engine functionality can be introduced without the requirements of modifying any
1239 existing clients, and at the same time, complex games can be created and maintained
1240 independently from engine internals. The details of this interface will be introduced in
1241 the next project.

1242 Before continuing, you may notice that the `MyGame.unload()` function is never
1243 called. This is because in this example the game loop never stopped cycling and `MyGame`
1244 is never unloaded. This issue will be addressed in the next project.

1245 Scene Object: Client Interface to the Game Engine

1246 At this point, in your game engine, the following is happening:

- 1247 • The `window.onload` function initializes the game engine and calls the
1248 `loop.start()` function, passing in `MyGame` as a parameter.
- 1249 • The `loop.start()` function, through the `resource_map`, waits for the
1250 completion of all asynchronous loading operations before it calls to
1251 initialize `MyGame` and starts the actual game loop cycle.

1252 From this discussion, it is interesting to recognize that any object with the
1253 appropriately defined public methods can replace the `MyGame` object. Effectively, at any
1254 point, it is possible to call the `loop.start()` function to initiate the loading of a new
1255 scene. This section expands on this idea by introducing the `Scene` object for interfacing
1256 the game engine with its clients.

The Scene Objects Project

1257

This project defines the Scene object as an abstract superclass for interfacing with your game engine. From this project on, all client code must be encapsulated in subclasses of the abstract Scene class, and the game engine will be able to interact with these classes in a coherent and well-defined manner. You can see an example of this project running in Figure 4-5. The source code to this project is defined in the chapter4/4.5.scene_objects folder.

1258

1259

1260

1261

1262

this figure will be printed in b/w



Figure 4-5. Running the Scene Objects project with both scenes

There are two distinct levels in this project: the MyGame level with a blue rectangle drawn above a red square over a gray background and the BlueLevel level with a red rectangle drawn above a rotated white square over a dark blue background. For simplicity, the controls for both levels are the same.

1263

1264

1265

1266

- **Left-/right-arrow key:** Move the front rectangle left and right
- **Q key:** Quits the game

1267

1268

Notice that on each level, moving the front rectangle toward the left to touch the left boundary will cause the loading of the other level. The MyGame level will cause BlueLevel to be loaded, and BlueLevel will cause the MyGame level to be loaded.

1269

1270

1271

The goals of the project are as follows:

- To define the abstract Scene class to interface to the game engine
- To experience game engine support for scene transitions
- To create scene-specific loading and unloading support

1272

1273

1274

1275

1276 The Abstract Scene Class

1277 Based on the experience from the previous project, an abstract Scene class for
 1278 encapsulating the interface to the game engine must at the very least define these
 1279 functions: `init()`, `draw()`, `update()`, `load()`, and `unload()`. Missing from this list is the
 1280 support for level transitions to start, advance to the next level, and, if desired, to stop
 1281 the game.

- 1282 1. Create a new JavaScript file in the `src/engine` folder and name
 1283 it `scene.js`, and import from the `loop` module and the engine
 1284 access file `index.js`. These two modules are required because the
 1285 Scene object must start and end the game loop when the game
 1286 level begins and ends, and the engine must be cleaned up if a level
 1287 should decide to terminate the game.

```
1288 import * as loop from "./core/loop.js";
1289 import engine from "./index.js";
```

1290 **Note** The game loop must not be running before a Scene has begun. This is
 1291 because the required resources must be properly loaded before the `update()`
 1292 function of the Scene can be called from the running game loop. Similarly,
 1293 unloading of a level can only be performed after a game loop has stopped running.

- 1294 2. Define JavaScript Error objects for warning the client in case of
 1295 misuse:

```
1296 const kAbstractClassError = new Error("Abstract Class")
1297 const kAbstractMethodError = new Error("Abstract Method")
```

- 1298 3. Create a new class named Scene and export it:

```
1299 class Scene { ... implementation to follow ... }
1300 export default Scene;
```

4. Implement the constructor to ensure only subclasses of the Scene
class are instantiated: 1301
1302

```
constructor() {  
    if (this.constructor === Scene) {  
        throw kAbstractClassError  
    }  
}
```

- 1303
1304
1305
1306
1307
5. Define scene transition functions: start(), next(), and
stop(). The start() function is an async function because it is
responsible for starting the game loop, which in turn is waiting for
all the asynchronous loading to complete. Both the next() and
the stop() functions stop the game loop and call the unload()
function to unload the loaded resources. The difference is that the
next() function is expected to be overwritten and called from a
subclass where after unloading the current scene, the subclass can
proceed to advance to the next level. After unloading, the stop()
function assumes the game has terminated and proceeds to clean
up the game engine. 1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318

```
async start() {  
    await loop.start(this);  
}  
  
next() {  
    loop.stop();  
    this.unload();  
}  
  
stop() {  
    loop.stop();  
    this.unload();  
    engine.cleanUp();  
}
```

- 1331 6. Define the rest of the derived interface functions. Notice that
1332 the Scene class is an abstract class because all of the interface
1333 functions are empty. While a subclass can choose to only
1334 implement a selective subset of the interface functions, the draw()
1335 and update() functions are not optional because together they
1336 form the central core of a level.

```
1337 init() { /* to initialize the level (called from loop.start()) */ }  
1338 load() { /* to load necessary resources */ }  
1339 unload() { /* unload all resources */ }  
1340 // draw/update must be over-written by subclass  
1341 draw() { throw kAbstractMethodError; }  
1342 update() { throw kAbstractMethodError; }
```

1343 Together these functions present a protocol to interface with the game engine. It is
1344 expected that subclasses will override these functions to implement the actual game
1345 behaviors.

1346 **Note** JavaScript does *not* support abstract classes. The language does not
1347 prevent a game programmer from instantiating a Scene object; however, the
1348 created instance will be completely useless, and the error message will provide
1349 them with a proper warning.

1350 **Modify Game Engine to Support the Scene Object**

1351 The game engine must be modified in two important ways. First, the game engine access
1352 file, index.js, must be modified to export the newly introduced symbols to the client
1353 as is done with all new functionality. Second, the Scene.stop() function introduces
1354 the possibility of stopping the game and handles the cleanup and resource deallocation
1355 required.

Export the Scene Class to the Client

Edit the `index.js` file to import from `scene.js` and export `Scene` for the client: 1357

```
... identical to previous code ... 1358
import Scene from "./scene.js"; 1359
... identical to previous code ... 1360
export default { 1361
    ... identical to previous code ... 1362
    Camera, Scene, Transform, Renderable, 1363
    ... identical to previous code ... 1364
} 1365
```

Implement Engine Cleanup Support

It is important to release the allocated resources when the game engine shuts down. The cleanup process is rather involved and occurs in the reverse order of system component initialization. 1366

1. Edit `index.js` once again, this time to implement support for game engine cleanup. Import from the `loop` module, and then define and export the `cleanup()` function. 1370

```
... identical to previous code ... 1373
import * as loop from "./core/loop.js"; 1374
... identical to previous code ... 1375
function cleanUp() { 1376
    loop.cleanUp(); 1377
    input.cleanUp(); 1378
    shaderResources.cleanUp(); 1379
    vertexBuffer.cleanUp(); 1380
    glSys.cleanUp(); 1381
}
... identical to previous code ... 1382
export default { 1383
    ... identical to previous code ... 1384
    init, cleanUp, clearCanvas 1385
    ... identical to previous code ... 1386
} 1387
```

1389 **Note** Similar to other core engine internal components, such as `gl` or `vertex_`
1390 `buffer`, `loop` should not be accessed by the client. For this reason, `loop` module
1391 is imported but not exported by `index.js`, imported such that game loop cleanup
1392 can be invoked, not exported, such that the client can be shielded from irrelevant
1393 complexity within the engine.

1394 Notice that none of the components have defined their corresponding cleanup
1395 functions. You will now remedy this. In each of the following cases, make sure to
1396 remember to export the newly defined `cleanup()` function when appropriate.

1397 2. Edit `loop.js` to define and export a `cleanUp()` function to stop the
1398 game loop and unload the currently active scene:

1399 ... identical to previous code ...
1400 function cleanUp() {
1401 if (mLoopRunning) {
1402 stop();
1403 // unload all resources
1404 mCurrentScene.unload();
1405 mCurrentScene = null;
1406 }
1407 }
1408 export {start, stop, **cleanUp**}

1409 3. Edit `input.js` to define and export a `cleanUp()` function. For now,
1410 no specific resources need to be released.

1411 ... identical to previous code ...
1412 function cleanUp() {} // nothing to do for now
1413 export {keys, init, **cleanUp**,
1414 ... identical to previous code ...}

4. Edit `shader_resources.js` to define and export a `cleanUp()` function to clean up the created shader and unload its source code:

```
... identical to previous code ...  

function cleanUp() {  

    mConstColorShader.cleanUp();  

    text.unload(kSimpleVS);  

    text.unload(kSimpleFS);  

}  

export {init, cleanUp, getConstColorShader}
```

5. Edit `simple_shader.js` to define the `cleanUp()` function for the `SimpleShader` class to release the allocated WebGL resources:

```
cleanUp() {  

    let gl = glSys.get();  

    gl.detachShader(this.mCompiledShader, this.mVertexShader);  

    gl.detachShader(this.mCompiledShader, this.mFragmentShader);  

    gl.deleteShader(this.mVertexShader);  

    gl.deleteShader(this.mFragmentShader);  

    gl.deleteProgram(this.mCompiledShader);  

}  


```

6. Edit `vertex_buffer.js` to define and export a `cleanUp()` function to delete the allocated buffer memory:

```
... identical to previous code ...  

function cleanUp() {  

    if (mGLVertexBuffer !== null) {  

        glSys.get().deleteBuffer(mGLVertexBuffer);  

        mGLVertexBuffer = null;  

    }  

}  

export {init, get, cleanUp}
```

```

1445      7. Lastly, edit gl.js to define and export a cleanUp() function to
1446          inform the player that the engine is now shut down:
1447
1448  ... identical to previous code ...
1449
1450  function cleanUp() {
1451      if ((mGL == null) || (mCanvas == null))
1452          throw new Error("Engine cleanup: system is not initialized.");
1453      mGL = null;
1454      // let the user know
1455      mCanvas.style.position = "fixed";
1456      mCanvas.style.backgroundColor = "rgba(200, 200, 200, 0.5)";
1457      mCanvas = null;
1458      document.body.innerHTML +=
1459          "<br><br><h1>End of Game</h1><h1>GL System Shut Down</h1>";
1460  }
1461  export {init, get, cleanUp}

```

1460 Test the Scene Class Interface to the Game Engine

1461 With the abstract Scene object definition and the resource management modifications to
 1462 the game engine core components, it is now possible to stop an existing scene and load a
 1463 new scene at will. This section cycles between two subclasses of the Scene object, MyGame
 1464 and BlueLevel, to illustrate the loading and unloading of scenes.

1465 For simplicity, the two test scenes are almost identical to the MyGame scene from
 1466 the previous project. In this project, MyGame explicitly defines the scene in the init()
 1467 function, while the BlueScene, in a manner identical to the case in the previous project,
 1468 loads the scene content from the blue_level.xml file located in the assets folder. The
 1469 content and the parsing of the XML scene file are identical to those from the previous
 1470 project and thus will not be repeated.

1471 The MyGame Scene

1472 As mentioned, this scene defines in the init() function the identical content found
 1473 in the scene file from the previous project. In the following section, take note of the
 1474 definition and calls to next() and stop() functions.

1. Edit `my_game.js` to import from `index.js` and the newly defined `blue_level.js`. Note that with Scene class support, you no longer need to import from the loop module.

```
import engine from "../engine/index.js";
import BlueLevel from "./blue_level.js";
```

2. Define `MyGame` to be a subclass of the engine Scene class, and remember to export `MyGame`:

```
class MyGame extends engine.Scene {
  ... implementation to follow ...
}
export default MyGame;
```

Note The JavaScript `extends` keyword defines the parent/child relationship.

3. Define the `constructor()`, `init()`, and `draw()` functions. Note that the scene content defined in the `init()` function, with the exception of the camera background color, is identical to that of the previous project.

```
constructor() {
  super();
  // The camera to view the scene
  this.mCamera = null;

  // the hero and the support objects
  this.mHero = null;
  this.mSupport = null;
}

init() {
  // Step A: set up the cameras
  this.mCamera = new engine.Camera(
    vec2.fromValues(20, 60), // position of the camera
    165
```

CHAPTER 4 IMPLEMENTING COMMON COMPONENTS OF VIDEO GAMES

```
1503     20,                                // width of camera
1504     [20, 40, 600, 300]      // viewport (orgX, orgY, width, height)
1505 );
1506 this.mCamera.setBackgroundColor([0.8, 0.8, 0.8, 1]);
1507 // Step B: Create the support object in red
1508 this.mSupport = new engine.Renderable();
1509 this.mSupport.setColor([0.8, 0.2, 0.2, 1]);
1510 this.mSupport.getXform().setPosition(20, 60);
1511 this.mSupport.getXform().setSize(5, 5);
1512 // Step C: Create the hero object in blue
1513 this.mHero = new engine.Renderable();
1514 this.mHero.setColor([0, 0, 1, 1]);
1515 this.mHero.getXform().setPosition(20, 60);
1516 this.mHero.getXform().setSize(2, 3);
1517 }
1518 draw() {
1519     // Step A: clear the canvas
1520     engine.clearCanvas([0.9, 0.9, 0.9, 1.0]);
1521     // Step B: Activate the drawing Camera
1522     this.mCamera.setViewAndCameraMatrix();
1523     // Step C: draw everything
1524     this.mSupport.draw(this.mCamera);
1525     this.mHero.draw(this.mCamera);
1526 }
1527 4. Define the update() function; take note of the this.next() call
1528     when the mHero object crosses the x=11 boundary from the right
1529     and the this.stop() call when the Q key is pressed.
1530 update() {
1531     // let's only allow the movement of hero,
1532     // and if hero moves too far off, this level ends, we will
1533     // load the next level
1534     let deltaX = 0.05;
```

```

let xform = this.mHero.getXform();                                1535

// Support hero movements                                         1536
if (engine.input.isKeyPressed(engine.input.keys.Right)) {          1537
    xform.incXPosBy(deltaX);                                       1538
    if (xform.getXPos() > 30) { // right-bound of the window      1539
        xform.setPosition(12, 60);                                    1540
    }
}                                                               1541

}                                                               1542

if (engine.input.isKeyPressed(engine.input.keys.Left)) {           1543
    xform.incXPosBy(-deltaX);                                     1544
    if (xform.getXPos() < 11) { // left-bound of the window       1545
        this.next();                                              1546
    }
}                                                               1547

}                                                               1548

if (engine.input.isKeyPressed(engine.input.keys.Q))                1549
    this.stop(); // Quit the game                                 1550
}
}

```

5. Define the next() function to transition to the BlueLevel scene: 1552

```

next() {
    super.next(); // this must be called!                         1553
    // next scene to run                                         1554
    let nextLevel = new BlueLevel(); // next level to be loaded   1555
    nextLevel.start();                                           1556
}
}

```

Note The super.next() call, where the super class can stop the game loop and cause the unloading of this scene, is necessary and absolutely critical in causing the scene transition.

1559

1560

1561

- 1562 6. Lastly, modify the `window.onload()` function to replace access to
 1563 the loop module with a client-friendly `myGame.start()` function:

```
1564 window.onload = function () {  

1565   engine.init("GLCanvas");  

1566   let myGame = new MyGame();  

1567   myGame.start();  

1568 }
```

1569 The BlueLevel Scene

1570 The `BlueLevel` scene is almost identical to the `MyGame` object from the previous project
 1571 with the exception of supporting the new `Scene` class and scene transition:

- 1572 1. Create and edit `blue_level.js` file in the `my_game` folder to import
 1573 from the `engine index.js`, `MyGame`, and `SceneFileParser`. Define
 1574 and export `BlueLevel` to be a subclass of the `engine.Scene` class.

```
1575 // Engine Core stuff  

1576 import engine from "../engine/index.js";  

1577 // Local stuff  

1578 import MyGame from "./my_game.js";  

1579 import SceneFileParser from "./util/scene_file_parser.js";  

1580 class BlueLevel extends engine.Scene {  

1581   ... implementation to follow ...  

1582 }  

1583 export default BlueLevel
```

- 1584 2. Define the `init()`, `draw()`, `load()`, and `unload()` functions to be
 1585 identical to those in the `MyGame` class from the previous project.
- 1586 3. Define the `update()` function similar to that of the `MyGame` scene.
 1587 Once again, note the `this.next()` call when the object crosses the
 1588 `x=11` boundary from the right and the `this.stop()` call when the
 1589 Q key is pressed.

```

update() {
    // For this very simple game, let's move the first square
    let xform = this.mSQSet[1].getXform();
    let deltaX = 0.05;

    /// Move right and swap over
    if (engine.input.isKeyPressed(engine.input.keys.Right)) {
        xform.incXPosBy(deltaX);
        if (xform.getXPos() > 30) { // right-bound of the window
            xform.setPosition(12, 60);
        }
    }

    // test for white square movement
    if (engine.input.isKeyPressed(engine.input.keys.Left)) {
        xform.incXPosBy(-deltaX);
        if (xform.getXPos() < 11) { // this is the left-boundary
            this.next(); // go back to my game
        }
    }

    if (engine.input.isKeyPressed(engine.input.keys.Q))
        this.stop(); // Quit the game
}

```

4. Lastly, define the `next()` function to transition to the `MyGame` scene. It is worth reiterating that the call to `super.next()` is necessary because it is critical to stop the game loop and unload the current scene before proceeding to the next scene.

```

next() {
    super.next();
    let nextLevel = new MyGame(); // load the next level
    nextLevel.start();
}

```

1620 You can now run the project and view the scenes unloading and loading and quit
1621 the game at any point during the interaction. Your game engine now has a well-defined
1622 interface for working with its client. This interface follows the well-defined protocol of
1623 the Scene object.

- 1624 • `constructor()`: For declaring variables and defining constants.
1625 • `start()/stop()`: For starting a scene and stopping the game. These
1626 two methods are not meant to be overwritten by a subclass.

1627 The following interface methods are meant to be overwritten by subclasses.

- 1628 • `init()`: For instantiating the variables and setting up the game scene.
1629 • `load()/unload()`: For initiating the asynchronous loading and
1630 unloading of external resources.
1631 • `draw()/update()`: For continuously displaying the game state and
1632 receiving player input and implementing the game logic.
1633 • `next()`: For instantiating and transitioning to the next scene. Lastly,
1634 as a final reminder, it is absolutely critical for the subclass to call the
1635 `super.next()` to stop the game loop and unload the scene.

1636 Any objects that define these methods can be loaded and interacted with by your
1637 game engine. You can experiment with creating other levels.

1638 **Audio**

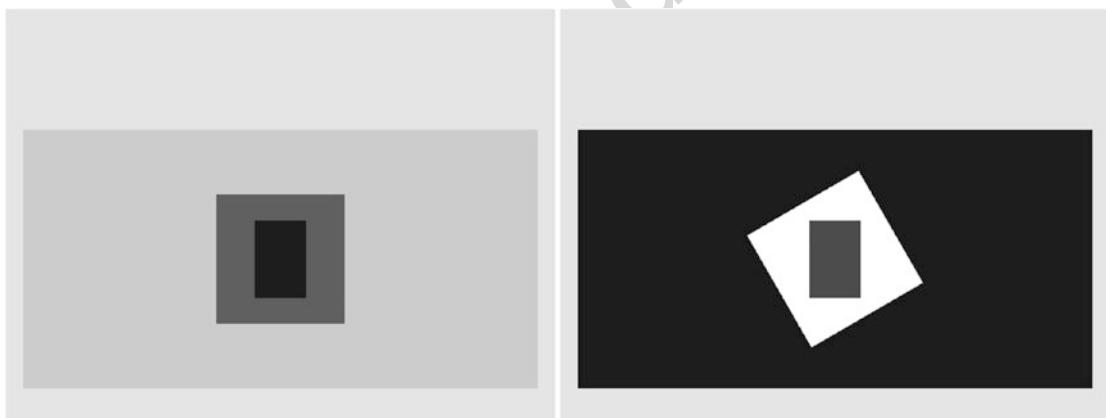
1639 Audio is an essential element of all video games. In general, audio effects in games fall
1640 into two categories. The first category is background audio. This includes background
1641 music or ambient effects and is often used to bring atmosphere or emotion to different
1642 portions of the game. The second category is sound effects. Sound effects are useful for
1643 all sorts of purposes, from notifying users of game actions to hearing the footfalls of your
1644 hero character. Usually, sound effects represent a specific action, triggered either by the
1645 user or by the game itself. Such sound effects are often thought of as an audio cue.

1646 One important difference between these two types of audio is how you control them.
1647 Sound effects or cues cannot be stopped or have their volume adjusted once they have

started; therefore, cues are generally short. On the other hand, background audio can
1648
be started and stopped at will. These capabilities are useful for stopping the background
1649 track completely and starting another one.
1650

The Audio Support Project

This project has identical MyGame and the BlueLevel scenes to the previous project.
1652
You can move the front rectangle left or right with the arrow keys, the intersection with
1653 the left boundary triggers the loading of the other scene, and the Q key quits the game.
1654 However, in this version, each scene plays background music and triggers a brief audio
1655 cue when the left-/right-arrow key is pressed. Notice that the volume varies for each type
1656 of audio clip. The implementation of this project also reinforces the concept of loading
1657 and unloading of external resources and the audio clips themselves. You can see an
1658 example of this project running in Figure 4-6. The source code to this project is defined
1659 in the chapter4/4.6.audio_support folder.
1660



this figure will be printed in b/w

Figure 4-6. Running the Audio Support project with both scenes

The controls of the project are as follows:

- **Left-/right-arrow key:** Moves the front rectangle left and right to
increase and decrease the volume of the background music
1662
1663
- **Q key:** Quits the game
1664

1665 The goals of the project are as follows:

- 1666 • To add audio support to the resource management system
- 1667 • To provide an interface to play audio for games

1668 You can find the following audio files in the assets/sounds folder:

- 1669 • bg_clip.mp3
- 1670 • blue_level_cue.wav
- 1671 • my_game_cue.wav

1672 Notice that the audio files are in two formats, mp3 and wav. While both are
1673 supported, audio files of these formats should be used with care. Files in .mp3 format are
1674 compressed and are suitable for storing longer durations of audio content, for example,
1675 for background music. Files in .wav format are uncompressed and should contain only
1676 very short audio snippet, for example, for storing cue effects.

1677 Define an Audio Resource Module

1678 While audio and text files are completely different, from the perspective of your game
1679 engine implementation, there are two important similarities. First, both are external
1680 resources and thus will be implemented similarly as engine components in the
1681 src/engine/resources folder. Second, both involve standardized file formats with
1682 well-defined API utilities. The Web Audio API will be used for the actual retrieving
1683 and playing of sound files. Even though this API offers vast capabilities, in the interests
1684 of focusing on the rest of the game engine development, only basic supports for
1685 background audio and effect cues are discussed.

1686 **Note** Interested readers can learn more about the Web Audio API from
1687 www.w3.org/TR/webaudio/.

1688 The latest policy for some browsers, including Chrome, is that audio will not be
1689 allowed to play until first interaction from the user. This means that the context
1690 creation will result in an initial warning from Chrome that is output to the runtime
1691 browser console. The audio will only be played after user input (e.g., mouse click or
1692 keyboard events).

1. In the `src/engine/resources` folder, create a new file and name it `audio.js`. This file will implement the module for the audio component. This component must support two types of functionality: loading and unloading of audio files and playing and controlling of the content of audio file for the game developer. 1693
1694
1695
1696
1697
2. The loading and unloading are similar to the implementations of `text` and `xml` modules where the core resource management functionality is imported from `resource_map`: 1698
1699
1700

```
"use strict"; 1701

import * as map from "../core/resource_map.js"; 1702
// functions from resource_map 1703
let unload = map.unload; 1704
let has = map.has; 1705
```

3. Define the decode and parsing functions, and call the `resource_map loadDecodeParse()` function to load an audio file. Notice that with the support from `resource_map` and the rest of the engine infrastructure, loading and unloading of external resources have become straightforward. 1706
1707
1708
1709
1710

```
function decodeResource(data) { return data.arrayBuffer(); } 1711
function parseResource(data) {
    return mAudioContext.decodeAudioData(data); } 1712
function load(path) {
    return map.loadDecodeParse(path, decodeResource, parseResource); 1713
} 1714
} 1715
} 1716
```

4. With the loading functionality completed, you can now define the audio control and manipulation functions. Declare variables to maintain references to the Web Audio context and background music and to control volumes. 1717
1718
1719
1720

```
let mAudioContext = null; 1721
let mBackgroundAudio = null; 1722
// volume control support
let mBackgroundGain = null; // background volume 1723
} 1724
```

CHAPTER 4 IMPLEMENTING COMMON COMPONENTS OF VIDEO GAMES

```
1725 let mCueGain = null;           // cue/special effects volume
1726 let mMasterGain = null;        // overall/master volume
1727 let kDefaultInitGain = 0.1;
1728
1729     5. Define the init() function to create and store a reference to the
1730         Web Audio context in mAudioContext, and initialize the audio
1731         volume gain controls for the background, cue, and a master that
1732         affects both. In all cases, volume gain of a 0 corresponds to no
1733         audio and 1 means maximum loudness.
1734
1735 function init() {
1736     try {
1737         let AudioContext = window.AudioContext ||
1738             window.webkitAudioContext;
1739         mAudioContext = new AudioContext();
1740
1741         // connect Master volume control
1742         mMasterGain = mAudioContext.createGain();
1743         mMasterGain.connect(mAudioContext.destination);
1744         // set default Master volume
1745         mMasterGain.gain.value = kDefaultInitGain;
1746
1747         // connect Background volume control
1748         mBackgroundGain = mAudioContext.createGain();
1749         mBackgroundGain.connect(mMasterGain);
1750         // set default Background volume
1751         mBackgroundGain.gain.value = 1.0;
1752
1753         // connect Cuevolume control
1754         mCueGain = mAudioContext.createGain();
1755         mCueGain.connect(mMasterGain);
1756         // set default Cue volume
1757         mCueGain.gain.value = 1.0;
1758     } catch (e) {
1759         throw new Error("...");
1760     }
1761 }
```

6. Define the `playCue()` function to play the entire duration of an audio clip with proper volume control. This function uses the audio file path as a resource name to find the loaded asset from the `resource_map` and then invokes the Web Audio API to play the audio clip. Notice that no reference to the `source` variable is kept, and thus once started, there is no way to stop the corresponding audio clip. A game should call this function to play short snippets of audio clips as cues. 1757
1758
1759
1760
1761
1762
1763
1764

```
function playCue(path, volume) {  
    let source = mAudioContext.createBufferSource();  
    source.buffer = map.get(path);  
    source.start(0);  
  
    // volume support for cue  
    source.connect(mCueGain);  
    mCueGain.gain.value = volume;  
}  
1765  
1766  
1767  
1768  
1769  
1770  
1771  
1772
```

7. Define the functionality to play, stop, query, and control the volume of the background music. In this case, the `mBackgroundAudio` variable keeps a reference to the currently playing audio, and thus, it is possible to stop the clip or change its volume. 1773
1774
1775
1776
1777

```
function playBackground(path, volume) {  
    if (has(path)) {  
        stopBackground();  
        mBackgroundAudio = mAudioContext.createBufferSource();  
        mBackgroundAudio.buffer = map.get(path);  
        mBackgroundAudio.loop = true;  
        mBackgroundAudio.start(0);  
  
        // connect volume accordingly  
        mBackgroundAudio.connect(mBackgroundGain);  
        setBackgroundVolume(volume);  
    }  
}  
1778  
1779  
1780  
1781  
1782  
1783  
1784  
1785  
1786  
1787  
1788  
1789
```

CHAPTER 4 IMPLEMENTING COMMON COMPONENTS OF VIDEO GAMES

```
1790 function stopBackground() {  
1791     if (mBackgroundAudio !== null) {  
1792         mBackgroundAudio.stop(0);  
1793         mBackgroundAudio = null;  
1794     }  
1795 }  
  
1796 function isBackgroundPlaying() {  
1797     return (mBackgroundAudio !== null);  
1798 }  
  
1799 function setBackgroundVolume(volume) {  
1800     if (mBackgroundGain !== null) {  
1801         mBackgroundGain.gain.value = volume;  
1802     }  
1803 }  
  
1804 function incBackgroundVolume(increment) {  
1805     if (mBackgroundGain !== null) {  
1806         mBackgroundGain.gain.value += increment;  
1807         // need this since volume increases when negative  
1808         if (mBackgroundGain.gain.value < 0) {  
1809             setBackgroundVolume(0);  
1810         }  
1811     }  
1812 }  
  
1813 8. Define functions for controlling the master volume, which adjusts  
1814     the volume of both the cue and the background music:
```

```
1815 function setMasterVolume(volume) {  
1816     if (mMasterGain !== null) {  
1817         mMasterGain.gain.value = volume;  
1818     }  
1819 }
```

```

function incMasterVolume(increment) {           1820
    if (mMasterGain !== null) {                 1821
        mMasterGain.gain.value += increment;     1822
        // need this since volume increases when negative
        if (mMasterGain.gain.value < 0) {          1823
            mMasterGain.gain.value = 0;           1824
        }
    }
}

```

9. Define a `cleanUp()` function to release the allocated HTML5 resources:

```

function cleanUp() {                         1831
    mAudioContext.close();                   1832
    mAudioContext = null;                  1833
}

```

10. Remember to export the functions from this module:

```

export {init, cleanUp,                      1836
        has, load, unload,                  1837
        playCue,                          1838
        playBackground, stopBackground,   1839
        isBackgroundPlaying,             1840
        setBackgroundVolume,              1841
        incBackgroundVolume,
        setMasterVolume, incMasterVolume
}

```

Export the Audio Module to the Client

Edit the `index.js` file to import from `audio.js`, initialize and cleanup the module accordingly, and to export to the client:

```

... identical to previous code ...
import * as audio from "./resources/audio.js";
... identical to previous code ...

```

```
1849 function init(htmlCanvasID) {  
1850     glSys.init(htmlCanvasID);  
1851     vertexBuffer.init();  
1852     shaderResources.init();  
1853     input.init();  
1854     audio.init();  
1855 }  
  
1856 function cleanUp() {  
1857     loop.cleanUp();  
1858     audio.cleanUp();  
1859     input.cleanUp();  
1860     shaderResources.cleanUp();  
1861     vertexBuffer.cleanUp();  
1862     glSys.cleanUp();  
1863 }  
1864 ... identical to previous code ...  
1865 export default {  
1866     // resource support  
1867     audio, text, xml  
1868     ... identical to previous code ...  
1869 }
```

1870 Testing the Audio Component

1871 To test the audio component, you must copy the necessary audio files into your game
1872 project. Create a new folder in the assets folder and name it sounds. Copy the bg_clip.
1873 mp3, blue_level_cue.wav, and my_game_cue.wav files into the sounds folder. You will
1874 now need to update the MyGame and BlueLevel implementations to load and use these
1875 audio resources.

Change MyGame.js

1876

Update MyGame scene to load the audio clips, play background audio, and cue the player when the arrow keys are pressed:

1. Declare constant file paths to the audio files in the constructor.

Recall that these file paths are used as resource names for loading, storage, and retrieval. Declaring these as constants for later reference is a good software engineering practice.

1879

1880

1881

1882

```
constructor() {  
    super();  
  
    // audio clips: supports both mp3 and wav formats  
    this.mBackgroundAudio = "assets/sounds/bg_clip.mp3";  
    this.mCue = "assets/sounds/my_game_cue.wav";  
    ... identical to previous code ...  
}
```

1883

1884

1885

1886

1887

1888

1889

2. Request the loading of audio clips in the load() function, and make sure to define the corresponding unload() function. Notice that the unloading of background music is preceded by stopping the music. In general, a resource's operations must be halted prior to its unloading.

1890

1891

1892

1893

1894

```
load() {  
    // loads the audios  
    engine.audio.load(this.mBackgroundAudio);  
    engine.audio.load(this.mCue);  
}  
  
unload() {
```

1895

1896

1897

1898

1899

1900

```
    // Step A: Game loop not running, unload all assets  
    // stop the background audio  
    engine.audio.stopBackground();  
  
    // unload the scene resources  
    engine.audio.unload(this.mBackgroundAudio);  
    engine.audio.unload(this.mCue);  
}
```

1901

1902

1903

1904

1905

1906

1908 3. Start the background audio at the end of the `init()` function.

1909 `init() {`
1910 ... identical to previous code ...
1911 `// now start the Background music ...`
1912 **`engine.audio.playBackground(this.mBackgroundAudio, 1.0);`**
1913 `}`

1914 4. In the `update()` function, cue the players when the right- and left-
1915 arrow keys are pressed, and increase and decrease the volume of
1916 the background music:

1917 `update() {`
1918 ... identical to previous code ...
1919 `// Support hero movements`
1920 `if (engine.input.isKeyPressed(engine.input.keys.Right)) {`
1921 **`engine.audio.playCue(this.mCue, 0.5);`**
1922 **`engine.audio.incBackgroundVolume(0.05);`**
1923 `xform.incXPosBy(deltaX);`
1924 `if (xform.getXPos() > 30) { // right-bound of the window`
1925 `xform.setPosition(12, 60);`
1926 `}`
1927 `}`

1928 `if (engine.input.isKeyPressed(engine.input.keys.Left)) {`
1929 **`engine.audio.playCue(this.mCue, 1.5);`**
1930 **`engine.audio.incBackgroundVolume(-0.05);`**
1931 `xform.incXPosBy(-deltaX);`
1932 `if (xform.getXPos() < 11) { // left-bound of the window`
1933 `this.next();`
1934 `}`
1935 `}`
1936 ... identical to previous code ...
1937 `}`

Change BlueLevel.js

The changes to the BlueLevel scene are similar to those of the MyGame scene but with a different audio cue:

1. In the BlueLevel constructor, add the following path names to the audio resources:

```
constructor() {  
    super();  
  
    // audio clips: supports both mp3 and wav formats  
    this.mBackgroundAudio = "assets/sounds/bg_clip.mp3";  
    this.mCue = "assets/sounds/blue_level_cue.wav";  
    ... identical to previous code ...  
}
```

2. Modify the load() and unload() functions for the audio clips:

```
load() {  
    engine.xml.load(this.mSceneFile);  
    engine.audio.load(this.mBackgroundAudio);  
    engine.audio.load(this.mCue);  
}  
  
unload() {  
    // stop the background audio  
    engine.audio.stopBackground();  
  
    // unload the scene file and loaded resources  
    engine.xml.unload(this.mSceneFile);  
    engine.audio.unload(this.mBackgroundAudio);  
    engine.audio.unload(this.mCue);  
}
```

3. In the same manner as MyGame, start the background audio in the init() function and cue the player when the left and right keys are pressed in the update() function. Notice that in this case, the audio cues are played with different volume settings.

CHAPTER 4 IMPLEMENTING COMMON COMPONENTS OF VIDEO GAMES

```
1968 init() {  
1969     ... identical to previous code ...  
1970     // now start the Background music ...  
1971     engine.audio.playBackground(this.mBackgroundAudio, 0.5);  
1972 }  
  
1973 update() {  
1974     ... identical to previous code ...  
1975     // Move right and swap over  
1976     if (engine.input.isKeyPressed(engine.input.keys.Right)) {  
1977         engine.audio.playCue(this.mCue, 0.5);  
1978         xform.incXPosBy(deltaX);  
1979         if (xform.getXPos() > 30) { // right-bound of the window  
1980             xform.setPosition(12, 60);  
1981         }  
1982     }  
1983     // Step A: test for white square movement  
1984     if (engine.input.isKeyPressed(engine.input.keys.Left)) {  
1985         engine.audio.playCue(this.mCue, 1.0);  
1986         xform.incXPosBy(-deltaX);  
1987         if (xform.getXPos() < 11) { // this is the left-boundary  
1988             this.next(); // go back to my game  
1989         }  
1990     }  
1991     ... identical to previous code ...  
1992 }
```

1993 You can now run the project and listen to the wonderful audio feedback. If you press
1994 and hold the arrow keys, there will be many cues repeatedly played. In fact, there are
1995 so many cues echoed that the sound effects are blurred into an annoying blast. This
1996 serves as an excellent example illustrating the importance of using audio cues with
1997 care and ensuring each individual cue is nice and short. You can try tapping the arrow
1998 keys to listen to more distinct and pleasant-sounding cues, or you can simply replace
1999 the `isKeyPressed()` function with the `isKeyClicked()` function and listen to each
2000 individual cue.

Summary

In this chapter, you learned how several common components of a game engine come together. Starting with the ever-important game loop, you learned how it implements an input, update, and draw pattern in order to surpass human perception or trick our senses into believing that the system is continuous and running in real time. This pattern is at the heart of any game engine. You learned how full keyboard support can be implemented with flexibility and reusability to provide the engine with a reliable input component. Furthermore, you saw how a resource manager can be implemented to load files asynchronously and how scenes can be abstracted to support scenes being loaded from a file, which can drastically reduce duplication in the code. Lastly, you learned how audio support supplies the client with an interface to load and play both ambient background audio and audio cues.

These components separately have little in common but together make up the core fundamentals of nearly every game. As you implement these core components into the game engine, the games that are created with the engine will not need to worry about the specifics of each component. Instead, the games programmer can focus on utilizing the functionality to hasten and streamline the development process. In the next chapter, you will learn how to create the illusion of an animation with external images.

Game Design Considerations

In this chapter, we discussed the *game loop* and the technical foundation contributing to the connection between what the player does and how the game responds. If a player selects a square that's drawn on the screen and moves it from location A to location B by using the arrow keys, for example, you'd typically want that action to appear as a smooth motion beginning as soon as the arrow key is pressed, without stutters, delays, or noticeable lag. The game loop contributes significantly to what's known as *presence* in game design; presence is the player's ability to feel as if they're connected to the game world, and responsiveness plays a key role in making players feel connected. Presence is reinforced when actions in the real world (such as pressing arrow keys) seamlessly translate to responses in the game world (such as moving objects, flipping switches, jumping, and so on); presence is compromised when actions in the real world suffer translation errors such as delays and lag.

As mentioned in Chapter 1, effective game mechanic design can begin with just a few simple elements. By the time you've completed the *Keyboard Support* project in this

2034 chapter, for example, many of the pieces will already be in place to begin constructing
2035 game levels: you've provided players with the ability to manipulate two individual
2036 elements on the screen (the red and white squares), and all that remains in order to
2037 create a basic game loop is to design a causal chain using those elements that results in
2038 a new event when completed. Imagine the *Keyboard Support* project is your game: how
2039 might you use what's available to create a causal chain? You might choose to play with
2040 the relationship between the squares, perhaps requiring that the red square be moved
2041 completely within the white square in order to unlock the next challenge; once the player
2042 successfully placed the red square in the white square, the level would complete. This
2043 basic mechanic may not be quite enough on its own to create an engaging experience,
2044 but by including just a few of the other eight elements of game design (systems design,
2045 setting, visual design, music and audio, and the like), it's possible to turn this one basic
2046 interaction into an almost infinite number of engaging experiences and create that sense
2047 of presence for players. You'll add more game design elements to these exercises as you
2048 continue through subsequent chapters.

2049 The *Resource Map* and *Shader Loader* project, the *Scene File* project, and the
2050 *Scene Objects* project are designed to help you begin thinking about architecting game
2051 designs from the ground up for maximum efficiency so that problems such as asset
2052 loading delays that detract from the player's sense of presence are minimized. As you
2053 begin designing games with multiple stages and levels and many assets, a resource
2054 management plan becomes essential. Understanding the limits of available memory
2055 and how to smartly load and unload assets can mean the difference between a great
2056 experience and a frustrating experience.

2057 We experience the world through our senses, and our feeling of presence in games
2058 tends to be magnified as we include additional sensory inputs. The *Audio Support*
2059 project adds basic audio to our simple state-changing exercise from the *Scene Objects*
2060 project in the form of a constant background score to provide ambient mood and
2061 includes a distinct movement sound for each of the two areas. Compare the two
2062 experiences and consider how different they feel because of the presence of sound
2063 cues; although the visual and interaction experience is identical between the two, the
2064 *Audio Support* project begins to add some emotional cues because of the beat of the
2065 background score and the individual tones the rectangle makes as it moves. Audio is
2066 a powerful enhancement to interactive experiences and can dramatically increase a
2067 player's sense of presence in game environments, and as you continue through the
2068 chapters, you'll explore how audio contributes to game design in more detail.

Author Queries

Chapter No.: 4 0005244097

Queries	Details Required	Author's Response
AU1	Please check if edit to sentence starting “Notice the similarity...” is okay.	
AU2	Please check if edit to sentence starting “In the update() function...” is okay.	

Uncorrected Proof

CHAPTER 5

1

Working with Textures, Sprites, and Fonts

2

3

After completing this chapter, you will be able to

- Use any image or photograph as a texture representing characters or objects in your game
- Understand and use texture coordinates to identify a location on an image
- Optimize texture memory utilization by combining multiple characters and objects into one image
- Produce and control animations using sprite sheets
- Display texts of different fonts and sizes anywhere in your game

4

5

6

7

8

9

10

11

12

Introduction

13

Custom-composed images are used to represent almost all objects including characters, backgrounds, and even animations in most 2D games. For this reason, the proper support of image operations is core to 2D game engines. A game typically works with an image in three distinct stages: loading, rendering, and unloading.

14

15

16

17

Loading is the reading of the image from the hard drive of the web server into the client's system main memory, where it is processed and stored in the graphics subsystem. *Rendering* occurs during gameplay when the loaded image is drawn continuously to represent the respective game objects. *Unloading* happens when an image is no longer required by the game and the associated resources are reclaimed for future uses. Because of the slower response time of the hard drive and the potentially

18

19

20

21

22

23

CHAPTER 5 WORKING WITH TEXTURES, SPRITES, AND FONTS

large amount of data that must be transferred and processed, loading images can take a noticeable amount of time. This, together with the fact that, just like the objects that images represent, the usefulness of an image is usually associated with individual game level, image loading and unloading operations typically occur during game-level transitions. To optimize the number of loading and unloading operations, it is a common practice to combine multiple lower-resolution images into a single larger image. This larger image is referred to as a *sprite sheet*.

To represent objects, images with meaningful drawings are pasted, or *mapped*, on simple geometries. For example, a horse in a game can be represented by a square that is mapped with an image of a horse. In this way, a game developer can manipulate the transformation of the square to control the horse. This mapping of images on geometries is referred to as *texture mapping* in computer graphics.

The illusion of movement, or animation, can be created by cycling through strategically mapping selected images on the same geometry. For example, during subsequent game loop updates, different images of the same horse with strategically drawn leg positions can be mapped on the same square to create the illusion that the horse is galloping. Usually, these images of different animated positions are stored in one sprite sheet or an animated sprite sheet. The process of sequencing through these images to create animation is referred to as *sprite animation* or *sprite sheet animation*.

This chapter first introduces you to the concept of texture coordinates such that you can understand and program with the WebGL texture mapping interface. You will then build a core texture component and the associated classes to support mapping with simple textures, working with sprite sheets that contain multiple objects, creating and controlling motions with animated sprite sheets, and extracting alphabet characters from a sprite sheet to display text messages.

Note A texture is an image that is loaded into the graphics system and ready to be mapped onto a geometry. When discussing the process of texture mapping, “an image” and “a texture” are often used interchangeably. A pixel is a color location in an image and a *texel* is a color location in a texture.

Texture Mapping and Texture Coordinates

As discussed, texture mapping is the process of pasting an image on a geometry, just like putting a sticker on an object. In the case of your game engine, instead of drawing a constant color for each pixel occupied by the unit square, you will create GLSL shaders to strategically select texels from the texture and display the corresponding texel colors at the screen pixel locations covered by the unit square. The process of selecting a texel, or converting a group of texels into a single color, to be displayed to a screen pixel location is referred to as texture sampling. To render a texture-mapped pixel, the texture must be sampled to extract a corresponding texel color.

The process of mapping a texture of any resolution to a fixed-size geometry can be daunting. The Texture Coordinate System that specifies the Texture Space is designed to hide the resolution of textures to facilitate this mapping process. As depicted in Figure 5-1, the Texture Coordinate System is a normalized system defined over the entire texture with the origin located at the lower-left corner and (1,1) located at the top-right corner. This simple fact that the normalized 0 to 1 range is always defined over the entire texture regardless of the resolution is the elegance of the Texture Coordinate System. Given a texture of any resolution, (0.5, 0.5) is always the center, (0, 1) is always the top-left corner, and so on. Notice that in Figure 5-1 the horizontal axis is labeled as the u axis, and the vertical axis is labeled as the v axis. Oftentimes a texture coordinate, or the uv values associated with a texture coordinate, is used interchangeably to refer to a location in the Texture Coordinate System.

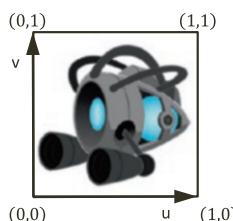


Figure 5-1. The Texture Coordinate System and the corresponding uv values defined for all images

Note There are conventions that define the v axis increasing either upward or downward. In all examples of this book, you will program WebGL to follow the convention in Figure 5-1, with the v axis increasing upward.

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

71

72

73

this figure will be printed in b/w

To map a texture onto a unit square, you must define a corresponding uv value for each of the vertex positions. As illustrated in Figure 5-2, in addition to defining the value of the xy position for each of the four corners of the square, to map an image onto this square, a corresponding uv coordinate must also be defined. In this case, the top-left corner has $xy=(-0.5, 0.5)$ and $uv=(0,1)$, the top-right corner has $xy=(0.5, 0.5)$ and $uv=(1, 1)$, and so on. Given this definition, it is possible to compute a unique uv value for any position inside the square by linearly interpolating the uv values defined at the vertices. For example, given the settings shown in Figure 5-2, you know that the midpoint along the top edge of the square maps to a uv of $(0.5, 1.0)$ in Texture Space, the midpoint along the left edge maps to a uv of $(0, 0.5)$, and so on.

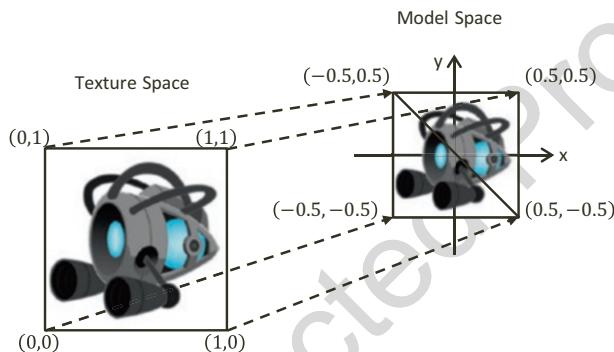
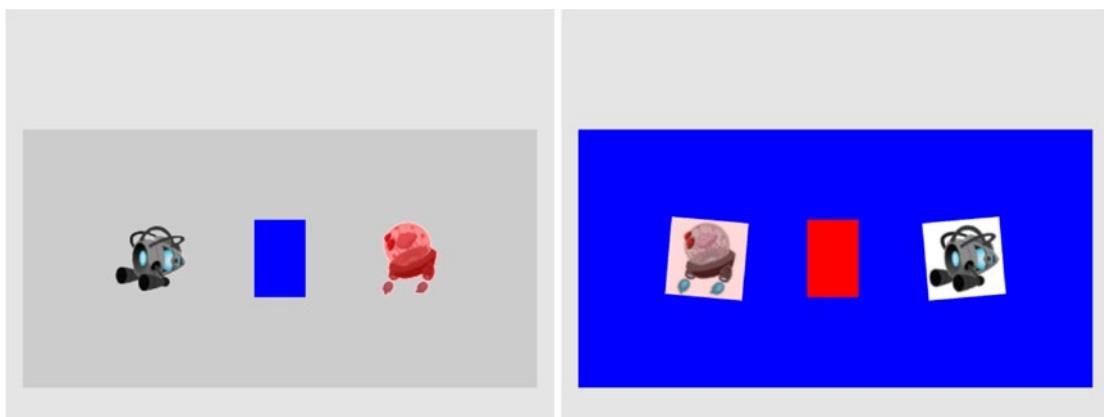


Figure 5-2. Defining Texture Space uv values to map the entire image onto the geometry in Model Space

The Texture Shaders Project

This project demonstrates the loading, rendering, and unloading of textures with WebGL. You can see an example of this project running in Figure 5-3 with the left and right screenshots from the two scenes implemented. Notice the natural-looking objects without white borders in the left screenshot and the images with white backgrounds in the right screenshot. This project will also highlight the differences between images with and without the alpha channel, or *transparency*. The source code to this project is defined in the chapter5/5.1.texture_shaders folder.



this figure will be printed in b/w

Figure 5-3. Running the Texture Shaders project with both scenes

The controls of the project are as follows, for both scenes:

- **Right-arrow key:** Moves the middle rectangle toward the right. If this rectangle passes the right window boundary, it will be wrapped to the left side of the window.
- **Left-arrow key:** Moves the middle rectangle toward the left. If this rectangle crosses the left window boundary, the game will transition to the next scene.

The goals of the project are as follows:

- To demonstrate how to define uv coordinates for geometries with WebGL
- To create a texture coordinate buffer in the graphics system with WebGL
- To build GLSL shaders to render the textured geometry
- To define the texture core engine component to load and process an image into a texture and to unload a texture
- To implement simple texture tinting, a modification of all texels with a programmer-specified color

You can find the following external resource files in the assets folder: a scene-level file (`blue_level.xml`) and four images (`minion_collector.jpg`, `minion_collector.png`, `minion_portal.jpg`, and `minion_portal.png`).

115 Overview

- 116 Creating and integrating textures involves relatively significant changes and new classes
117 to be added to the game engine. The following overview contextualizes and describes the
118 reasons for the changes:
- 119 • `texture_vs.glsl` and `texture_fs.glsl`: These are new files created
120 to define GLSL shaders for supporting drawing with uv coordinates.
121 Recall that the GLSL shaders must be loaded into WebGL and
122 compiled during the initialization of the game engine.
 - 123 • `vertex_buffer.js`: This file is modified to create a corresponding uv
124 coordinate buffer to define the texture coordinate for the vertices of
125 the unit square.
 - 126 • `texture_shader.js`: This is a new file that defines `TextureShader`
127 as a subclass of `SimpleShader` to interface the game engine with the
128 corresponding GLSL shaders (`TextureVS` and `TextureFS`).
 - 129 • `texture_renderable.js`: This is a new file that defines
130 `TextureRenderable` as a subclass of `Renderable` to facilitate the
131 creation, manipulation, and drawing of multiple instances of textured
132 objects.
 - 133 • `shader_resources.js`: Recall that this file defines a single instance
134 of `SimpleShader` to wrap over the corresponding GLSL shaders
135 to be shared system wide by all instances of `Renderable` objects.
136 In a similar manner, this file is modified to define an instance of
137 `TextureShader` to be shared by all instances of `TextureRenderable`
138 objects.
 - 139 • `gl.js`: This file is modified to configure WebGL to support drawing
140 with texture maps.
 - 141 • `texture.js`: This is a new file that defines the core engine
142 component that is capable of loading, activating (for rendering), and
143 unloading texture images.
 - 144 • `my_game.js` and `blue_level.js`: These game engine client files are
145 modified to test the new texture mapping functionality.

Two new source code folders, `src/engine/shaders` and `src/engine/renderables`,
 are created for organizing the engine source code. These folders are created in
 anticipation of the many new shader and renderer types required to support the
 corresponding texture-related functionality. Once again, continuous source code
 reorganization is important in supporting the corresponding increase in complexity. A
 systematic and logical source code structure is critical in maintaining and expanding the
 functionality of large software systems.

Extension of SimpleShader/Renderable Architecture

Recall that the `SimpleShader`/`Renderable` object pair is designed to support the
 loading of relevant game engine data to the `SimpleVS`/`FS` GLSL shaders and to support
 instantiating multiple copies of `Renderable` geometries by the game engine clients.
 As illustrated in Figure 5-4, the horizontal dotted line separates the game engine from
 WebGL. Notice that the GLSL shaders, `SimpleVS` and `SimpleFS`, are modules in WebGL
 and outside the game engine. The `SimpleShader` object maintains references to all
 attributes and uniform variables in the GLSL shaders and acts as the conduit for sending
 all transformation and vertex information to the `SimpleVS`/`FS` shaders. Although not
 depicted explicitly in Figure 5-4, there is only one instance of the `SimpleShader` object
 created in the game engine, in `shader_resources`, and this instance is shared by all
`Renderable` objects.

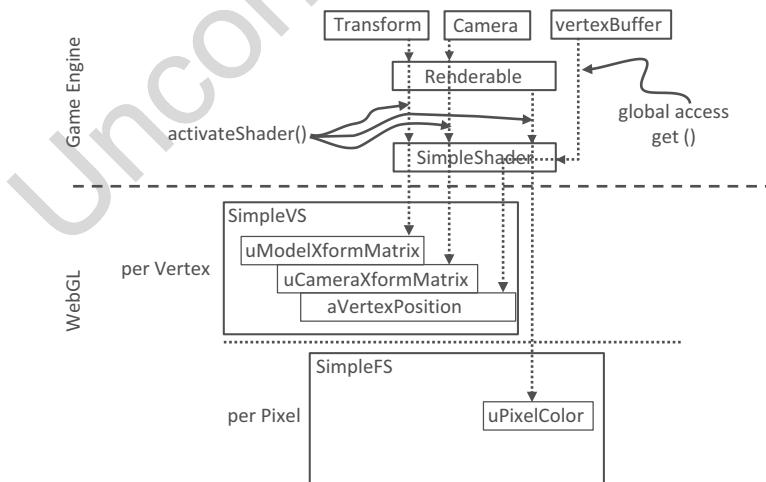


Figure 5-4. The `SimpleShader` and `Renderable` architecture

146
 147
 148
 149
 150
 151
 152

153
 154
 155
 156
 157
 158
 159
 160
 161
 162
 163
 164

this figure will be printed in b/w

165 The proper support of texture mapping demands new GLSL vertex and
 166 fragment shaders and thus requires that a corresponding shader and renderable
 167 object pair be defined in the game engine. As illustrated in Figure 5-5, both the
 168 GLSL TextureVS/FS shaders and TextureShader/TextureRenderable object
 169 pair are extensions (or subclasses) to the corresponding existing objects. The
 170 TextureShader/TextureRenderable object pair extends from the corresponding
 171 SimpleShader/Renderable objects to forward texture coordinates to the GLSL shaders.
 172 The TextureVS/FS shaders are extensions to the corresponding SimpleVS/FS shaders to
 173 read texels from the provided texture map when computing pixel colors. Note that since
 174 GLSL does not support subclassing, the TextureVS/FS source code is copied from the
 175 SimpleVS/FS files.

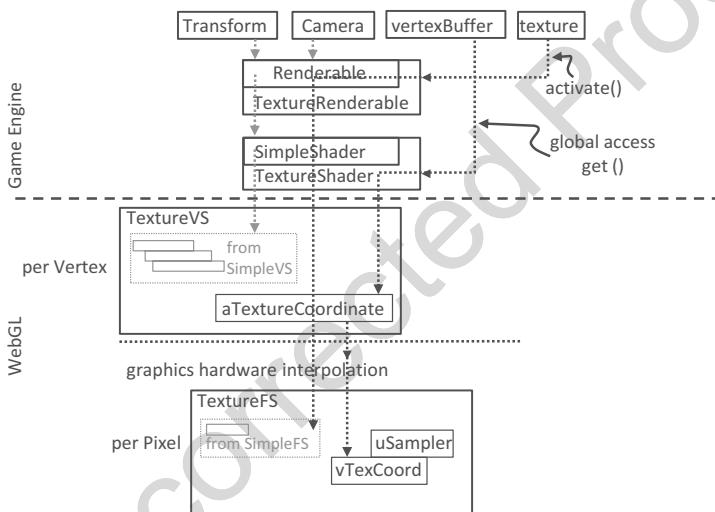


Figure 5-5. The TextureVS/FS GLSL shaders and the corresponding TextureShader/TextureRenderable object pair

176 GLSL Texture Shader

177 To support drawing with textures, you must create a shader that accepts both geometric (xy)
 178 and texture (uv) coordinates at each of the vertices. You will create new GLSL texture
 179 vertex and fragment shaders by copying and modifying the corresponding SimpleVS and
 180 SimpleFS programs. You can now begin to create the texture vertex shader.

- 181 1. Create a new file in the `src/glsl_shaders` folder and name it
 182 `texture_vs.glsl`.

2. Add the following code to the `texture_vs.glsL` file:

```

attribute vec3 aVertexPosition; // expects one vertex position          184
attribute vec2 aTextureCoordinate; // texture coordinate attribute      185

// texture coordinate that maps image to the square                  186
varying vec2 vTexCoord;                                              187

// to transform the vertex position                                 188
uniform mat4 uModelXformMatrix;                                     189
uniform mat4 uCameraXformMatrix;                                    190

void main(void) {                                                     191
    // Convert the vec3 into vec4 for scan conversion and           192
    // transform by uModelXformMatrix and uCameraXformMatrix before 193
    // assign to gl_Position to pass the vertex to the fragment shader 194
    gl_Position = uCameraXformMatrix *
                  uModelXformMatrix *
                  vec4(aVertexPosition, 1.0);                           195
    // pass the texture coordinate to the fragment shader            196
    vTexCoord = aTextureCoordinate;                                  197
}

```

You may notice that the `TextureVS` shader is similar to the `SimpleVS` shader, with only three additional lines of code:

- a. The first additional line adds the `aTextureCoordinate` attribute. This defines a vertex to include a `vec3` (`aVertexPosition`, the xyz position of the vertex) and a `vec2` (`aTextureCoordinate`, the uv coordinate of the vertex).
203
204
205
206
- b. The second declares the `varying vTexCoord` variable. The `varying` keyword in GLSL signifies that the associated variable will be linearly interpolated and passed to the fragment shader. As explained earlier and illustrated in Figure 5-2, uv values are defined only at vertex positions. In this case, the `varying vTexCoord` variable instructs the graphics hardware to linearly interpolate the uv values to compute the texture coordinate for each invocation of the fragment shader.
207
208
209
210
211
212
213
214

215 c. The third and final line assigns the vertex uv coordinate
216 values to the varying variable for interpolation and
217 forwarding to the fragment shader.

218 With the vertex shader defined, you can now create the associated fragment shader:

- 219 1. Create a new file in the `src/glsl_shaders` folder and name it
220 `texture_fs.glsl`.
- 221 2. Add the following code to the `texture_fs.glsl` file to declare
222 the variables. The `sampler2D` data type is a GLSL utility that is
223 capable of reading texel values from a 2D texture. In this case, the
224 `uSampler` object will be bound to a GLSL texture such that texel
225 values can be sampled for every pixel rendered. The `uPixelColor`
226 is the same as the one from `SimpleFS`. The `vTexCoord` is the
227 interpolated uv coordinate value for each pixel.

```
228 // The object that fetches data from texture.  
229 // Must be set outside the shader.  
230 uniform sampler2D uSampler;  
231 // Color of pixel  
232 uniform vec4 uPixelColor;  
233 // "varying" keyword signifies that the texture coordinate will be  
234 // interpolated and thus varies.  
235 varying vec2 vTexCoord;
```

- 236 3. Add the following code to compute the color for each pixel:

```
237 void main(void) {  
238     // texel color look up based on interpolated UV value in vTexCoord  
239     vec4 c = texture2D(uSampler, vec2(vTexCoord.s, vTexCoord.t));  
240     // tint the textured. transparent area defined by the texture  
241     vec3 r = vec3(c) * (1.0-uPixelColor.a) +
```

```

    vec3(uPixelColor) * uPixelColor.a;          242
    vec4 result = vec4(r, c.a);                  243
    gl_FragColor = result;                      244
}

```

The `texture2D()` function samples and reads the texel value from the texture that is associated with `uSampler` using the interpolated uv values from `vTexCoord`. In this example, the texel color is modified, or tinted, by a weighted sum of the color value defined in `uPixelColor` according to the *transparency* or the value of the corresponding alpha channel. In general, there is no agreed-upon definition for tinting texture colors. You are free to experiment with different ways to combine `uPixelColor` and the sampled texel color. For example, you can try multiplying the two. In the provided source code file, a few alternatives are suggested. Please do experiment with them.

Define and Set Up Texture Coordinates

Recall that all shaders share the same xy coordinate buffer of a unit square that is defined in the `vertex_buffer.js` file. In a similar fashion, a corresponding buffer must be defined to supply texture coordinates to the GLSL shaders.

1. Modify `vertex_buffer.js` to define both xy and uv coordinates for the unit square. As illustrated in Figure 5-2, the `mTextureCoordinates` variable defines the uv values for the corresponding four xy values of the unit square defined sequentially in `mVerticesOfSquare`. For example, (1, 1) are the uv values associated with the (0.5, 0.5, 0) xy position, (0, 1) for (-0.5, 0.5, 0), and so on.

```

// First: define the vertices for a square           268
let mVerticesOfSquare = [                         269
  0.5, 0.5, 0.0,                                270
  -0.5, 0.5, 0.0,                               271
  0.5, -0.5, 0.0,                               272
  -0.5, -0.5, 0.0                                273
];

```

CHAPTER 5 WORKING WITH TEXTURES, SPRITES, AND FONTS

```
275 // Second: define the corresponding texture coordinates
276 let mTextureCoordinates = [
277     1.0, 1.0,
278     0.0, 1.0,
279     1.0, 0.0,
280     0.0, 0.0
281 ];
282
283     2. Define the variable, mGLTextureCoordBuffer, to keep a reference
284     to the WebGL buffer storage for the texture coordinate values of
285     mTextureCoordinates and the corresponding getter function:
286
287     let mGLTextureCoordBuffer = null;
288     function getTexCoord() { return mGLTextureCoordBuffer; }
289
290     3. Modify the init() function to include a step D to initialize the
291     texture coordinates as a WebGL buffer. Notice the initialization
292     process is identical to that of the vertex xy coordinates
293     except that the reference to the new buffer is stored in
294     mGLTextureCoordBuffer and the transferred data are the uv
295     coordinate values.
296
297     function init() {
298         let gl = glSys.get();
299
300         ... identical to previous code ...
301
302         // Step D: Allocate and store texture coordinates
303         // Create a buffer on the gl context for texture coordinates
304         mGLTextureCoordBuffer = gl.createBuffer();
305
306         // Activate texture coordinate buffer
307         gl.bindBuffer(gl.ARRAY_BUFFER, mGLTextureCoordBuffer);
308
309         // Loads textureCoordinates into the mGLTextureCoordBuffer
310         gl.bufferData(gl.ARRAY_BUFFER,
311                         new Float32Array(mTextureCoordinates), gl.STATIC_DRAW);
312     }
```

4. Remember to release the allocated buffer during final cleanup:

```
function cleanUp() {  
    ... identical to previous code ...  
  
    if (mGLTextureCoordBuffer !== null) {  
        gl.deleteBuffer(mGLTextureCoordBuffer);  
        mGLTextureCoordBuffer = null;  
    }  
}
```

5. Finally, remember to export the changes:

```
export {init, cleanUp, get, getTexCoord}
```

Interface GLSL Shader to the Engine

Just as the SimpleShader object was defined to interface to the SimpleVS and SimpleFS shaders, a corresponding shader object needs to be created in the game engine to interface to the TextureVS and TextureFS GLSL shaders. As mentioned in the overview of this project, you will also create a new folder to organize the growing number of different shaders.

- Create a new folder called shaders in src/engine. Move the simple_shader.js file into this folder, and do not forget to update the reference path in index.js.
- Create a new file in the src/engine/shaders folder and name it texture_shader.js.

```
class TextureShader extends SimpleShader {  
    constructor(vertexShaderPath, fragmentShaderPath) {  
        // Call super class constructor  
        super(vertexShaderPath, fragmentShaderPath);  
  
        // reference to aTextureCoordinate within the shader  
        this.mTextureCoordinateRef = null;  
  
        // get the reference of aTextureCoordinate within the shader  
        let gl = glSys.get();  
        this.mTextureCoordinateRef = gl.getAttribLocation(  
            this.mTextureCoordinateRef);  
    }  
}
```

```

335                     this.mCompiledShader,
336                     "aTextureCoordinate");
337             this.mSamplerRef = gl.getUniformLocation(this.mCompiledShader,
338                                         "uSampler");
339         }
340         ... implementation to follow ...

```

341 In the listed code, take note of the following:

- 342 a. The defined TextureShader class is an extension, or
343 subclass, to the SimpleShader class.
 - 344 b. The constructor implementation first calls super(), the
345 constructor of SimpleShader. Recall that the SimpleShader
346 constructor will load and compile the GLSL shaders
347 defined by the vertexShaderPath and fragmentShaderPath
348 parameters and set mVertexPositionRef to reference the
349 aVertexPosition attribute defined in the shader.
 - 350 c. In the rest of the constructor, the mTextureCoordinateRef
351 keeps a reference to the aTextureCoordinate attribute
352 defined in the texture_vs.glsl.
 - 353 d. In this way, both the vertex position (aVertexPosition) and
354 texture coordinate (aTextureCoordinate) attributes are
355 referenced by a JavaScript TextureShader object.
- 356 3. Override the activate() function to enable the texture coordinate
357 data. The superclass super.activate() function sets up the xy
358 vertex position and passes the values of pixelColor, trsMatrix,
359 and cameraMatrix to the shader. The rest of the code binds
360 mTextureCoordinateRef, the texture coordinate buffer defined in
361 the vertex_buffer module, to the aTextureCoordinate attribute in
362 the GLSL shader and mSampler to texture unit 0 (to be detailed later).

```

363 // Overriding the Activation of the shader for rendering
364 activate(pixelColor, trsMatrix, cameraMatrix) {
365     // first call the super class's activate
366     super.activate(pixelColor, trsMatrix, cameraMatrix);

```

```

// now our own functionality: enable texture coordinate array      367
let gl = glSys.get();                                         368
gl.bindBuffer(gl.ARRAY_BUFFER, this._getTexCoordBuffer());       369
gl.vertexAttribPointer(this.mTextureCoordinateRef, 2,           370
                      gl.FLOAT, false, 0, 0);                         371
gl.enableVertexAttribArray(this.mTextureCoordinateRef);        372
// bind uSampler to texture 0                                     373
gl.uniform1i(this.mSamplerRef, 0);                           374
    // texture.activateTexture() binds to Texture0            375
}

```

With the combined functionality of SimpleShader and TextureShader, after the activate() function call, both of the attribute variables (aVertexPosition and aTextureCoordinate) in the GLSL texture_vs shader are connected to the corresponding buffers in the WebGL memory.

Facilitate Sharing with shader_resources

In the same manner that SimpleShader is a reusable resource, only one instance of the TextureShader needs to be created, and this instance can be shared. The shader_resources module should be modified to reflect this.

1. In shader_resources.js, add the variables to hold a texture shader:

```

// Texture Shader
let kTextureVS = "src/glsl_shaders/texture_vs.glsl"; // VertexShader 386
let kTextureFS = "src/glsl_shaders/texture_fs.glsl"; // FragmentShader 387
let mTextureShader = null;                            388

```

2. Define a function to retrieve the texture shader:

```
function getTextureShader() { return mTextureShader; }
```

3. Create the instance of texture shader in the createShaders() function:

```

function createShaders() {
    mConstColorShader = new SimpleShader(kSimpleVS, kSimpleFS);
    mTextureShader = new TextureShader(kTextureVS, kTextureFS);          395
}

```

CHAPTER 5 WORKING WITH TEXTURES, SPRITES, AND FONTS

397 4. Modify the `init()` function to append the `loadPromise` to include
398 the loading of the texture shader source files:

```
399   function init() {  
400     let loadPromise = new Promise(  
401       async function(resolve) {  
402         await Promise.all([  
403           text.load(kSimpleFS),  
404           text.load(kSimpleVS),  
405           text.load(kTextureFS),  
406           text.load(kTextureVS)  
407         ]);  
408         resolve();  
409       }).then(  
410         function resolve() { createShaders(); }  
411       );  
412       map.pushPromise(loadPromise);  
413   }
```

414 5. Remember to release newly allocated resources during cleanup:

```
415   function cleanUp() {  
416     mConstColorShader.cleanUp();  
417     mTextureShader.cleanUp();  
  
418     text.unload(kSimpleVS);  
419     text.unload(kSimpleFS);  
420     text.unload(kTextureVS);  
421     text.unload(kTextureFS);  
422   }
```

423 6. Lastly, remember to export the newly defined functionality:

```
424   export {init, cleanUp, getConstColorShader, getTextureShader}
```

TextureRenderable Class

425

Just as the Renderable class encapsulates and facilitates the definition and drawing of multiple instances of SimpleShader objects, a corresponding TextureRenderable class needs to be defined to support the drawing of multiple instances of TextureShader objects.

426

427

428

429

Changes to the Renderable Class

430

As mentioned in the project overview, for the same reason as creating and organizing shader classes in the Shaders folder, a renderables folder should be created to organize the growing number of different kinds of Renderable objects. In addition, the Renderable class must be modified to support it being the base class of all Renderable objects.

431

432

433

434

435

1. Create the `src/engine/renderables` folder and move `renderable.js` into this folder. Remember to update `index.js` to reflect the file location change. 436
437
438
2. Define the `_setShader()` function to set the shader for the Renderable. This is a protected function which allows subclasses to modify the `mShader` variable to refer to the appropriate shaders for each corresponding subclass. 439
440
441
442

```
// this is private/protected
_setShader(s) { this.mShader = s; }
```

443

444

Note Functions with names that begin with “`_`” are either private or protected and should not be called from outside of the class. This is a convention followed in this book and not enforced by JavaScript.

445

446

447

Define the TextureRenderable Class

448

You are now ready to define the TextureRenderable class. As noted, TextureRenderable is derived from and extends the Renderable class functionality to render texture mapped objects.

449

450

451

CHAPTER 5 WORKING WITH TEXTURES, SPRITES, AND FONTS

- 452 1. Create a new file in the `src/engine/renderables` folder and
453 name it `texture_renderable.js`. Add the constructor. Recall
454 that `super()` is a call to the superclass (`Renderable`) constructor;
455 similarly, the `super.setColor()` and `super._setShader()`
456 are calls to the superclass functions. As will be detailed when
457 discussing the engine texture resource module, the `myTexture`
458 parameter is the path to the file that contains the texture image.

```
459 class TextureRenderable extends Renderable {  
460     constructor(myTexture) {  
461         super();  
462         super.setColor([1, 1, 1, 0]); // Alpha 0: no texture tinting  
463         super._setShader(shaderResources.getTextureShader());  
464         this.mTexture = myTexture; // cannot be a "null"  
465     }  
466     ... implementation to follow ...
```

- 467 2. Define a `draw()` function to append the function defined in the
468 `Renderable` class to support textures. The `texture.activate()`
469 function activates and allows drawing with the specific texture.
470 The details of this function will be discussed in the following
471 section.

```
472     draw(camera) {  
473         // activate the texture  
474         texture.activate(this.mTexture);  
475         super.draw(camera);  
476     }
```

- 477 3. Define a getter and setter for the texture reference:

```
478     getTexture() { return this.mTexture; }  
479     setTexture(newTexture) { this.mTexture = newTexture; }
```

- 480 4. Finally, remember to export the class:

```
481     export default TextureRenderable;
```

Texture Support in the Engine

To support drawing with textures, the rest of the game engine requires two main modifications: WebGL context configuration and a dedicated engine component to support operations associated with textures.

Configure WebGL to Support Textures

The configuration of WebGL context must be updated to support textures. In `gl.js`, update the `init()` function according to the following:

```
function init(htmlCanvasID) {  
    mCanvas = document.getElementById(htmlCanvasID);  
    if (mCanvas == null)  
        throw new Error("Engine init [" +  
                        htmlCanvasID + "] HTML element id not found");  
  
    // the standard or experimental webgl and binds to the Canvas area  
    // store the results to the instance variable mGL  
    mGL = mCanvas.getContext("webgl2", {alpha: false}) ||  
          mCanvas.getContext("experimental-webgl2", {alpha: false});  
  
    if (mGL === null) {  
        document.write("<br><b>WebGL 2 is not supported!</b>");  
        return;  
    }  
  
    // Allows transparency with textures.  
    mGL.blendFunc(mGL.SRC_ALPHA, mGL.ONE_MINUS_SRC_ALPHA);  
    mGL.enable(mGL.BLEND);  
  
    // Set images to flip y axis to match the texture coordinate space.  
    mGL.pixelStorei(mGL.UNPACK_FLIP_Y_WEBGL, true);  
}
```

The parameter passed to `mCanvas.getContext()` informs the browser that the canvas should be opaque. This can speed up the drawing of transparent content and images. The `blendFunc()` function enables transparencies when drawing images with the alpha channel. The `pixelStorei()` function defines the origin of the uv coordinate to be at the lower-left corner.

513 Create the Texture Resource Module

514 Similar to text and audio files, a new engine component must be defined to support the
 515 corresponding texture operations including loading from the server file system, storing
 516 via the WebGL context to the GPU memory, activating the texture buffer for drawing, and
 517 removing from the GPU:

- 518 1. Create a new file in the `src/engine/resources` folder and name
 519 it `texture.js`. This file will implement the Textures engine
 520 component.
- 521 2. Define the `TextureInfo` class to represent a texture in the game
 522 engine. The `mWidth` and `mHeight` are the pixel resolution of the
 523 texture image, and `mGLTexID` is a reference to the WebGL texture
 524 storage.

```
525 class TextureInfo {
526   constructor(w, h, id) {
527     this.mWidth = w;
528     this.mHeight = h;
529     this.mGLTexID = id;
530   }
531 }
```

532 **Note** For efficiency reasons, many graphics hardware only supports texture with
 533 image resolutions that are in powers of 2, such as 2×4 ($2^1 \times 2^2$), or 4×16 ($2^2 \times 2^4$), or
 534 64×256 ($2^6 \times 2^8$), and so on. This is also the case for WebGL. All examples in this
 535 book only work with textures with resolutions that are powers of 2.

- 536 3. Import the core resource management functionality from the
 537 `resource_map`:

```
538 import * as map from "../core/resource_map.js";
539 // functions from resource_map
540 let has = map.has;
541 let get = map.get;
```

4. Define a function to load an image asynchronously as a promise and push the promise to be part of the pending promises in the map. Distinct from the text and audio resources, JavaScript Image API supports straightforward image file loading, and the map. loadDecideParse() is not required in this case. Once an image is loaded, it is passed to the processLoadedImage() function with its file path as the name. 542
 543
 544
 545
 546
 547
 548

```
// Loads a texture so that it can be drawn. 549
function load(textureName) { 550
  let image = new Image(); 551
  let texturePromise = new Promise( 552
    function(resolve) { 553
      image.onload = resolve; 554
      image.src = textureName; 555
    }).then( 556
      function resolve() { 557
        processLoadedImage(textureName, image); } 558
    ); 559
  map.pushPromise(texturePromise); 560
  return texturePromise; 561
} 562
```

5. Add an unload() function to clean up the engine and release WebGL resources: 563
 564

```
// Remove the reference to allow associated memory 565
// to be available for subsequent garbage collection 566
function unload(textureName) { 567
  let texInfo = get(textureName); 568
  if (map.unload(textureName)) { 569
    let gl = glSys.get(); 570
    gl.deleteTexture(texInfo.mGLTexID); 571
  } 572
} 573
```

CHAPTER 5 WORKING WITH TEXTURES, SPRITES, AND FONTS

- 574 6. Now define the `processLoadedImage()` function to convert the
575 format of an image and store it to the WebGL context. The
576 `gl.createTexture()` function creates a WebGL texture buffer
577 and returns a unique ID. The `texImage2D()` function stores the
578 image into the WebGL texture buffer, and `generateMipmap()`
579 computes a mipmap for the texture. Lastly, a `TextureInfo` object
580 is instantiated to refer to the WebGL texture and stored into the
581 `resource_map` according to the file path to the texture image file.

```
582 function processLoadedImage(path, image) {  
583     let gl = glSys.get();  
584     // Generate a texture reference to the webGL context  
585     let textureID = gl.createTexture();  
586     // binds texture reference with current texture in the webGL  
587     gl.bindTexture(gl.TEXTURE_2D, textureID);  
588     // Loads texture to texture data structure with descriptive info.  
589     // Parameters:  
590     // 1: "binding point" or target the texture is being loaded to.  
591     // 2: Level of detail. Used for mipmapping. 0 is base texture level.  
592     // 3: Internal format. The composition of each element. i.e. pixels.  
593     // 4: Format of texel data. Must match internal format.  
594     // 5: The data type of the texel data.  
595     // 6: Texture Data.  
596     gl.texImage2D(gl.TEXTURE_2D, 0,  
597                   gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE, image);  
598     // Creates a mipmap for this texture.  
599     gl.generateMipmap(gl.TEXTURE_2D);  
600     // Tells WebGL done manipulating data at the mGL.TEXTURE_2D target.  
601     gl.bindTexture(gl.TEXTURE_2D, null);  
602     let texInfo = new TextureInfo(image.naturalWidth,  
603                                   image.naturalHeight, textureID);  
604     map.set(path, texInfo);  
605 }
```

Note A *mipmap* is a representation of the texture image that facilitates high-quality rendering. Please consult a computer graphics reference book to learn more about mipmap representation and the associated texture mapping algorithms.

7. Define a function to activate a WebGL texture for drawing: 610

```
function activate(textureName) { 611
    let gl = glSys.get(); 612
    let texInfo = get(textureName); 613

    // Binds texture reference to the current webGL texture functionality 614
    gl.activeTexture(gl.TEXTURE0);
    gl.bindTexture(gl.TEXTURE_2D, texInfo.mGLTexID);

    // To prevent texture wrapping 617
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.CLAMP_TO_EDGE);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.CLAMP_TO_EDGE);

    // Handles how magnification and minimization filters will work. 620
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.LINEAR);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER,
                    gl.LINEAR_MIPMAP_LINEAR);

    // For the texture to look "sharp" do the following: 624
    // gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
    // gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
}
```

a. The `get()` function locates the `TextureInfo` object from the `resource_map` based on the `textureName`. The located `mGLTexID` is used in the `bindTexture()` function to activate the corresponding WebGL texture buffer for rendering. 628

- 632 b. The `texParameteri()` function defines the rendering
633 behavior for the texture. The `TEXTURE_WRAP_S/T` parameters
634 ensure that the texel values will not wrap around at the
635 texture boundaries. The `TEXTURE_MAG_FILTER` parameter
636 defines how to magnify a texture, in other words, when a
637 low-resolution texture is rendered to many pixels in the game
638 window. The `TEXTURE_MIN_FILTER` parameter defines how to
639 minimize a texture, in other words, when a high-resolution
640 texture is rendered to a small number of pixels.
- 641 c. The `LINEAR` and `LINEAR_MIPMAP_LINEAR` configurations
642 generate smooth textures by blurring the details of the original
643 images, while the commented out `NEAREST` option will result in
644 unprocessed textures best suitable for pixelated effects. Notice
645 that in this case, color boundaries of the texture image may
646 appear jagged.

647 **Note** In general, it is best to use texture images with similar resolution as the
648 number of pixels occupied by the objects in the game. For example, a square that
649 occupies a 64×64 pixel space should ideally use a 64×64 pixel texture.

- 650 8. Define a function to deactivate a texture as follows. This function
651 sets the WebGL context to a state of not working with any texture.

```
652 function deactivate() {  
653     let gl = glSys.get();  
654     gl.bindTexture(gl.TEXTURE_2D, null);  
655 }
```

- 656 9. Finally, remember to export the functionality:

```
657 export {has, get, load, unload,  
658         TextureInfo,  
659         activate, deactivate}
```

Export New Functionality to the Client

The last step in integrating texture functionality into the engine involves modifying the engine access file, `index.js`. Edit `index.js` and add in the following import and export statements to grant the client access to the texture resource module and the `TextureRenderable` class:

```
... identical to previous code ... 660
import * as texture from "./resources/texture.js"; 661
// renderables 662
import Renderable from "./renderables/renderable.js"; 663
import TextureRenderable from "./renderables/texture_renderable.js"; 664
... identical to previous code ... 665
export default { 666
    // resource support 667
    audio, text, xml, texture, 668
    // input support 669
    input, 670
    // Util classes 671
    Camera, Scene, Transform, 672
    // Renderables 673
    Renderable, TextureRenderable, 674
    // functions 675
    init, cleanUp, clearCanvas 676
} 677
```

Testing of Texture Mapping Functionality

With the described modifications, the game engine can now render constant color objects as well as objects with interesting and different types of textures. The following testing code is similar to that from the previous example where two scenes, `MyGame` and `BlueLevel`, are used to demonstrate the newly added texture mapping functionality. The main modifications include the loading and unloading of texture images and the creation and drawing of `TextureRenderable` objects. In addition, the `MyGame` scene

690 highlights transparent texture maps with alpha channel using PNG images, and the
 691 BlueScene scene shows corresponding textures with images in the JPEG format.

692 As in all cases of building a game, it is essential to ensure that all external resources
 693 are properly organized. Recall that the assets folder is created specifically for the
 694 organization of external resources. Take note of the four new texture files located in the
 695 assets folder: `minion_collector.jpg`, `minion_collector.png`, `minion_portal.jpg`,
 696 and `minion_portal.png`.

697 Modify the BlueLevel Scene File to Support Textures

698 The `blue_level.xml` scene file is modified from the previous example to support texture
 699 mapping:

```
700 <MyGameLevel>

701     <!-- cameras -->
702         <!-- Viewport: x, y, w, h -->
703     <Camera CenterX="20" CenterY="60" Width="20"
704             Viewport="20 40 600 300"
705             BgColor="0 0 1 1.0"/>

706     <!-- The red rectangle -->
707     <Square PosX="20" PosY="60" Width="2" Height="3"
708             Rotation="0" Color="1 0 0 1" />

709     <!-- Textures Square -->
710     <TextureSquare PosX="15" PosY="60" Width="3" Height="3"
711             Rotation="-5" Color="1 0 0 0.3"
712             Texture="assets/minion_portal.jpg" />

713     <TextureSquare PosX="25" PosY="60" Width="3" Height="3"
714             Rotation="5" Color="0 0 0 0"
715             Texture="assets/minion_collector.jpg"/>
716         <!-- without tinting, alpha should be 0 -->

717 </MyGameLevel>
```

718 The `TextureSquare` element is similar to `Square` with the addition of a `Texture`
 719 attribute that specifies which image file should be used as a texture map for the square.
 720 Note that as implemented in `texture_fs.gls1`, the alpha value of the `Color` element is

used for tinting the texture map. The XML scene description is meant to support slight tinting of the `minion_portal.jpg` texture and no tinting of the `minion_collector.jpg` texture. This texture tinting effect can be observed in the right image of Figure 5-3. In addition, notice that both images specified are in the JPEG format. Since the JPEG format does not support the storing of alpha channel, the unused regions of the two images show up as white areas outside the portal and collector minions in the right image of Figure 5-3.

Modify SceneFileParser

The scene file parser, `scene_file_parser.js`, is modified to support the parsing of the updated `blue_scene.xml`, in particular, to parse `Square` elements into `Renderable` objects and `TextureSquare` elements into `TextureRenderable` objects. For details of the changes, please refer to the source code file in the `src/my_game/util` folder.

Test BlueLevel with JPEGs

The modifications to `blue_level.js` are in the constructor, `load()`, `unload()`, `next()`, and `init()` functions where the texture images are loaded and unloaded and new `TextureRenderable` objects are parsed:

1. Edit `blue_level.js` and modify the constructor to define constants to represent the texture images:

```
class BlueLevel extends engine.Scene {
    constructor() {
        super();
        // scene file name
        this.kSceneFile = "assets/blue_level.xml";

        // textures: (Note: jpg does not support transparency)
        this.kPortal = "assets/minion_portal.jpg";
        this.kCollector = "assets/minion_collector.jpg";

        // all squares
        this.mSqSet = [];           // these are the Renderable objects

        // The camera to view the scene
        this.mCamera = null;
    }
    ... implementation to follow ...
}
```

CHAPTER 5 WORKING WITH TEXTURES, SPRITES, AND FONTS

752 2. Initiate loading of the textures in the load() function:

```
753     load() {  
754         // load the scene file  
755         engine.xml.load(this.kSceneFile);  
  
756         // load the textures  
757         engine.texture.load(this.kPortal);  
758         engine.texture.load(this.kCollector);  
759     }
```

760 3. Likewise, add code to clean up by unloading the textures in the
761 unload() function:

```
762     unload() {  
763         // unload the scene file and loaded resources  
764         engine.xml.unload(this.kSceneFile);  
765         engine.texture.unload(this.kPortal);  
766         engine.texture.unload(this.kCollector);  
767     }
```

768 4. Support loading of the next scene with the next() function:

```
769     next() {  
770         super.next();  
771         let nextLevel = new MyGame(); // load the next level  
772         nextLevel.start();  
773     }
```

774 5. Parse the textured squares in the init() function:

```
775     init() {  
776         let sceneParser = new SceneFileParser(this.kSceneFile);  
  
777         // Step A: Read in the camera  
778         this.mCamera = sceneParser.parseCamera();
```

```
// Step B: Read all the squares and textureSquares  
sceneParser.parseSquares(this.mSqSet);  
sceneParser.parseTextureSquares(this.mSqSet);  
}
```

6. Include appropriate code in the update() function to continuously change the tinting of the portal TextureRenderable, as follows:

```
update() {  
    ... identical to previous code ...  
  
// continuously change texture tinting  
let c = this.mSqSet[1].getColor();  
let ca = c[3] + deltaX;  
if (ca > 1) {  
    ca = 0;  
}  
c[3] = ca;  
}
```

- a. Index 1 of mSqSet is the portal TextureRenderable object, and index 3 of the color array is the alpha channel.
- b. The listed code continuously increases and wraps the alpha value of the mColor variable in the TextureRenderable object. Recall that the values of this variable are passed to TextureShader and then loaded to the uPixelColor of TextureFS for tinting the texture map results.
- c. As defined in the first TextureSquare element in the blue_scene.xml file, the color defined for the portal object is red. For this reason, when running this project, in the blue level, the portal object appears to be blinking in red.

806 Test MyGame with PNGs

807 Similar to the BlueLevel scene, MyGame is a straightforward modification of the
808 previous example with changes to load and unload texture images and to create
809 TextureRenderable objects:

- 810 1. Edit `my_game.js`; modify the `MyGame` constructor to define
811 texture image files and the variables for referencing the
812 `TextureRenderable` objects:

```
813 class MyGame extends engine.Scene {  
814     constructor() {  
815         super();  
  
816         // textures:  
817         this.kPortal = "assets/minion_portal.png"; // with transparency  
818         this.kCollector = "assets/minion_collector.png";  
  
819         // The camera to view the scene  
820         this.mCamera = null;  
  
821         // the hero and the support objects  
822         this.mHero = null;  
823         this.mPortal = null;  
824         this.mCollector = null;  
825     }  
826 }
```

- 826 2. Initiate the loading of the textures in the `load()` function:

```
827 load() {  
828     // loads the textures  
829     engine.texture.load(this.kPortal);  
830     engine.texture.load(this.kCollector);  
831 }
```

3. Make sure you remember to unload the textures in `unload()`:

```
unload() {  
    // Game loop not running, unload all assets  
    engine.texture.unload(this.kPortal);  
    engine.texture.unload(this.kCollector);  
}  
833  
834  
835  
836  
837
```

4. Define the `next()` function to start the blue level:

```
next() {  
    super.next();  
    // starts the next level  
    let nextLevel = new BlueLevel(); // next level to be loaded  
    nextLevel.start();  
}  
839  
840  
841  
842  
843  
844
```

5. Create and initialize the `TextureRenderables` objects in the `init()` function:

```
init() {  
    // Step A: set up the cameras  
    this.mCamera = new engine.Camera(  
        vec2.fromValues(20, 60), // position of the camera  
        20, // width of camera  
        [20, 40, 600, 300] // viewport (X, Y, width, height)  
    );  
    this.mCamera.setBackgroundColor([0.8, 0.8, 0.8, 1]);  
    // sets the background to gray  
  
    // Step B: Create the game objects  
    this.mPortal = new engine.TextureRenderable(this.kPortal);  
    this.mPortal.setColor([1, 0, 0, 0.2]); // tints red  
    this.mPortal.getXform().setPosition(25, 60);  
    this.mPortal.getXform().setSize(3, 3);  
  
    this.mCollector = new engine.TextureRenderable(this.kCollector);  
    this.mCollector.setColor([0, 0, 0, 0]); // No tinting  
    this.mCollector.getXform().setPosition(15, 60);  
    this.mCollector.getXform().setSize(3, 3);  
}  
845  
846  
847  
848  
849  
850  
851  
852  
853  
854  
855  
856  
857  
858  
859  
860  
861  
862  
863  
864
```

```

865 // Step C: Create the hero object in blue
866 this.mHero = new engine.Renderable();
867 this.mHero.setColor([0, 0, 1, 1]);
868 this.mHero.getXform().setPosition(20, 60);
869 this.mHero.getXform().setSize(2, 3);
870 }

```

871 Remember that the texture file path is used as the unique identifier
 872 in the `resource_map`. For this reason, it is essential for file texture
 873 loading and unloading and for the creation of `TextureRenderable`
 874 objects to refer to the same file path. In the given code, all three
 875 functions refer to the same constants defined in the constructor.

- 876 6. The modification to the `draw()` function draws the two new
 877 `TextureRenderable` objects by calling their corresponding `draw()`
 878 functions, while the modification to the `update()` function is
 879 similar to that of the `BlueLevel` discussed earlier. Please refer to
 880 the `my_game.js` source code file in the `src/my_game` folder for
 881 details.

882 When running the example for this project in the `chapter5/5.1.texture_shaders`
 883 folder, once again take note of the results of continuously changing the texture tinting—
 884 the blinking of the portal minion in red. In addition, notice the differences between the
 885 PNG-based textures in the `MyGame` level and the corresponding JPEG ones with white
 886 borders in the `BlueLevel`. It is visually more pleasing and accurate to represent objects
 887 using textures with the alpha (or transparency) channel. PNG is one of the most popular
 888 image formats that supports the alpha channel.

889 Observations

890 This project has been the longest and most complicated one that you have worked
 891 with. This is because working with texture mapping requires you to understand texture
 892 coordinates, the implementation cuts across many of the files in the engine, and the
 893 fact that actual images must be loaded, converted into textures, and stored/accessed
 894 via WebGL. To help summarize the changes, Figure 5-6 shows the game engine states in
 895 relation to the states of an image used for texture mapping and some of the main game
 896 engine operations.

The left column of Figure 5-6 identifies the main game engine states, from WebGL initialization to the initialization of a scene, to the game loop, and to the eventual unloading of the scene. The middle column shows the corresponding states of an image that will be used as a texture. Initially, this image is stored on the server file system. During the scene initialization, the Scene.load() function will invoke the engine/resources/texture.load() function to load the image and cause the loaded image to be processed by the engine/resources/texture.processLoadedImage() function into a corresponding WebGL texture to be stored in the GPU texture buffer. During the game loop cycle, the TextureRenderable.draw() function activates the appropriate WebGL texture via the engine/resources/texture.activate() function. This enables the corresponding GLSL fragment shader to sample from the correct texture during rendering. Finally, when a texture is no longer needed by the game, the Scene.unload() function will call engine/resources/texture.unload() to remove the loaded image from the system.

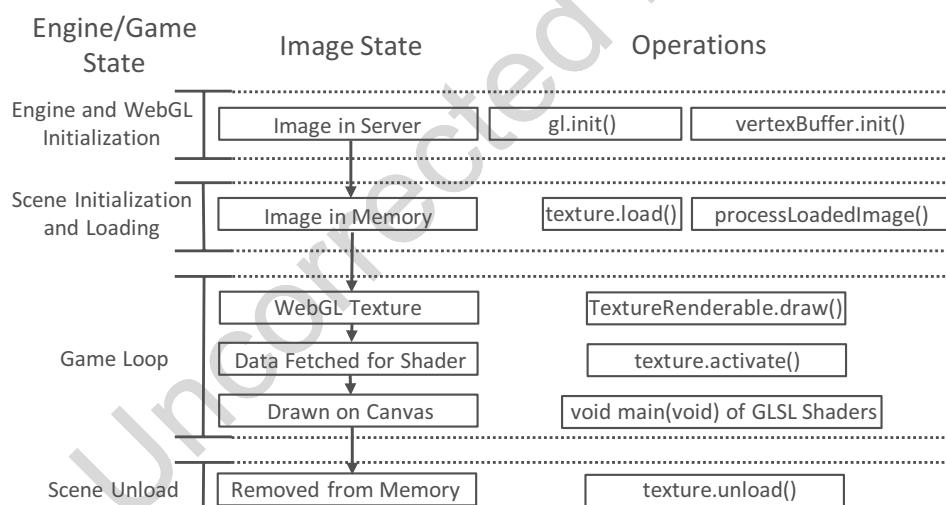


Figure 5-6. Overview of the states of an image file and the corresponding WebGL texture

911

Drawing with Sprite Sheets

912 As described earlier, a sprite sheet is an image that is composed of multiple lower-
 913 resolution images that individually represent different objects. Each of these individual
 914 images is referred to as a *sprite sheet element*. For example, Figure 5-7 is a sprite sheet
 915 with 13 elements from 4 different objects. Each of the top two rows contains five
 916 elements of the same object in different animated positions, and in the last row, there
 917 are three elements of different objects: the character Dye, the portal minion, and the
 918 collector minion. The artist or software program that created the sprite sheet must
 919 communicate the pixel locations of each sprite element to the game developer, in much
 920 the same way as illustrated in Figure 5-7.

this figure will be printed in b/w

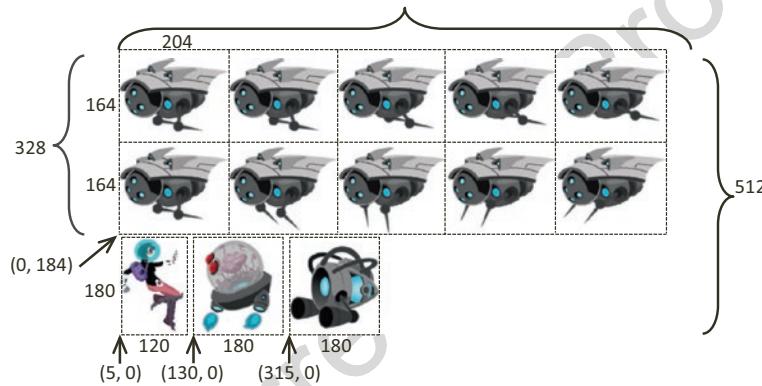


Figure 5-7. Example sprite sheet: *minion_sprite.png* composed of lower-resolution images of different objects

921 Sprite sheets are defined to optimize both memory and processing requirements. For
 922 example, recall that WebGL only supports textures that are defined by images with $2^x \times 2^y$
 923 resolutions. This requirement means that the Dye character at a resolution of 120x180
 924 must be stored in a 128x256 ($2^7 \times 2^8$) image in order for it to be created as a WebGL
 925 texture. Additionally, if the 13 elements of Figure 5-7 were stored as separate image files,
 926 then it would mean 13 slow file system accesses would be required to load all the images,
 927 instead of one single system access to load the sprite sheet.

928 The key to working with a sprite sheet and the associated elements is to remember
 929 that the texture coordinate uv values are defined over the 0 to 1 normalized range
 930 regardless of the actual image resolution. For example, Figure 5-8 focuses on the uv

values of the collector minion in Figure 5-7, the rightmost sprite element on the third row. The top, center, and bottom rows of Figure 5-8 show coordinate values of the portal element.

- **Pixel positions:** The lower-left corner is (315, 0), and the upper-right corner is (495, 180). 934
935
- **UV values:** The lower-left corner is (0.308, 0.0), and the upper-right corner is (0.483, 0.352). 936
937
- **Use in Model Space:** Texture mapping of the element is accomplished by associating the corresponding uv values with the xy values at each vertex position. 938
939
940

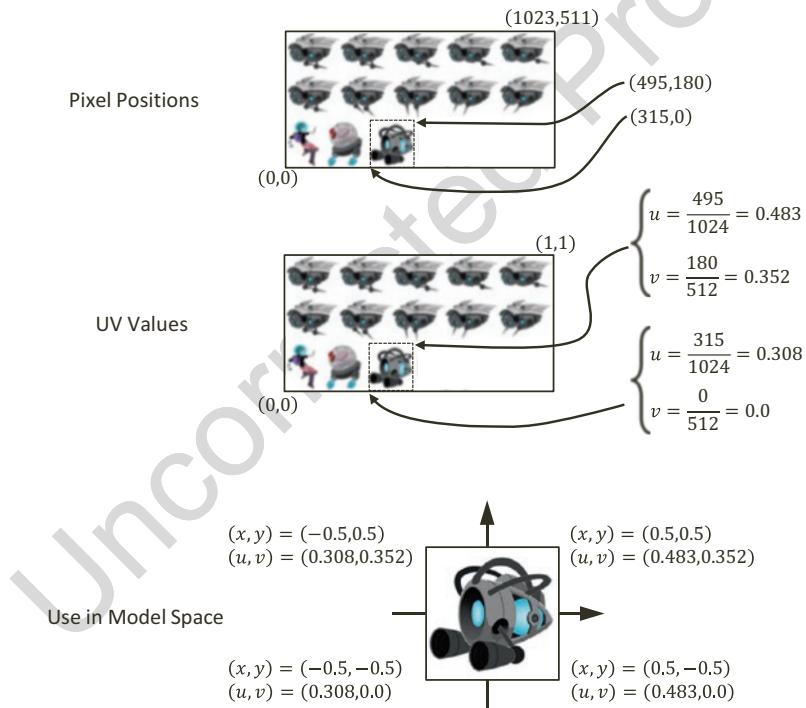


Figure 5-8. A conversion of coordinate from pixel position to uv values and used for mapping on geometry

941 The Sprite Shaders Project

942 This project demonstrates how to draw objects with sprite sheet elements by defining
 943 appropriate abstractions and classes. You can see an example of this project running
 944 in Figure 5-9. The source code to this project is defined in the chapter5/5.2.sprite_
 945 shaders folder.

this figure will be printed in b/w

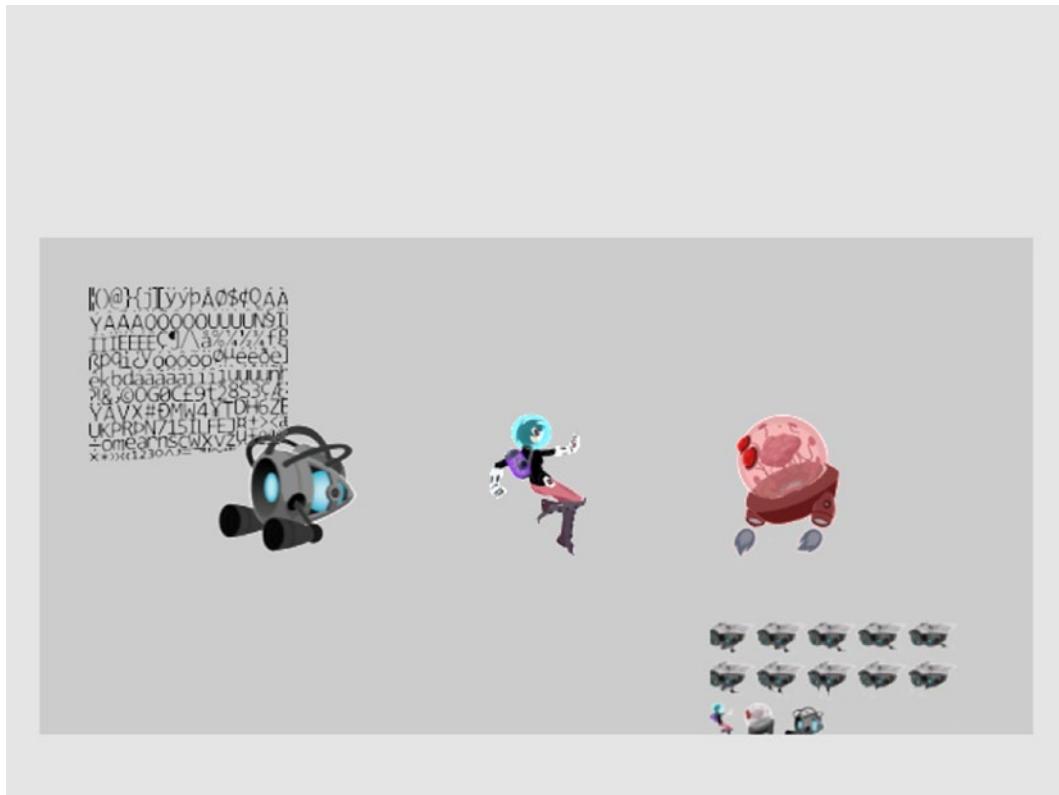


Figure 5-9. Running the Sprite Shaders project

946 The controls of the project are as follows:

- 947 • **Right-arrow key:** Moves the Dye character (the hero) right and loops
 948 to the left boundary when the right boundary is reached
- 949 • **Left-arrow key:** Moves the hero left and resets the position to the
 950 middle of the window when the left boundary is reached

The goals of the project are as follows:

- To gain a deeper understanding for texture coordinate 952
- To experience defining subregions within an image for texture 953
mapping 954
- To draw squares by mapping from sprite sheet elements 955
- To prepare for working with sprite animation and bitmap fonts 956

You can find the following external resource files in the assets folder: consolas-72.png 957 and minion_sprite.png. Notice that minion_sprite.png is the image shown in Figure 5-7. 958

As depicted in Figure 5-5, one of the main advantages and shortcomings of the 959 texture support defined in the previous section is that the texture coordinate accessed 960 via the `getTexCoord()` function is statically defined in the `vertex_buffer.js` file. This 961 is an advantage because in those cases where an entire image is mapped onto a square, 962 all instances of `TextureShader` objects can share the same default uv values. This is 963 also a shortcoming because the static texture coordinate buffer does not allow working 964 with different subregions of an image and thus does not support working with sprite 965 sheet elements. As illustrated in Figure 5-10, the example from this section overcomes 966 this shortcoming by defining a per-object texture coordinate in the `SpriteShader` and 967 `SpriteRenderable` objects. Notice that there are no new GLSL shaders defined since 968 their functionality remains the same as `TextureVS/FS`. 969

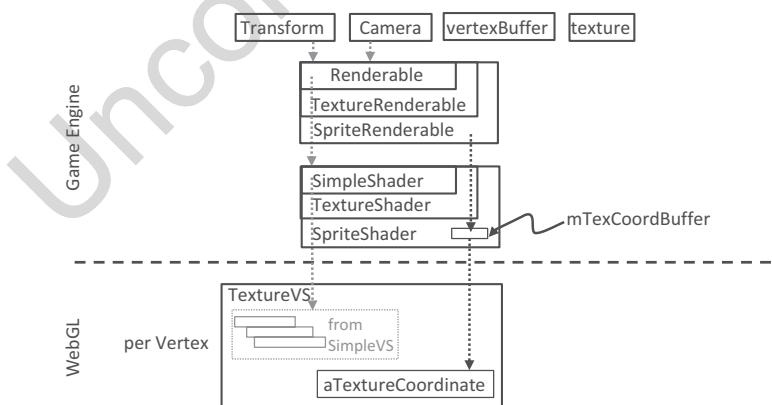


Figure 5-10. Defining a texture coordinate buffer in the `SpriteShader`

this figure will be printed in b/w

Interface GLSL Texture Shaders to the Engine with SpriteShader

971 Shaders supporting texture mapping with sprite sheet elements must be able to identify
 972 distinct subregions of an image. To support this functionality, you will implement the
 973 `SpriteShader` to define its own texture coordinate. Since this new shader extends the
 974 functionality of `TextureShader`, it is logical to implement it as a subclass.

- 975 1. Create a new file in the `src/engine/shaders` folder and name it
 `sprite_shader.js`.
- 977 2. Define the `SpriteShader` class and its constructor to extend the
 `TextureShader` class:

```

979 class SpriteShader extends TextureShader {
980     constructor(vertexShaderPath, fragmentShaderPath) {
981         // Call super class constructor
982         super(vertexShaderPath, fragmentShaderPath);
983
983         this.mTexCoordBuffer = null; // gl buffer with texture coordinate
984
984         let initTexCoord = [
985             1.0, 1.0,
986             0.0, 1.0,
987             1.0, 0.0,
988             0.0, 0.0
989         ];
990
990         let gl = glSys.get();
991         this.mTexCoordBuffer = gl.createBuffer();
992
992         gl.bindBuffer(gl.ARRAY_BUFFER, this.mTexCoordBuffer);
993         gl.bufferData(gl.ARRAY_BUFFER,
994                         new Float32Array(initTexCoord), gl.DYNAMIC_DRAW);
995                         // DYNAMIC_DRAW: says buffer content may change!
996     }
997 ... implementation to follow ...

```

SpriteShader defines its own texture coordinate buffer in WebGL,
and the reference to this buffer is kept by `mTexCoordBuffer`. Notice
that when creating this buffer in the WebGL `bufferData()` function,
the `DYNAMIC_DRAW` option is specified. This is compared with the
`STATIC_DRAW` option used in `vertex_buffer.js` when defining
the system default texture coordinate buffer. The dynamic option
informs the WebGL graphics system that the content to this buffer
will be subjected to changes.

3. Define a function to set the WebGL texture coordinate buffer:

```
setTextureCoordinate(texCoord) {  
    let gl = glSys.get();  
    gl.bindBuffer(gl.ARRAY_BUFFER, this.mTexCoordBuffer);  
    gl.bufferSubData(gl.ARRAY_BUFFER, 0, new Float32Array(texCoord));  
}
```

Note that `texCoord` parameter is an array of eight floating-point
numbers that specifies texture coordinate locations to the WebGL
context. The format and content of this array are defined by the
WebGL interface to be top-right, top-left, bottom-right, and
bottom-left corners. In your case, these should be the four corners
of a sprite sheet element.

4. Override the texture coordinate accessing function, `_getTexCoordBuffer()`, such that when the shader is activated,
the locally allocated dynamic buffer is returned and not the global
static buffer. Note that the `activate()` function is inherited from
`TextureShader`.

```
_getTexCoordBuffer() {  
    return this.mTexCoordBuffer;  
}
```

5. Remember to export the class:

```
export default SpriteShader;
```

1028 SpriteRenderable Class

1029 Similar to the Renderable class (which are shaded with SimpleShader) and
 1030 TextureRenderable class (which are shaded with TextureShader), a corresponding
 1031 SpriteRenderable class should be defined to represent objects that will be shaded with
 1032 SpriteShader:

- 1033 1. Create a new file in the `src/engine/renderables` folder and name
 it `sprite_renderable.js`.
- 1035 2. Define the `SpriteRenderable` class and constructor to extend
 from the `TextureRenderable` class. Notice that the four instance
 variables, `mElmLeft`, `mElmRight`, `mElmTop`, and `mElmBottom`,
 together identify a subregion within the Texture Space. These are
 the bounds of a sprite sheet element.

```
1040 class SpriteRenderable extends TextureRenderable {
1041     constructor(myTexture) {
1042         super(myTexture);
1043         super._setShader(shaderResources.getSpriteShader());
1044         // sprite coordinate
1045         this.mElmLeft = 0.0;    // texture coordinate bound
1046         this.mElmRight = 1.0;   // 0-left, 1-right
1047         this.mElmTop = 1.0;    // 1-top  0-bottom
1048         this.mElmBottom = 0.0; // of image
1049     }
1050     ... implementation to follow ...

```

- 1051 3. Define an enumerated data type with values that identify
 corresponding offset positions of a WebGL texture coordinate
 specification array:

```
1054 // texture coordinate array is an array of 8 floats where elements:
1055 // [0] [1]: is u/v coordinate of Top-Right
1056 // [2] [3]: is u/v coordinate of Top-Left
1057 // [4] [5]: is u/v coordinate of Bottom-Right
1058 // [6] [7]: is u/v coordinate of Bottom-Left
1059 const eTexCoordArrayIndex = Object.freeze({
```

```
eLeft: 2, 1060
eRight: 0, 1061
eTop: 1, 1062
eBottom: 5 1063
}); 1064
```

Note An enumerated data type has a name that begins with an “e,” as in eTexCoordArrayIndex.

4. Define functions to allow the specification of uv values for a sprite sheet element in both texture coordinate space (normalized between 0 and 1) and with pixel positions (which are converted to uv values):

```
// specify element region by texture coordinate (between 0 to 1) 1070
setElementUVCoordinate(left, right, bottom, top) { 1071
    this.mElmLeft = left; 1072
    this.mElmRight = right; 1073
    this.mElmBottom = bottom; 1074
    this.mElmTop = top; 1075
}

// element region defined pixel positions (0 to image resolutions) 1077
setElementPixelPositions(left, right, bottom, top) { 1078
    let texInfo = texture.get(this.mTexture); 1079
    // entire image width, height 1080
    let imageW = texInfo.mWidth; 1081
    let imageH = texInfo.mHeight; 1082

    this.mElmLeft = left / imageW; 1083
    this.mElmRight = right / imageW; 1084
    this.mElmBottom = bottom / imageH; 1085
    this.mElmTop = top / imageH; 1086
} 1087
```

Note that the setElementPixelPositions() function converts from pixel to texture coordinate before storing the results with the corresponding instance variables.

- 1090 5. Add a function to construct the texture coordinate specification
 1091 array that is appropriate for passing to the WebGL context:

```
1092 getElementUVCoordinateArray() {  

1093     return [  

1094         this.mElmRight, this.mElmTop,           // x,y of top-right  

1095         this.mElmLeft, this.mElmTop,  

1096         this.mElmRight, this.mElmBottom,  

1097         this.mElmLeft, this.mElmBottom  

1098     ];  

1099 }
```

- 1100 6. Override the draw() function to load the specific texture
 1101 coordinate values to WebGL context before the actual drawing:

```
1102 draw(camera) {  

1103     // set the current texture coordinate  

1104     // activate the texture  

1105     this.mShader.setTextureCoordinate(this.getElementUVCoordinateArray());  

1106     super.draw(camera);  

1107 }
```

- 1108 7. Finally, remember to export the class and the defined enumerated
 1109 type:

```
1110 export default SpriteRenderable;  

1111 export {eTexCoordArrayIndex}
```

1112 Facilitate Sharing with shader_resources

1113 Similar to SimpleShader and TextureShader, the SpriteShader is a resource that can be
 1114 shared. Thus, it should be added to the engine's shaderResources.

- 1115 1. In the engine/core/shader_resources.js file, import SpriteShader,
 1116 add a variable for storing, and define the corresponding getter
 1117 function to access the shared SpriteShader instance:

```
1118 import SpriteShader from "../shaders/sprite_shader.js";  

1119 let mSpriteShader = null;  

1120 function getSpriteShader() { return mSpriteShader; }
```

2. Modify the `createShaders()` function to create the `SpriteShader`:

```
function createShaders() {  
    mConstColorShader = new SimpleShader(kSimpleVS, kSimpleFS);  
    mTextureShader = new TextureShader(kTextureVS, kTextureFS);  
    mSpriteShader = new SpriteShader(kTextureVS, kTextureFS);  
}  
1122  
1123  
1124  
1125  
1126
```

Notice that the `SpriteShader` actually wraps over the existing GLSL shaders defined in the `texture_vs.gls1` and `texture_fs.gls1` files. From the perspective of WebGL, the functionality of drawing with texture remains the same. The only difference with `SpriteShader` is that the texture's coordinate values are now programmable.

3. Update the `cleanUp()` function for proper release of resources:

```
function cleanUp() {  
    mConstColorShader.cleanUp();  
    mTextureShader.cleanUp();  
    mSpriteShader.cleanUp();  
    ... identical to previous code ...  
}  
1133  
1134  
1135  
1136  
1137  
1138
```

4. Make sure to export the new functionality:

```
export {init, cleanUp,  
       getConstColorShader, getTextureShader, getSpriteShader}  
1140  
1141
```

Export New Functionality to the Client

The last step in integrating sprite element functionality into the engine involves modifying the engine access file, `index.js`. Edit `index.js` and add in the following import and export statements to grant client access to `SpriteRenderable` and `eTexCoordArrayIndex`, the enumerated data type for accessing the WebGL texture coordinate array.

```
// renderables  
import Renderable from "./renderables/renderable.js";  
import TextureRenderable from "./renderables/texture_renderable.js";  
1148  
1149  
1150
```

```

1151 import SpriteRenderable from "./renderables/sprite_renderable.js";
1152 import { eTexCoordArrayIndex } from "./renderables/sprite_renderable.js";
1153 ... identical to previous code ...
1154
1155 export default {
1156     ... identical to previous code ...
1157     // Renderables
1158     Renderable, TextureRenderable, SpriteRenderable,
1159     // constants
1160     eTexCoordArrayIndex,
1161     // functions
1162     init, cleanUp, clearCanvas
1163 }

```

1163 Testing the SpriteRenderable

1164 There are two important functionalities of sprite elements and texture coordinate
 1165 that should be tested: the proper extraction, drawing, and controlling of a sprite sheet
 1166 element as an object; and the changing and controlling of uv coordinate on an object.
 1167 For proper testing of the added functionality, you must modify the `my_game.js` file.

- 1168 1. The constructing, loading, unloading, and drawing of `MyGame` are
 1169 similar to previous examples, so the details will not be repeated here.
 1170 Please refer to the source code in the `src/my_game` folder for details.
- 1171 2. Modify the `init()` function as follows.

```

1172 init() {
1173     // Step A: set up the cameras
1174     this.mCamera = new engine.Camera(
1175         vec2.fromValues(20, 60),    // position of the camera
1176         20,                      // width of camera
1177         [20, 40, 600, 300]        // viewport (orgX, orgY, width, height)
1178     );
1179     this.mCamera.setBackgroundColor([0.8, 0.8, 0.8, 1]);
1180     // sets the background to gray

```

```

// Step B: Create the support objects          1181
this.mPortal = new engine.SpriteRenderable(this.kMinionSprite);      1182
this.mPortal.setColor([1, 0, 0, 0.2]); // tints red           1183
this.mPortal.getXform().setPosition(25, 60);                      1184
this.mPortal.getXform().setSize(3, 3);                            1185
this.mPortal.setElementPixelPositions(130, 310, 0, 180);          1186

this.mCollector = new engine.SpriteRenderable(this.kMinionSprite);   1187
this.mCollector.setColor([0, 0, 0, 0]); // No tinting            1188
this.mCollector.getXform().setPosition(15, 60);                     1189
this.mCollector.getXform().setSize(3, 3);                           1190
this.mCollector.setElementUVCoordinate(0.308, 0.483, 0, 0.352);    1191

// Step C: Create the font and minion images using sprite        1192
this.mFontImage = new engine.SpriteRenderable(this.kFontImage);     1193
this.mFontImage.setColor([1, 1, 1, 0]);                         1194
this.mFontImage.getXform().setPosition(13, 62);                  1195
this.mFontImage.getXform().setSize(4, 4);                        1196

this.mMinion = new engine.SpriteRenderable(this.kMinionSprite);    1197
this.mMinion.setColor([1, 1, 1, 0]);                         1198
this.mMinion.getXform().setPosition(26, 56);                  1199
this.mMinion.getXform().setSize(5, 2.5);                      1200

// Step D: Create hero object with texture from lower-left corner 1201
this.mHero = new engine.SpriteRenderable(this.kMinionSprite);      1202
this.mHero.setColor([1, 1, 1, 0]);                         1203
this.mHero.getXform().setPosition(20, 60);                  1204
this.mHero.getXform().setSize(2, 3);                        1205
this.mHero.setElementPixelPositions(0, 120, 0, 180);        1206
}                                                               1207

a. After the camera is set up in step A, notice that in          1208
   step B both mPortal and mCollector are created             1209
   based on the same image, kMinionSprite, with              1210
   the respective setElementPixelPositions() and            1211
   setElementUVCoordinate() calls to specify the actual sprite 1212
   element to use for rendering.                           1213

```

CHAPTER 5 WORKING WITH TEXTURES, SPRITES, AND FONTS

- 1214 b. Step C creates two additional `SpriteRenderable` objects:
1215 `mFontImage` and `mMinion`. The sprite element uv coordinate
1216 settings are the defaults where the texture image will cover
1217 the entire geometry.
- 1218 c. Similar to step B, step D creates the hero character as a
1219 `SpriteRenderable` object based on the same `kMinionSprite`
1220 image. The sprite sheet element that corresponds to the hero
1221 is identified with the `setElementPixelPositions()` call.

1222 Notice that in this example, four of the five `SpriteRenderable`
1223 objects created are based on the same `kMinionSprite` image.

- 1224 3. The `update()` function is modified to support the controlling of
1225 the hero object and changes to the uv values.

```
1226 update() {  
1227     // let's only allow the movement of hero,  
1228     let deltaX = 0.05;  
1229     let xform = this.mHero.getXform();  
  
1230     // Support hero movements  
1231     if (engine.input.isKeyPressed(engine.input.keys.Right)) {  
1232         xform.incXPosBy(deltaX);  
1233         if (xform.getXPos() > 30) { // right-bound of the window  
1234             xform.setPosition(12, 60);  
1235         }  
1236     }  
  
1237     if (engine.input.isKeyPressed(engine.input.keys.Left)) {  
1238         xform.incXPosBy(-deltaX);  
1239         if (xform.getXPos() < 11) { // left-bound of the window  
1240             xform.setPosition(20);  
1241         }  
1242     }  
  
1243     // continuously change texture tinting  
1244     let c = this.mPortal.getColor();  
1245     let ca = c[3] + deltaX;
```

```

if (ca > 1) {                                1246
    ca = 0;                                    1247
}
c[3] = ca;                                    1248
// New update code for changing the sub-texture regions being shown" 1249
let deltaT = 0.001;                           1250
                                                1251

// The font image:
// zoom into the texture by updating texture coordinate           1252
// For font: zoom to the upper left corner by changing bottom right 1253
let texCoord = this.mFontImage.getElementUVCoordinateArray(); 1254
// The 8 elements:                                              1255
//     mTexRight,   mTexTop,          // x,y of top-right        1256
//     mTexLeft,    mTexTop,          1257
//     mTexRight,   mTexBottom,       1258
//     mTexLeft,    mTexBottom       1259
let b = texCoord[engine.eTexCoordArrayIndex.eBottom] + deltaT; 1260
let r = texCoord[engine.eTexCoordArrayIndex.eRight] - deltaT; 1261
if (b > 1.0) {                                         1262
    b = 0;                                     1263
}
if (r < 0) {                                         1264
    r = 1.0;                                     1265
}
this.mFontImage.setElementUVCoordinate(                      1266
    texCoord[engine.eTexCoordArrayIndex.eLeft],            1267
    r,                                            1268
    b,                                            1269
    texCoord[engine.eTexCoordArrayIndex.eTop]             1270
);                                         1271
//                                         1272

// The minion image:
// For minion: zoom to the bottom right corner by changing top left 1273
texCoord = this.mMinion.getElementUVCoordinateArray(); 1274
// The 8 elements:                                              1275

```

CHAPTER 5 WORKING WITH TEXTURES, SPRITES, AND FONTS

```
1280      //      mTexRight,  mTexTop,          // x,y of top-right
1281      //      mTexLeft,   mTexTop,
1282      //      mTexRight,  mTexBottom,
1283      //      mTexLeft,   mTexBottom
1284  let t = texCoord[engine.eTexCoordArrayIndex.eTop] - deltaT;
1285  let l = texCoord[engine.eTexCoordArrayIndex.eLeft] + deltaT;
1286  if (l > 0.5) {
1287      l = 0;
1288  }
1289  if (t < 0.5) {
1290      t = 1.0;
1291  }
1292  this.mMinion.setElementUVCoordinate(
1293      l,
1294      texCoord[engine.eTexCoordArrayIndex.eRight],
1295      texCoord[engine.eTexCoordArrayIndex.eBottom],
1296      t
1297 );
1298 }
```

- 1299 a. Observe that the keyboard control and the drawing of the
1300 hero object are identical to previous projects.
- 1301 b. Notice the calls to setElementUVCoordinate() for
1302 mFontImage and mMinion. These calls continuously decrease
1303 and reset the V values that correspond to the bottom, the
1304 U values that correspond to the right for mFontImage, the
1305 V values that correspond to the top, and the U values that
1306 correspond to the left for mMinion. The end results are the
1307 continuous changing of texture and the appearance of a
1308 zooming animation on these two objects

Sprite Animations

In games, you often want to create animations that reflect the movements or actions of your characters. In the previous chapter, you learned about moving the geometries of these objects with transformation operators. However, as you have observed when controlling the hero character in the previous example, if the textures on these objects do not change in ways that correspond to the control, the interaction conveys the sensation of moving a static image rather than setting a character in motion. What is needed is the ability to create the illusion of animations on geometries when desired.

In the previous example, you observed from the `mFontImage` and `mMinion` objects that the appearance of an animation can be created by constantly changing the `uv` values on a texture-mapped geometry. As discussed at the beginning of this chapter, one way to control this type of animation is by working with an animated sprite sheet.

Overview of Animated Sprite Sheets

Recall that an animated sprite sheet is a sprite sheet that contains the sequence of images of an object in an animation, typically in one or more rows or columns. For example, in Figure 5-11 you can see a 2x5 animated sprite sheet that contains two separate animations organized in two rows. The animations depict an object retracting its spikes toward the right in the top row and extending them toward the left in the bottom row. In this example, the animations are separated into rows. It is also possible for an animated sprite sheet to define animations that are along columns. The organization of a sprite sheet and the details of element pixel locations are generally handled by its creator and must be explicitly communicated to the game developer for use in games.



Figure 5-11. An animated sprite sheet organized into two rows representing two animated sequences of the same object

this figure will be printed in b/w

1332 Figure 5-12 shows that to achieve the animated effect of an object retracting its
1333 spikes toward the right, as depicted by the top row of Figure 5-11, you map the elements
1334 from the left to the right in the sequence 1, 2, 3, 4, 5. When these images are mapped
1335 onto the same geometry, sequenced, and looped in an appropriate rate, it conveys the
1336 sense that the object is indeed repeating the action of retracting its spikes. Alternatively,
1337 if the sequence is reversed where the elements are mapped in the right-to-left sequence,
1338 it would create the animation that corresponds to the object extending the spikes toward
1339 the left. It is also possible to map the sequence in a swing loop from left to right and then
1340 back from right to left. In this case, the animation would correspond to the object going
1341 through the motion of retracting and extending its spikes continuously.

this figure will be printed in b/w



Figure 5-12. A sprite animation sequence that loops

1342 **The Sprite Animation Project**

1343 This project demonstrates how to work with an animated sprite sheet and
1344 generate continuous sprite animations. You can see an example of this project
1345 running in Figure 5-13. The project scene contains the objects from the previous
1346 scene plus two animated objects. The source code to this project is defined in the
1347 chapter5/5.3.sprite_animate_shaders folder.

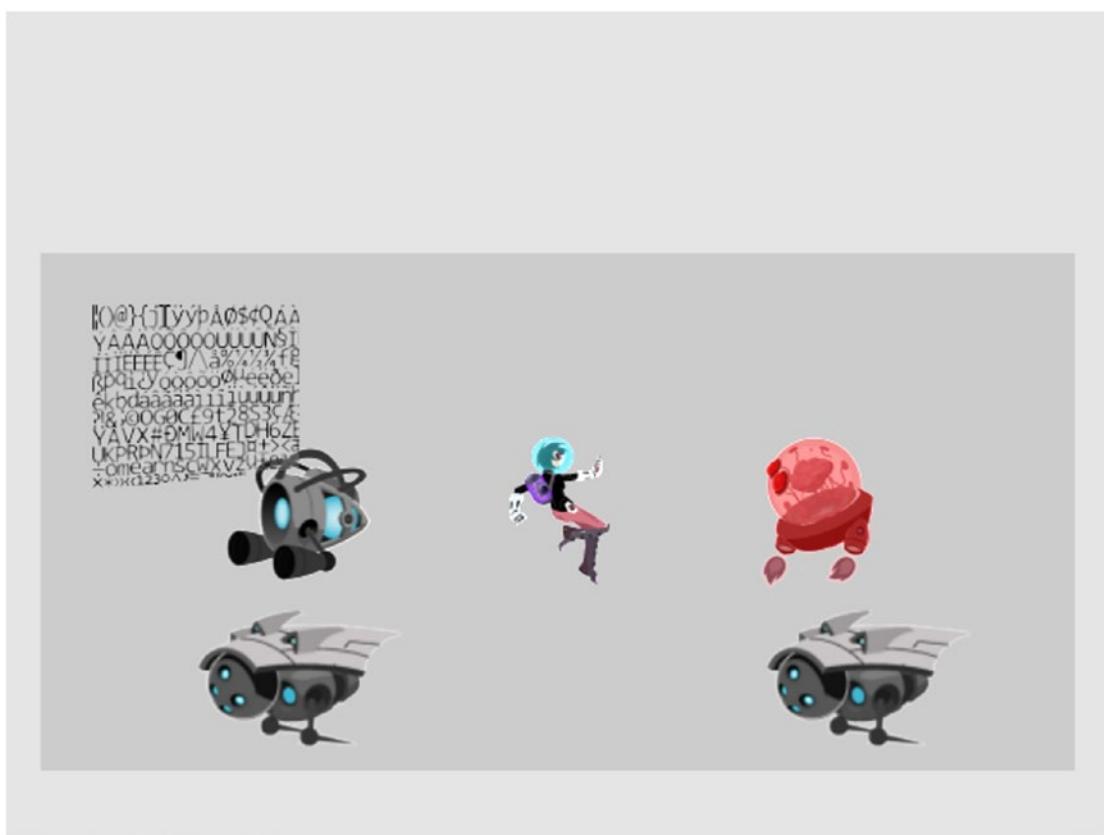


Figure 5-13. Running the *Sprite Animate Shaders* project

The controls of the project are as follows:

- **Right-arrow key:** Moves the hero right; when crossing the right boundary, the hero is wrapped back to the left boundary 1348
- **Left-arrow key:** Opposite movements of the right arrow key 1349
- **Number 1 key:** Animates by showing sprite elements continuously from right to left 1350
- **Number 2 key:** Animates by showing sprite elements moving back and forth continuously from left to right and right to left 1351
- **Number 3 key:** Animates by showing sprite elements continuously from left to right 1352
- **Number 4 key:** Animates by showing sprite elements continuously from right to left 1353
- **Number 5 key:** Animates by showing sprite elements moving back and forth continuously from left to right and right to left 1354
- **Number 6 key:** Animates by showing sprite elements continuously from left to right 1355
- **Number 7 key:** Animates by showing sprite elements continuously from right to left 1356
- **Number 8 key:** Animates by showing sprite elements moving back and forth continuously from left to right and right to left 1357

- 1358 • **Number 4 key:** Increases the animation speed
- 1359 • **Number 5 key:** Decreases the animation speed

1360 The goals of the project are as follows:

- 1361 • To understand animated sprite sheets
- 1362 • To experience the creation of sprite animations
- 1363 • To define abstractions for implementing sprite animations

1364 You can find the same files as in the previous project in the assets folder.

1365 **SpriteAnimateRenderable Class**

1366 Sprite animation can be implemented by strategically controlling the uv values of a
 1367 `SpriteRenderable` to display the appropriate sprite element at desired time periods.
 1368 For this reason, only a single class, `SpriteAnimateRenderable`, needs to be defined to
 1369 support sprite animations.

1370 For simplicity and ease of understanding, the following implementation assumes
 1371 that all sprite elements associated with an animation are always organized along the
 1372 same row. For example, in Figure 5-11, the rightward retraction and leftward extension
 1373 movements of the spikes are each organized along a row; neither spans more than
 1374 one single row, and neither is organized along a column. Animated sprite elements
 1375 organized along a column are not supported.

- 1376 1. Create a new file in the `src/engine/renderables` folder and name
 1377 it `sprite_animate_renderable.js`.
- 1378 2. Define an enumerated data type for the three different sequences
 1379 to animate:

```
1380 // Assumption: first sprite is always the leftmost element.
1381 const eAnimationType = Object.freeze({
 1382   eRight: 0, // from left to right, when ended, start from left again
 1383   eLeft: 1, // from right animate left-wards,
 1384   eSwing: 2 // left to right, then, right to left
1385 });


```

The eAnimationType enum defines three modes for animation:

- a. eRight starts at the leftmost element and animates by
iterating toward the right along the same row. When the last
element is reached, the animation continues by starting from
the leftmost element again.
 - b. eLeft is the reverse of eRight; it starts from the right,
animates toward the left, and continues by starting from the
rightmost element after reaching the leftmost element.
 - c. eSwing is a continuous loop from left to right and then from
right to left.
3. Define the SpriteAnimateRenderable class to extend from
SpriteRenderable and define the constructor:

```
class SpriteAnimateRenderable extends SpriteRenderable {
    constructor(myTexture) {
        super(myTexture);
        super._setShader(shaderResources.getSpriteShader());

        // All coordinates are in texture coordinate (UV between 0 to 1)
        // Information on the sprite element
        this.mFirstElmLeft = 0.0; // 0.0 is left corner of image
        this.mElmTop = 1.0; // image top corner (from SpriteRenderable)
        this.mElmWidth = 1.0;
        this.mElmHeight = 1.0;
        this.mWidthPadding = 0.0;
        this.mNumElems = 1; // number of elements in an animation

        // per animation settings
        this.mUpdateInterval = 1; // how often to advance
        this.mAnimationType = eAnimationType.eRight;

        this.mCurrentAnimAdvance = -1;
        this.mCurrentElm = 0;
        this._initAnimation();
    }
    ... implementation to follow ...
}
```

1418 The `SpriteAnimateRenderable` constructor defines three sets of
1419 variables:

- 1420 a. The first set, including `mFirstElmLeft`, `mElmTop`, and so
1421 on, defines the location and dimensions of each sprite
1422 element and the number of elements in the animation. This
1423 information can be used to accurately compute the texture
1424 coordinate for each sprite element when the elements are
1425 ordered by rows and columns. Note that all coordinates are
1426 in texture coordinate space (0 to 1).
- 1427 b. The second set stores information on how to animate, the
1428 `mAnimationType` of left, right, or swing, and how much time,
1429 `mUpdateInterval`, to wait before advancing to the next
1430 sprite element. This information can be changed during
1431 runtime to reverse, loop, or control the speed of a character's
1432 movement.
- 1433 c. The third set, `mCurrentAnimAdvance` and `mCurrentElm`,
1434 describes offset for advancing and the current frame
1435 number. Both of these variables are in units of element
1436 counts and are not designed to be accessed by the game
1437 programmer because they are used internally to compute
1438 the next sprite element for display.

1439 The `_initAnimation()` function computes the values of
1440 `mCurrentAnimAdvance` and `mCurrentElm` to initialize an animation
1441 sequence.

- 1442 4. Define the `_initAnimation()` function to compute the proper
1443 values for `mCurrentAnimAdvance` and `mCurrentElm` according to the
1444 current animation type:

```
1445 _initAnimation() {  
1446     // Currently running animation  
1447     this.mCurrentTick = 0;  
1448     switch (this.mAnimationType) {  
1449         case eAnimationType.eRight:  
1450             this.mCurrentElm = 0;
```

```

    this.mCurrentAnimAdvance = 1; // either 1 or -1           1451
    break;                                              1452
  case eAnimationType.eSwing:                         1453
    this.mCurrentAnimAdvance = -1 * this.mCurrentAnimAdvance; 1454
    this.mCurrentElm += 2 * this.mCurrentAnimAdvance;        1455
    break;                                              1456
  case eAnimationType.eLeft:                           1457
    this.mCurrentElm = this.mNumElems - 1;                1458
    this.mCurrentAnimAdvance = -1; // either 1 or -1       1459
    break;                                              1460
}
this._setSpriteElement();                            1462
}                                              1463

```

The `mCurrentElm` is the number of elements to offset from the leftmost, and `mCurrentAnimAdvance` records whether the `mCurrentElm` offset should be incremented (for rightward animation) or decremented (for leftward animation) during each update. The `_setSpriteElement()` function is called to set the uv values that correspond to the currently identified sprite element for displaying.

5. Define the `_setSpriteElement()` function to compute and load the uv values of the currently identified sprite element for rendering:

```

_setSpriteElement() {
  let left = this.mFirstElmLeft +
    (this.mCurrentElm * (this.mElmWidth+this.mWidthPadding));
  super.setElementUVCoordinate(left, left + this.mElmWidth,
    this.mElmTop - this.mElmHeight, this.mElmTop);
}

```

The variable `left` is the left u value of `mCurrentElm` and is used to compute the right value, with the assumption that all animation sequences are along the same row of sprite elements and that the top and bottom v values are constant where they do not change over a given animation sequence. These uv values are set to the super class `SpriteRenderable` for drawing.

CHAPTER 5 WORKING WITH TEXTURES, SPRITES, AND FONTS

1485 6. Define a function to set the animation type. Note that the
1486 animation is always reset to start from the beginning when the
1487 animation type (left, right, or swing) is changed.

```
1488 setAnimationType(animationType) {  
1489     this.mAnimationType = animationType;  
1490     this.mCurrentAnimAdvance = -1;  
1491     this.mCurrentElm = 0;  
1492     this._initAnimation();  
1493 }
```

1494 7. Define a function for specifying a sprite animation sequence. The
1495 inputs to the function are in pixels and are converted to texture
1496 coordinates by dividing by the width and height of the image.

```
1497 // Always set the leftmost element to be the first  
1498 setSpriteSequence(  
1499     topPixel, // offset from top-left  
1500     leftPixel, // offset from top-left  
1501     elmWidthInPixel,  
1502     elmHeightInPixel,  
1503     numElements, // number of elements in sequence  
1504     wPaddingInPixel // left/right padding  
1505 ) {  
1506     let texInfo = texture.get(this.mTexture);  
1507     // entire image width, height  
1508     let imageW = texInfo.mWidth;  
1509     let imageH = texInfo.mHeight;  
1510  
1511     this.mNumElems = numElements; // number of elements in animation  
1512     this.mFirstElmLeft = leftPixel / imageW;  
1513     this.mElmTop = topPixel / imageH;  
1514     this.mElmWidth = elmWidthInPixel / imageW;  
1515     this.mElmHeight = elmHeightInPixel / imageH;  
1516     this.mWidthPadding = wPaddingInPixel / imageW;  
1517     this._initAnimation();  
1518 }
```

8. Define functions to change animation speed, either directly or by
an offset: 1518
1519

```
setAnimationSpeed(tickInterval) { 1520
    this.mUpdateInterval = tickInterval; } 1521
incAnimationSpeed(deltaInterval) { 1522
    this.mUpdateInterval += deltaInterval; } 1523
```

9. Define a function to advance the animation for each game loop
update: 1524
1525

```
updateAnimation() { 1526
    this.mCurrentTick++;
    if (this.mCurrentTick >= this.mUpdateInterval) { 1527
        this.mCurrentTick = 0;
        this.mCurrentElm += this.mCurrentAnimAdvance;
        if ((this.mCurrentElm>=0) && (this.mCurrentElm<this.mNumElems)) { 1528
            this._setSpriteElement(); 1529
        } else { 1530
            this._initAnimation(); 1531
        }
    } 1532
}
```

Each time the updateAnimation() function is called, the
mCurrentTick counter is incremented, and when the number of
ticks reaches the mUpdateInterval value, the animation is re-
initialized by the _initAnimation() function. It is important to
note that the time unit for controlling the animation is the number
of times the updateAnimation() function is called and not the
real-world elapsed time. Recall that the engine loop.loopOnce()
function ensures system-wide updates to occur at kMPF intervals
even when frame rate lags. The game engine architecture ensures
the updateAnimation() function calls are kMPF milliseconds apart. 1538
1539
1540
1541
1542
1543
1544
1545
1546
1547

- 1548 10. Finally, remember to export the defined class and enumerated
1549 animation type:

```
1550 export default SpriteAnimateRenderable;  
1551 export {eAnimationType}
```

1552 Export New Functionality to the Client

1553 The last step in integrating animated sprite element functionality into the
1554 engine involves modifying the engine access file, `index.js`. Edit `index.js` and
1555 add in the following import and export statements to grant client access to
1556 `SpriteAnimateRenderable` and `eAnimationType`:

```
1557 // renderables  
1558 import Renderable from "./renderables/renderable.js";  
1559 import SpriteRenderable from "./renderables/sprite_renderable.js";  
1560 import SpriteAnimateRenderable from  
1561                    "./renderables/sprite_animate_renderable.js";  
1562 import { eTexCoordArrayIndex } from "./renderables/sprite_renderable.js";  
1563 import { eAnimationType } from  
1564                    "./renderables/sprite_animate_renderable.js";  
1565 ... identical to previous code ...  
1566 export default {  
1567     ... identical to previous code ...  
1568     // Renderables  
1569     Renderable, TextureRenderable,  
1570     SpriteRenderable, SpriteAnimateRenderable,  
1571     // constants  
1572     eTexCoordArrayIndex, eAnimationType,  
1573     // functions  
1574     init, cleanUp, clearCanvas  
1575 }
```

Testing Sprite Animation

1576

The test cases for the `SpriteAnimateRenderable` object must demonstrate the game programmer's control over the modes (left, right, swing) and speed of animation. The `MyGame` object is modified to accomplish these purposes.

1. The constructing, loading, unloading, and drawing of `MyGame` are similar to the previous example and the details are not repeated.

2. In the `init()` function, add code to create and initialize the `SpriteAnimateRenderable` objects between steps C and D:

```
init() {  
    ... identical to previous code ...  
  
    // The right minion  
    this.mRightMinion = new engine.SpriteAnimateRenderable(  
        this.kMinionSprite);  
    this.mRightMinion.setColor([1, 1, 1, 0]);  
    this.mRightMinion.getXform().setPosition(26, 56.5);  
    this.mRightMinion.getXform().setSize(4, 3.2);  
    this.mRightMinion.setSpriteSequence(  
        512, 0,    // first element pixel positions: top: 512 left: 0  
        204, 164, // widthxheight in pixels  
        5,        // number of elements in this sequence  
        0);       // horizontal padding in between  
    this.mRightMinion.setAnimationType(engine.eAnimationType.eRight);  
    this.mRightMinion.setAnimationSpeed(50);  
    // the left minion  
    this.mLeftMinion = new engine.SpriteAnimateRenderable(  
        this.kMinionSprite);  
    this.mLeftMinion.setColor([1, 1, 1, 0]);  
    this.mLeftMinion.getXform().setPosition(15, 56.5);  
    this.mLeftMinion.getXform().setSize(4, 3.2);  
    this.mLeftMinion.setSpriteSequence(  
        348, 0,    // first element pixel positions: top: 164 left: 0  
        204, 164, // widthxheight in pixels  
        5,        // number of elements in this sequence
```

CHAPTER 5 WORKING WITH TEXTURES, SPRITES, AND FONTS

```
1609     0);      // horizontal padding in between  
1610     this.mLeftMinion.setAnimationType(engine.eAnimationType.eRight);  
1611     this.mLeftMinion.setAnimationSpeed(50);  
  
1612     ... identical to previous code ...  
1613 }
```

1614 The `SpriteAnimateRenderable` objects are created in similar ways
1615 as `SpriteRenderable` objects with a sprite sheet as the texture
1616 parameter. In this case, it is essential to call the `setSpriteSequence()`
1617 function to identify the elements involved in the animation including
1618 the location, dimension, and total number of elements.

- 1619 3. The `update()` function must invoke the `SpriteAnimateRenderable`
1620 object's `updateAnimation()` function to advance the sprite animation:

```
1621 update() {  
1622     ... identical to previous code ...  
  
1623     // remember to update the minion's animation  
1624     this.mRightMinion.updateAnimation();  
1625     this.mLeftMinion.updateAnimation();  
  
1626     // Animate left on the sprite sheet  
1627     if (engine.input.isKeyClicked(engine.input.keys.One)) {  
1628         this.mRightMinion.setAnimationType(engine.eAnimationType.eLeft);  
1629         this.mLeftMinion.setAnimationType(engine.eAnimationType.eLeft);  
1630     }  
  
1631     // swing animation  
1632     if (engine.input.isKeyClicked(engine.input.keys.Two)) {  
1633         this.mRightMinion.setAnimationType(engine.eAnimationType.eSwing);  
1634         this.mLeftMinion.setAnimationType(engine.eAnimationType.eSwing);  
1635     }  
  
1636     // Animate right on the sprite sheet  
1637     if (engine.input.isKeyClicked(engine.input.keys.Three)) {  
1638         this.mRightMinion.setAnimationType(engine.eAnimationType.eRight);  
1639         this.mLeftMinion.setAnimationType(engine.eAnimationType.eRight);  
1640     }
```

```

// decrease duration of each sprite element to speed up animation      1641
if (engine.input.isKeyClicked(engine.input.keys.Four)) {                  1642
    this.mRightMinion.incAnimationSpeed(-2);                            1643
    this.mLeftMinion.incAnimationSpeed(-2);                           1644
}
                                                               1645

// increase duration of each sprite element to slow down animation     1646
if (engine.input.isKeyClicked(engine.input.keys.Five)) {                 1647
    this.mRightMinion.incAnimationSpeed(2);                            1648
    this.mLeftMinion.incAnimationSpeed(2);                           1649
}
                                                               1650

... identical to previous code ...                                     1651
}
                                                               1652

```

The keys 1, 2, and 3 change the animation type, and keys 4 and 5 change the animation speed. Note that the limit of the animation speed is the update rate of the game loop.

Fonts and Drawing of Text

A valuable tool that many games use for a variety of tasks is text output. Drawing of text messages is an efficient way to communicate to the user as well as you, the developer. For example, text messages can be used to communicate the game's story, the player's score, or debugging information during development. Unfortunately, WebGL does not support the drawing of text. This section briefly introduces bitmap fonts and introduces FontRenderable objects to support the drawing of texts.

Bitmap Fonts

A font must be defined such that individual characters can be extracted for the drawing of text messages. A bitmap font, as the name implies, is a simple map describing which bit (or pixel) must be switched on to represent characters in the font. Combining all characters of a bitmap font into a single image and defining an accompanied decoding description document provide a straightforward solution for drawing text output. For example, Figure 5-14 shows a bitmap font sprite where all the defined characters are tightly organized into the same image. Figure 5-15 is a snippet of the accompanying decoding description in XML format.



Figure 5-14. An example bitmap font sprite image

```
<?xml version="1.0"?>
<font>
  <info face="Consolas" size="24" bold="0" italic="0" charset="" unicode="1" stretchH="100" smooth="1" aa="1"
padding="0,0,0,0" spacing="1,1" outline="0"/>
  <common lineHeight="24" base="19" scaleW="256" scaleH="128" pages="1" packed="0" alphaChnl="0" redChnl="3"
greenChnl="3" blueChnl="3"/>
  <pages>
    <page id="0" file="Consolas-24-NoKerning_0.png" />
  </pages>
  <chars count="193">
    <char id="0" x="252" y="35" width="3" height="1" xoffset="-1" yoffset="23" xadvance="11" page="0" chnl="15" />
    <char id="13" x="254" y="0" width="0" height="1" xoffset="0" yoffset="23" xadvance="0" page="0" chnl="15" />
    <char id="32" x="17" y="38" width="3" height="1" xoffset="-1" yoffset="23" xadvance="11" page="0" chnl="15" />
```

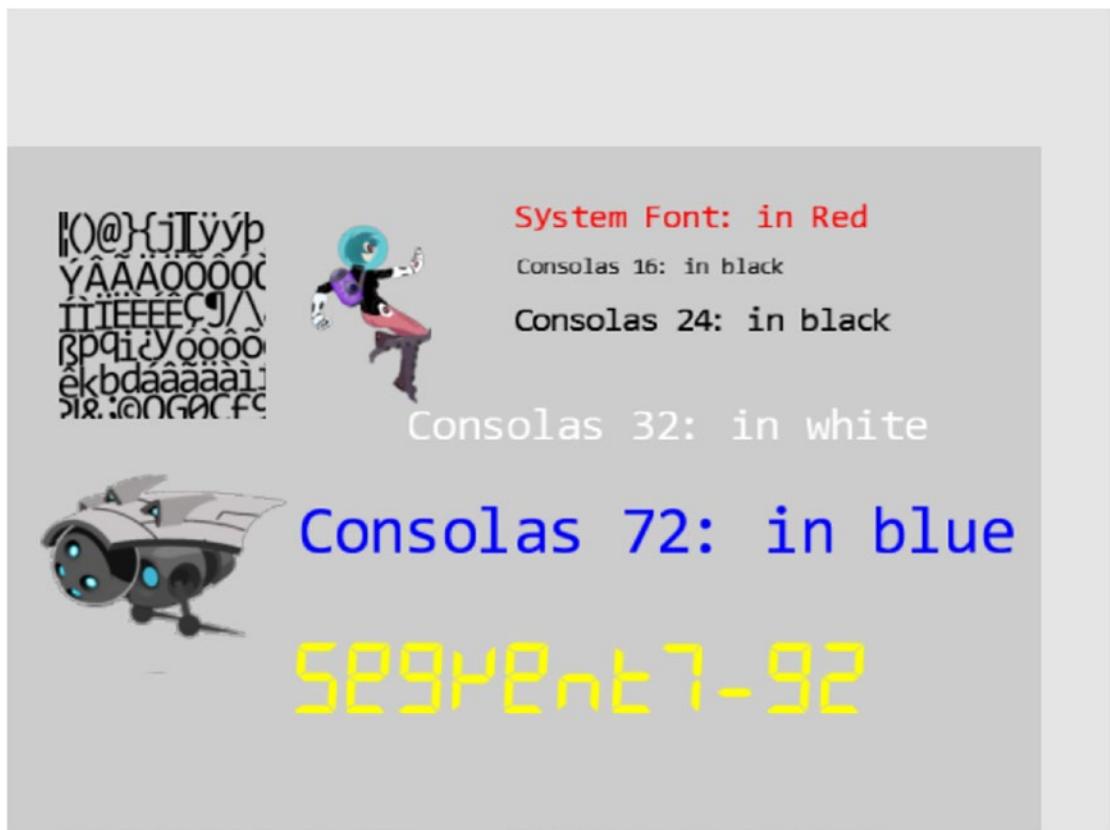
Figure 5-15. A snippet of the XML file with the decoding information for the bitmap font image shown in Figure 5-14

1672 Notice that the decoding information as shown in Figure 5-15 uniquely defines the
1673 uv coordinate positions for each character in the image, as shown in Figure 5-14. In this
1674 way, displaying individual characters from a bitmap font sprite image can be performed
1675 in a straightforward manner by the SpriteRenderable objects.

1676 **Note** There are many bitmap font file formats. The format used in this book is
1677 the AngleCode BMFont-compatible font in XML form. BMFont is an open source
1678 software that converts vector fonts, such as TrueType and OpenType, into bitmap
1679 fonts. See www.angelcode.com/products/bmfont/ for more information.

The Font Support Project

This project demonstrates how to draw text from a bitmap font using the SpriteRenderable object. You can see an example of this project running in Figure 5-16. The source code to this project is defined in the chapter5/5.4.font_support folder.



this figure will be printed in b/w

Figure 5-16. Running the Font Support project

The controls of the project are as follows:

- **Number keys 0, 1, 2, and 3:** Select the Consolas, 16, 24, 32, or 72 fonts, respectively, for size modification.
- **Up/down key while holding down X/Y key:** Increases or decreases (arrow keys) the width (X key) or the height (Y key) of the selected font.
- **Left- and right-arrow keys:** Move the hero left or right. The hero wraps if it exits the bounds.

1691 The goals of the project are as follows:

- 1692 • To understand bitmap fonts
1693 • To gain a basic understanding of drawing text strings in a game
1694 • To implement text drawing support in your game engine

1695 You can find the following external resource files in the assets folder:

1696 consolas-72.png and minion_sprite.png. In the assets/fonts folder are the
1697 bitmap font sprite image files and the associated XML files that contain the decoding
1698 information: consolas-16.fnt, consolas-16.png, consolas-24.fnt, consolas-24.
1699 png, consolas-32.fnt, consolas-32.png, consolas-72.fnt, consolas-72.png,
1700 segment7-96.fnt, segment7-96.png, system-default-font.fnt, and system-default-
1701 font.png.

1702 Notice that the .fnt and .png files are paired. The former contains decoding
1703 information for the latter. These file pairs must be included in the same folder for the
1704 engine to load the font properly. system-default-font is the default font for the game
1705 engine, and it is assumed that this font is always present in the asset/fonts folder.

1706 **Note** The actions of parsing, decoding, and extracting of character information
1707 from the .fnt files are independent from the foundational operations of a game
1708 engine. For this reason, the details of these operations are not presented. If you are
1709 interested, you should consult the source code.

1710 Loading and Storing Fonts in the Engine

1711 Loading font files is special because fonts are defined in pairs: the .fnt file that contains
1712 decoding information and the corresponding .png sprite image file. However, since the
1713 .fnt file is an XML file and the .png file is a simple texture image, the loading of these
1714 two types of files is already supported by the engine. The details of loading and storing
1715 fonts in the engine are encapsulated by a new engine component, font.

- 1716 1. Create a new file in the src/engine/resources folder and name it
1717 font.js.

2. Import the resource management functionality from the `xml` module
for loading the `.fnt` file and the texture module for the `.png` sprite
image file, and define local constants for these file extensions:

```
import * as xml from "./xml.js";  
import * as texture from "./texture.js";
```

```
let kDescExt = ".fnt"; // extension for the bitmap font description  
let kImageExt = ".png"; // extension for the bitmap font image
```

3. Define a class for storing uv coordinate locations and the size
associated with a character. This information can be computed
based on the contents from the `.fnt` file.

```
class CharacterInfo {  
    constructor() {  
        // in texture coordinate (0 to 1) maps to the entire image  
        this.mTexCoordLeft = 0;  
        this.mTexCoordRight = 1;  
        this.mTexCoordBottom = 0;  
        this.mTexCoordTop = 0;  
  
        // nominal char size, 1 is "standard width/height" of a char  
        this.mCharWidth = 1;  
        this.mCharHeight = 1;  
        this.mCharWidthOffset = 0;  
        this.mCharHeightOffset = 0;  
  
        // reference of char width/height ratio  
        this.mCharAspectRatio = 1;  
    }  
}
```

4. Define two functions to return proper extensions based on a path
with no file extension. Note that `fontName` is a path to the font
files but without any file extensions. For example, `assets/fonts/
system-default-font` is the string and the two functions identify
the two associated `.fnt` and `.png` files.

CHAPTER 5 WORKING WITH TEXTURES, SPRITES, AND FONTS

```
1749 function descName(fontName) { return fontName+kDescExt; }
1750 function imageName(fontName) { return fontName+kImageExt; }

1751      5. Define the load() and unload() functions. Notice that two file
1752          operations are actually invoked in each: one for the .fnt and the
1753          second for the .png files.

1754 function load(fontName) {
1755     xml.load(descName(fontName));
1756     texture.load(imageName(fontName));
1757 }

1758 function unload(fontName) {
1759     xml.unload(descName(fontName));
1760     texture.unload(imageName(fontName));
1761 }

1762      6. Define a function to inquire the loading status of a given font:

1763 function has(fontName) {
1764     return texture.has(imageName(fontName)) &&
1765         xml.has(descName(fontName));
1766 }

1767      7. Define a function to compute CharacterInfo based on the
1768          information presented in the .fnt file:

1769 function getCharInfo(fontName, aChar) {
1770     ... details omitted for lack of relevancy
1771
1772     returnInfo = new CharacterInfo();
1773
1774     // computes and fills in the contents of CharacterInfo
1775     ... details omitted for lack of relevancy
1776
1777     return returnInfo;
1778 };

1779      Details of decoding and extracting information for a given
1780      character are omitted because they are unrelated to the rest of the
1781      game engine implementation.
```

Note For details of the .fnt format information, please refer to www.angelcode.com/products/bmfont/doc/file_format.html. 1779
1780

8. Finally, remember to export the functions from this module: 1781

```
export {has, load, unload,  
        imageName, descName,  
        CharacterInfo,  
        getCharInfo  
} 1782  
1783  
1784  
1785  
1786
```

Adding a Default Font to the Engine 1787

A default system font should be provided by the game engine for the convenience of the game programmer. To accomplish this, an engine utility should be defined to load and initialize default resources to be shared with the game developer. Recall that the shader resources module in the src/engine/core folder is defined to support engine-wide sharing of shaders. This pattern can be duplicated for sharing of default resources with the client. A default_resources module can be defined in the src/engine/resources folder to accomplish this sharing. 1788
1789
1790
1791
1792
1793
1794

1. Create a file in the src/engine/resources folder and name it default_resources.js, import functionality from the font and resource_map modules, and define a constant string and its getter function for the path to the default system font: 1795
1796
1797
1798

```
import * as font from "./font.js";  
import * as map from "../core/resource_map.js";  
  
// Default font  
let kDefaultFont = "assets/fonts/system_default_font";  
var getDefaultFont = function() { return kDefaultFont; } 1799  
1800  
1801  
1802  
1803
```

CHAPTER 5 WORKING WITH TEXTURES, SPRITES, AND FONTS

1804 2. Define an `init()` function to issue the default system font
1805 loading request in a JavaScript Promise and append the Promise
1806 to the array of outstanding load requests in the `resource_map`.
1807 Recall that the `loop.start()` function in the `loop` module waits
1808 for the fulfillment of all `resource_map` loading promises before
1809 starting the game loop. For this reason, as in the case of all other
1810 asynchronously loaded resources, by the time the game loop
1811 begins, the default system font will have been properly loaded.

```
1812 function init() {  
1813     let loadPromise = new Promise(  
1814         async function (resolve) {  
1815             await Promise.all([  
1816                 font.load(kDefaultFont)  
1817             ]);  
1818             resolve();  
1819         }).then(  
1820             function resolve() { /* nothing to do for font */ }  
1821         );  
1822     map.pushPromise(loadPromise);  
1823 }
```

1824 3. Define the `cleanUp()` function to release all allocated resources,
1825 in this case, unload the font:

```
1826 // unload all resources  
1827 function cleanUp() {  
1828     font.unload(kDefaultFont);  
1829 }
```

1830 4. Lastly, remember to export all defined functionality:

```
1831 export {  
1832     init, cleanUp,  
1833     // default system font name: this is guaranteed to be loaded  
1834     getDefaultFontName  
1835 }
```

Defining a FontRenderable Object to Draw Texts

The defined font module is capable of loading font files and extracting per-character uv coordinate and size information. With this functionality, the drawing of a text string can be accomplished by identifying each character in the string, retrieving the corresponding texture mapping information, and rendering the character using a SpriteRenderable object. The FontRenderable object will be defined to accomplish this.

1. Create a new file in the `src/engine/renderables` folder and name it `font_renderable.js`. 1842
2. Define the `FontRenderable` class and its constructor to accept a string as its parameter: 1844

```
class FontRenderable { 1846
    constructor(aString) { 1847
        this.mFontName = defaultResources.getDefaultFontName(); 1848
        this.mOneChar = new SpriteRenderable( 1849
            font.imageName(this.mFontName)); 1850
        this.mXform = new Transform(); // to move this object around 1851
        this.mText = aString; 1852
    } 1853
    ... implementation to follow ... 1854
```

- a. The `aString` variable is the message to be drawn. 1855
 - b. Notice that `FontRenderable` objects do not customize the behaviors of `SpriteRenderable` objects. Rather, it relies on a `SpriteRenderable` object to draw each character in the string. For this reason, `FontRenderable` is not a subclass of but instead contains an instance of the `SpriteRenderable` object, the `mOneChar` variable. 1856
3. Define the `draw()` function to parse and draw each character in the string using the `mOneChar` variable: 1862

```
draw(camera) { 1864
    // we will draw the text string by calling mOneChar for each of the 1865
    // chars in the mText string. 1866
```

CHAPTER 5 WORKING WITH TEXTURES, SPRITES, AND FONTS

```
1867     let widthOfOneChar = this.mXform.getWidth() / this.mText.length;
1868     let heightOfOneChar = this.mXform.getHeight();
1869     let yPos = this.mXform.getYPos();
1870
1871     // center position of the first char
1872     let xPos = this.mXform.getXPos() -
1873         (widthOfOneChar / 2) + (widthOfOneChar * 0.5);
1874     let charIndex, aChar, charInfo, xSize, ySize, xOffset, yOffset;
1875     for (charIndex = 0; charIndex < this.mText.length; charIndex++) {
1876         aChar = this.mText.charCodeAt(charIndex);
1877         charInfo = font.getCharInfo(this.mFontName, aChar);
1878
1879         // set the texture coordinate
1880         this.mOneChar.setElementUVCoordinate(
1881             charInfo.mTexCoordLeft, charInfo.mTexCoordRight,
1882             charInfo.mTexCoordBottom, charInfo.mTexCoordTop);
1883
1884         // now the size of the char
1885         xSize = widthOfOneChar * charInfo.mCharWidth;
1886         ySize = heightOfOneChar * charInfo.mCharHeight;
1887         this.mOneChar.getXform().setSize(xSize, ySize);
1888
1889         // how much to offset from the center
1890         xOffset = widthOfOneChar * charInfo.mCharWidthOffset * 0.5;
1891         yOffset = heightOfOneChar * charInfo.mCharHeightOffset * 0.5;
1892
1893         this.mOneChar.getXform().setPosition(xPos-xOffset, yPos-yOffset);
1894
1895         this.mOneChar.draw(camera);
1896
1897         xPos += widthOfOneChar;
1898     }
1899 }
```

The dimension of each character is defined by widthOfOneChar and heightOfOneChar where the width is simply dividing the total FontRenderable width by the number of characters in the string. The for loop then performs the following operations:

- a. Extracts each character in the string 1897
 - b. Calls the getCharInfo() function to receive the character's uv values and size information in charInfo 1898
1899
 - c. Uses the uv values from charInfo to identify the sprite element location for mOneChar (by calling and passing the information to the mOneChar.setElementUVCoordinate() function) 1900
1901
1902
1903
 - d. Uses the size information from charInfo to compute the actual size (xSize and ySize) and location offset for the character (xOffset and yOffset) and draws the character mOneChar with the appropriate settings 1904
1905
1906
1907
4. Implement the getters and setters for the transform, the text message to be drawn, the font to use for drawing, and the color: 1908
1909

```

getXform() { return this.mXform; } 1910
getText() { return this.mText; } 1911

setText(t) {
    this.mText = t;
    this.setTextHeight(this.getXform().getHeight()); 1912
} 1913
1914
getFontName() { return this.mFontName; } 1916
setFontName(f) {
    this.mFontName = f;
    this.mOneChar.setTexture(font.imageName(this.mFontName)); 1917
} 1918
1919
setColor(c) { this.mOneChar.setColor(c); } 1921
getColor() { return this.mOneChar.getColor(); } 1922

```

- 1923 5. Define the `setTextHeight()` function to define the height of the
 1924 message to be output:

```
1925 setTextHeight(h) {
1926   let charInfo = font.getCharInfo(this.mFontName, "A".charCodeAt(0));
1927   let w = h * charInfo.mCharAspectRatio;
1928   this.getXform().setSize(w * this.mText.length, h);
1929 }
```

1930 Notice that the width of the entire message to be drawn is
 1931 automatically computed based on the message string length and
 1932 maintaining the character width to height aspect ratio.

- 1933 6. Finally, remember to export the defined class:

```
1934 export default FrontRenderable;
```

1935 **Note** `FontRenderable` does not support the rotation of the entire message.
 1936 Text messages are always drawn horizontally from left to right.

1937 Initialize, Cleaning, and Export Font Functionality

1938 As in all engine functionality, it is important to update the engine access file, `index.js`,
 1939 to grant access to the game developer. In this case, it is also essential to initialize and
 1940 clean up resources associated with the default system font.

- 1941 1. Edit `index.js` to import functionality from the `font` and `default_`
 1942 resources modules and the `FontRenderable` class:

```
1943 // resources
1944 import * as audio from "./resources/audio.js";
1945 import * as text from "./resources/text.js";
1946 import * as xml from "./resources/xml.js";
1947 import * as texture from "./resources/texture.js";
1948 import * as font from "./resources/font.js";
1949 import * as defaultResources from "./resources/default_resources.js";
1950 ... identical to previous code ...
```

```
// renderables 1951
import Renderable from "./renderables/renderable.js"; 1952
import SpriteRenderable from "./renderables/sprite_renderable.js"; 1953
import SpriteAnimateRenderable from 1954
    "./renderables/sprite_animate_renderable.js"; 1955
import FontRenderable from "./renderables/font_renderable.js"; 1956
... identical to previous code ... 1957
```

2. Add default resources initialization and cleanup in the engine
init() and cleanUp() functions:

```
function init(htmlCanvasID) { 1960
    glSys.init(htmlCanvasID); 1961
    vertexBuffer.init(); 1962
    input.init(); 1963
    audio.init(); 1964
    shaderResources.init(); 1965
    defaultResources.init(); 1966
}

function cleanUp() { 1968
    loop.cleanUp(); 1969
    shaderResources.cleanUp(); 1970
    defaultResources.cleanUp(); 1971
    audio.cleanUp(); 1972
    input.cleanUp(); 1973
    vertexBuffer.cleanUp(); 1974
    glSys.cleanUp(); 1975
}
```

3. Remember to export the newly defined functionality:

```
export default { 1978
    // resource support
    audio, text, xml, texture, font, defaultResources, 1979
    ... identical to previous code ... 1980
} 1981
```

```
1982    // Renderables  
1983    Renderable, TextureRenderable,  
1984    SpriteRenderable, SpriteAnimateRenderable, FontRenderable,  
1985    ... identical to previous code ...  
1986 }
```

1987 Testing Fonts

1988 You can now modify the MyGame scene to print messages with the various fonts found in
1989 the assets folder:

- 1990 4. In the `my_game.js` file, modify the constructor to define
1991 corresponding variables for printing the messages, and modify the
1992 `draw()` function to draw all objects accordingly. Please refer to the
1993 `src/my_game/my_game.js` file for the details of the code.
- 1994 5. Modify the `load()` function to load the textures and fonts. Once
1995 again, notice that the font paths, for example, `assets/fonts/`
1996 `consolas-16`, do not include file name extensions. Recall that this
1997 path will be appended with `.fnt` and `.png`, where two separate
1998 files will be loaded to support the drawing of fonts.

```
1999 load() {  
2000     // Step A: loads the textures  
2001     engine.texture.load(this.kFontImage);  
2002     engine.texture.load(this.kMinionSprite);  
2003     // Step B: loads all the fonts  
2004     engine.font.load(this.kFontCon16);  
2005     engine.font.load(this.kFontCon24);  
2006     engine.font.load(this.kFontCon32);  
2007     engine.font.load(this.kFontCon72);  
2008     engine.font.load(this.kFontSeg96);  
2009 }
```

6. Modify the `unload()` function to unload the textures and fonts:

```

unload() {
    engine.texture.unload(this.kFontImage); 2011
    engine.texture.unload(this.kMinionSprite); 2012
    engine.texture.unload(this.kMinionSprite); 2013

    // unload the fonts
    engine.font.unload(this.kFontCon16); 2014
    engine.font.unload(this.kFontCon24); 2015
    engine.font.unload(this.kFontCon32); 2016
    engine.font.unload(this.kFontCon72); 2017
    engine.font.unload(this.kFontSeg96); 2018
}

```

7. Define a private `_initText()` function to set the color, location, and height of a `FontRenderable` object. Modify the `init()` function to set up the proper WC system and initialize the fonts. Notice the calls to `setFont()` function to change the font type for each message.

```

_initText(font, posX, posY, color, textH) { 2026
    font.setColor(color); 2027
    font.getXform().setPosition(posX, posY); 2028
    font.setTextHeight(textH); 2029
}

init() { 2031
    // Step A: set up the cameras
    this.mCamera = new engine.Camera( 2032
        vec2.fromValues(50, 33),      // position of the camera 2033
        100,                      // width of camera 2034
        [0, 0, 600, 400]           // viewport (orgX, orgY, width, height) 2035
    );
    this.mCamera.setBackgroundColor([0.8, 0.8, 0.8, 1]); 2036
    // sets the background to gray 2037

    // Step B: Create the font and minion images using sprite
    this.mFontImage = new engine.SpriteRenderable(this.kFontImage); 2038
}

```

CHAPTER 5 WORKING WITH TEXTURES, SPRITES, AND FONTS

```
2042     this.mFontImage.setColor([1, 1, 1, 0]);  
2043     this.mFontImage.getXform().setPosition(15, 50);  
2044     this.mFontImage.getXform().setSize(20, 20);  
  
2045     // The right minion  
2046     this.mMinion = new engine.SpriteAnimateRenderable(  
2047             this.kMinionSprite);  
2048     this.mMinion.setColor([1, 1, 1, 0]);  
2049     this.mMinion.getXform().setPosition(15, 25);  
2050     this.mMinion.getXform().setSize(24, 19.2);  
2051     this.mMinion.setSpriteSequence(512, 0, // first element: top, left  
2052         204, 164, // widthxheight in pixels  
2053         5, // number of elements in this sequence  
2054         0); // horizontal padding in between  
2055     this.mMinion.setAnimationType(engine.eAnimationType.eSwing);  
2056     this.mMinion.setAnimationSpeed(15);  
2057     // show each element for mAnimSpeed updates  
  
2058     // Step D: Create hero object with texture from lower-left corner  
2059     this.mHero = new engine.SpriteRenderable(this.kMinionSprite);  
2060     this.mHero.setColor([1, 1, 1, 0]);  
2061     this.mHero.getXform().setPosition(35, 50);  
2062     this.mHero.getXform().setSize(12, 18);  
2063     this.mHero.setElementPixelPositions(0, 120, 0, 180);  
  
2064     // Create the fonts  
2065     this.mTextSysFont = new engine.FontRenderable("System Font: in Red");  
2066     this._initText(this.mTextSysFont, 50, 60, [1, 0, 0, 1], 3);  
  
2067     this.mTextCon16 = new engine.FontRenderable("Consolas 16: in black");  
2068     this.mTextCon16.setFontName(this.kFontCon16);  
2069     this._initText(this.mTextCon16, 50, 55, [0, 0, 0, 1], 2);  
  
2070     this.mTextCon24 = new engine.FontRenderable("Consolas 24: in black");  
2071     this.mTextCon24.setFontName(this.kFontCon24);  
2072     this._initText(this.mTextCon24, 50, 50, [0, 0, 0, 1], 3);  
  
2073     this.mTextCon32 = new engine.FontRenderable("Consolas 32: in white");
```

```

this.mTextCon32.setFontName(this.kFontCon32);          2074
this._initText(this.mTextCon32, 40, 40, [1, 1, 1, 1], 4); 2075

this.mTextCon72 = new engine.FontRenderable("Consolas 72: in blue"); 2076
this.mTextCon72.setFontName(this.kFontCon72);           2077
this._initText(this.mTextCon72, 30, 30, [0, 0, 1, 1], 6); 2078

this.mTextSeg96 = new engine.FontRenderable("Segment7-92"); 2079
this.mTextSeg96.setFontName(this.kFontSeg96);            2080
this._initText(this.mTextSeg96, 30, 15, [1, 1, 0, 1], 7); 2081

this.mTextToWork = this.mTextCon16;                     2082
}

```

8. Modify the update() function with the following:

```

update() {                                         2085
    ... identical to previous code ...          2086

    // choose which text to work on           2087
    if (engine.input.isKeyClicked(engine.input.keys.Zero)) {
        this.mTextToWork = this.mTextCon16;       2088
    }
    if (engine.input.isKeyClicked(engine.input.keys.One)) {
        this.mTextToWork = this.mTextCon24;       2089
    }
    if (engine.input.isKeyClicked(engine.input.keys.Three)) {
        this.mTextToWork = this.mTextCon32;       2090
    }
    if (engine.input.isKeyClicked(engine.input.keys.Four)) {
        this.mTextToWork = this.mTextCon72;       2091
    }

    let deltaF = 0.005;                         2092
    if (engine.input.isKeyPressed(engine.input.keys.Up)) {
        if (engine.input.isKeyPressed(engine.input.keys.X)) {
            this.mTextToWork.getXform().incWidthBy(deltaF); 2093
        }
    }
}

```

CHAPTER 5 WORKING WITH TEXTURES, SPRITES, AND FONTS

```
2105     if (engine.input.isKeyPressed(engine.input.keys.Y)) {
2106         this.mTextToWork.getXform().incHeightBy(deltaF);
2107     }
2108     this.mTextSysFont.setText(
2109         this.mTextToWork.getXform().getWidth().toFixed(2) + "x" +
2110         this.mTextToWork.getXform().getHeight().toFixed(2));
2111 }
2112
2113 if (engine.input.isKeyPressed(engine.input.keys.Down)) {
2114     if (engine.input.isKeyPressed(engine.input.keys.X)) {
2115         this.mTextToWork.getXform().incWidthBy(-deltaF);
2116     }
2117     if (engine.input.isKeyPressed(engine.input.keys.Y)) {
2118         this.mTextToWork.getXform().incHeightBy(-deltaF);
2119     }
2120     this.mTextSysFont.setText(
2121         this.mTextToWork.getXform().getWidth().toFixed(2) + "x" +
2122         this.mTextToWork.getXform().getHeight().toFixed(2));
2123 }
```

The listed code shows that you can perform the following operations during runtime:

- a. Select which FontRenderable object to work with based on keyboard 0 to 4 input.
- b. Control the width and height of the selected FontRenderable object when both the left/right arrow and X/Y keys are pressed.

You can now interact with the Font Support project to modify the size of each of the displayed font message and to move the hero toward the left and right.

Summary

2133

In this chapter, you learned how to paste, or texture map, images on unit squares to better represent objects in your games. You also learned how to identify a selected subregion of an image and texture map to the unit square based on the normalize-ranged Texture Coordinate System. The chapter then explained how sprite sheets can reduce the time required for loading texture images while facilitating the creation of animations. This knowledge was then generalized and applied to the drawing of bitmap fonts.

2134

2135

2136

2137

2138

2139

The implementation of texture mapping and sprite sheet rendering takes advantage of an important aspect of game engine architecture: the SimpleShader/Renderable object pair where JavaScript SimpleShader objects are defined to interface with corresponding GLSL shaders and Renderable objects to facilitate the creation and interaction with multiple object instances. For example, you created TextureShader to interface with TextureVS and TextureFS GLSL shaders and created TextureRenderable for the game programmers to work with. This same pattern is repeated for SpriteShader and SpriteRenderable. The experience from SpriteShader objects paired with SpriteAnimateRenderable shows that, when appropriate, the same shader object can support multiple renderable object types in the game engine. This SimpleShader/Renderable pair implementation pattern will appear again in Chapter 8, when you learn to create 3D illumination effects.

2140

2141

2142

2143

2144

2145

2146

2147

2148

2149

2150

2151

At the beginning of this chapter, your game engine supports the player manipulating objects with the keyboard and the drawing of these objects in various sizes and orientations. With the functionality from this chapter, you can now represent these objects with interesting images and create animations of these objects when desired. In the next chapter, you will learn about defining and supporting behaviors for these objects including pseudo autonomous behaviors such as chasing and collision detections.

2152

2153

2154

2155

2156

2157

2158

Game Design Considerations

2159

In Chapter 4, you learned how responsive game feedback is essential for making players feel connected to a game world and that this sense of connection is known as *presence* in game design. As you move through future chapters in this book, you'll notice that most game design is ultimately focused on enhancing the sense of presence in one way or another, and you'll discover that visual design is one of the most important contributors to presence. Imagine, for example, a game where an object controlled by the player

2160

2161

2162

2163

2164

2165

CHAPTER 5 WORKING WITH TEXTURES, SPRITES, AND FONTS

2166 (referred to as the *hero* moving forward) must maneuver through a 2D platformer-style
2167 game world; the player's goal might be to use the mouse and keyboard to jump the hero
2168 between individual surfaces rendered in the game without falling through gaps that exist
2169 between those surfaces. The visual representation of the hero and other objects in the
2170 environment determines how the player identifies with the game setting, which in turn
2171 determines how effectively the game creates presence: Is the hero represented as a living
2172 creature or just an abstract shape like a square or circle? Are the surfaces represented as
2173 building rooftops, as floating rocks on an alien planet, or simply as abstract rectangles?
2174 There is no right or wrong answer when it comes to selecting a visual representation or
2175 game setting, but it is important to design a visual style for all game elements that feels
2176 unified and integrated into whatever game setting you choose (e.g., abstract rectangle
2177 platforms may negatively impact presence if your game setting is a tropical rainforest).

2178 The Texture Shaders project demonstrated how .png images, with transparency,
2179 more effectively integrate game elements into the game environment than formats like
2180 .jpg that don't support transparency. If you move the hero (represented here as simply
2181 a rectangle) to the right, nothing on the screen changes, but if you move the hero to the
2182 left, you'll eventually trigger a state change that alters the displayed visual elements as
2183 you did in the Scene Objects project from Chapter 4. Notice how much more effectively
2184 the robot sprites are integrated into the game scene when they're .png files with
2185 transparency on the gray background compared to when they're .jpg images without
2186 transparency on the blue background.

2187 The Sprite Shaders project introduces a hero that more closely matches other
2188 elements in the game setting: you've replaced the rectangle from the Texture Shaders
2189 project with a humanoid figure stylistically matched to the flying robots on the screen,
2190 and the area of the rectangular hero image not occupied by the humanoid figure is
2191 transparent. If you were to combine the hero from the Sprite Shaders project with the
2192 screen-altering action in the Texture Shaders project, imagine that as the hero moves
2193 toward the robot on the right side of the screen, the robot might turn red when the hero
2194 gets too close. The coded events are still simple at this point, but you can see how the
2195 visual design and a few simple triggered actions can already begin to convey a game
2196 setting and enhance presence.

2197 Note that as game designers, we often become enamored with highly detailed and
2198 elaborate visual designs, and we begin to believe that higher fidelity and more elaborate
2199 visual elements are required to make the best games; this drive for ever-more powerful
2200 graphics is the familiar race that many AAA games engage in with their competition.

While it's true that game experiences and the sense of presence can be considerably
enhanced when paired with excellent art direction, excellence does not always require
elaborate and complex. Great art direction relies on developing a unified visual language
where all elements harmonize with each other and contribute to driving the game
forward and that harmony can be achieved with anything from simple shapes and colors
in a 2D plane to hyperreal 3D environments and every combination in between.

2201
2202
2203
2204
2205
2206

Adding animated motion to the game's visual elements can further enhance game
presence because animation brings a sense of cinematic dynamism to gameplay
that further connects players to the game world. We typically experience motion in
our world as interconnected systems; when you walk across the room, for example,
you don't just glide without moving your body but move different parts of your body
together in different ways. By adding targeted animations to objects onscreen that
cause those objects to behave in ways you might expect complex systems to move or
act, you connect players in a more immersive and convincing way to what's going on in
the game world. The Sprite Animation project demonstrates how animation increases
presence by allowing you to articulate the flying robot's spikes, controlling direction
and speed. Imagine again combining the Sprite Animation project with the earlier
projects in this chapter; as the hero moves closer to the robot, it might first turn red,
eventually triggering the robot's animations and moving it either toward or away from
the player. Animations often come fairly late in the game design process because it's
usually necessary to first have the game mechanic and other systems well defined to
avoid time-consuming changes that may be required as environments and level designs
are updated. Designers typically use simple placeholder assets in the early stages of
development, adding polished and animated final assets only when all of the other
elements of gameplay have been finalized to minimize the need for rework.

2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225

As was the case with visual design, the animation approach need not be complex
to be effective. While animation needs to be intentional and unified and should feel
smooth and stutter-free unless it's intentionally designed to be otherwise, a wide degree
of artistic license can be employed in how movement is represented onscreen.

2226
2227
2228
2229

The Font Support project introduced you to game fonts. While fonts rarely have a
direct impact on gameplay, they can have a dramatic impact on presence. Fonts are
a form of visual communication, and the style of the font is often as important as the
words it conveys in setting tone and mood and can either support or detract from the
game setting and visual style. Pay particular attention to the fonts displayed in this
project, and note how the yellow font conveys a digital feeling that's matched to the

2230
2231
2232
2233
2234
2235

2236 science fiction–inspired visual style of the hero and robots, while the Consolas font
2237 family with its round letterforms feels a bit out of place with this game setting (sparse
2238 though the game setting may still be). As a more extreme example, imagine how
2239 disconnected a flowing calligraphic script font (the type typically used in high-fantasy
2240 games) would appear in a futuristic game that takes place on a spaceship.

2241 There are as many visual style possibilities for games as there are people and ideas,
2242 and great games can feature extremely simple graphics. Remember that excellent game
2243 design is a combination of the nine contributing elements (return to the introduction if
2244 you need to refresh your memory), and the most important thing to keep in mind as a
2245 game designer is maintaining focus on how each of those elements harmonizes with and
2246 elevates the others to create something greater than the sum of its parts.

Author Query

Chapter No.: 5 0005244098

Queries	Details Required	Author's Response
AU1	Please check if edit to sentence starting "The variable left..." is okay.	

Uncorrected Proof

CHAPTER 6

Defining Behaviors and Detecting Collisions

After completing this chapter, you will be able to

- Implement autonomous, controlled, gradual turning, and target-locked chasing behaviors
- Collide textured objects accurately
- Understand the efficiency concerns of pixel-accurate collision
- Program with pixel-accurate collision effectively and efficiently

Introduction

By this point, your game engine is capable of implementing games in convenient coordinate systems as well as presenting and animating objects that are visually appealing. However, there is a lack of abstraction support for the behaviors of objects. You can see the direct results of this shortcoming in the `init()` and `update()` functions of the `MyGame` objects in all the previous projects: the `init()` function is often crowded with mundane per-game object settings, while the `update()` function is often crowded with conditional statements for controlling objects, such as checking for key presses for moving the hero.

A well-designed system should hide the initialization and controls of individual objects with proper object-oriented abstractions or classes. An abstract `GameObject` class should be introduced to encapsulate and hide the specifics of its initialization and behaviors. There are two main advantages to this approach. First, the `init()` and `update()` functions of a game level can focus on managing individual game object

CHAPTER 6 DEFINING BEHAVIORS AND DETECTING COLLISIONS

and the interactions of these objects without being clustered with details specific to different types of objects. Second, as you have experienced with the Renderable and SimpleShader class hierarchies, proper object-oriented abstraction creates a standardized interface and facilitates code sharing and reuse.

As you transition from working with the mere drawing of objects (in other words, Renderable) to programming with the behavior of objects (in other words, GameObject), you will immediately notice that for the game to be entertaining or fun, the objects need to interact. Interesting behaviors of objects, such as facing or evading enemies, often require the knowledge of the relative positions of other objects in the game. In general, resolving relative positions of all objects in a 2D world is nontrivial. Fortunately, typical video games require the knowledge of only those objects that are in close proximity to each other or are about to collide or have collided.

An efficient but somewhat crude approximation to detect collision is to compute the bounds of an object and approximate object collisions based on colliding bounding boxes. In the simplest cases, bounding boxes are rectangular boxes with edges that are aligned with the x/y axes. These are referred to as axis-aligned bounding boxes or AABBs. Because of the axis alignments, it is computationally efficient to detect when two AABBs overlap or when collision is about to occur.

Many 2D game engines can also detect the actual collision between two textured objects by comparing the location of pixels from both objects and detecting the situation when at least one of the nontransparent pixels overlaps. This computationally intensive process is known as per-pixel-accurate collision detection, pixel-accurate collision, or per-pixel collision.

This chapter begins by introducing the GameObject class to provide a platform for abstracting game object behaviors. The GameObject class is then generalized to introduce common behavior attributes including speed, movement direction, and targetlocked chasing. The rest of the chapter focuses on deriving an efficient per-pixel accurate collision implementation that supports both textured and animated sprite objects.

Game Objects

As mentioned, an abstraction that encapsulates the intrinsic behavior of typical game objects should be introduced to minimize the clustering of code in the init() and update() functions of a game level and to facilitate reuse. This section introduces the

simple GameObject class to illustrate how the cleaner and uncluttered `init()` and `update()` functions clearly reflect the in-game logic and to demonstrate how the basic platform for abstracting object behaviors facilitates design and code reuse.

57
58
59

The Game Objects Project

60
61
62
63
64
65
66
67

This project defines the simple `GameObject` class as the first step in building an abstraction to represent actual objects with behaviors in a game. You can see an example of this project running in Figure 6-1. Notice the many minions charging from right to left and wrapping around when they reach the left boundary. This project leads you to create the infrastructure to support the many minions while keeping the logic in the `MyGame` level simple. The source code to this project is defined in the `chapter6/6.1.game_objects` folder.



this figure will be printed in b/w

Figure 6-1. Running the Game Objects project

68 The controls of the project are as follows:

- 69 • **WASD keys:** Move the hero up, left, down, and right

70 The goals of the project are as follows:

- 71 • To begin defining the `GameObject` class to encapsulate object
72 behaviors in games
- 73 • To demonstrate the creation of subclasses to the `GameObject` class to
74 maintain the simplicity of the `MyGame` level `update()` function
- 75 • To introduce the `GameObjectSet` class demonstrating support for a
76 set of homogenous objects with an identical interface

77 You can find the following external resource file in the assets folder: `minion_`
78 `sprite.png`; you'll also find the fonts folder that contains the default system fonts. Note
79 that, as shown in Figure 6-2, the `minion_sprite.png` image file has been updated from
80 the previous project to include two extra sprite elements: the DyePack and the Brain
81 minion.

this figure will be printed in b/w

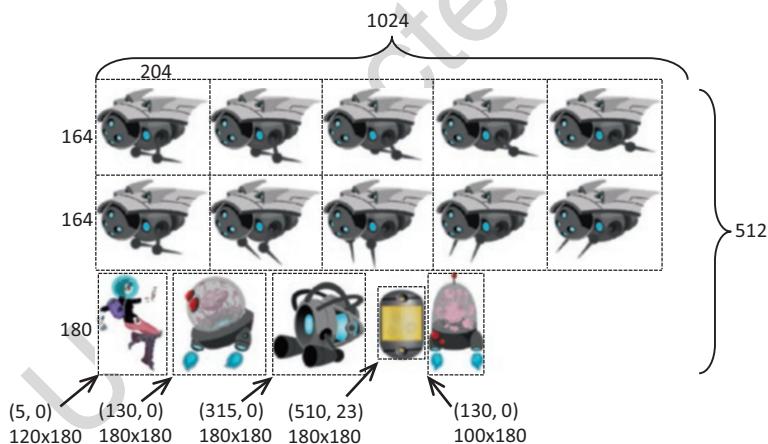


Figure 6-2. The new sprite elements of the `minion_sprite.png` image

82 Define the `GameObject` Class

83 The goal is to define a logical abstraction to encapsulate all relevant behavioral
84 characteristics of a typical object in a game including the ability to control positions,
85 drawing, and so on. As in the case for the Scene objects in the earlier chapter, the main
86 result is to provide a well-defined interface governing the functions that subclasses

implement. The more sophisticated behaviors will be introduced in the next section. 87
 This example only demonstrates the potential of the `GameObject` class with minimal 88
 behaviors defined. 89

1. Add a new folder `src/engine/game_objects` for storing 90
`GameObject`-related files. 91
2. Create a new file in this folder, name it `game_object.js`, and add 92
 the following code: 93

```
class GameObject { 94
  constructor(renderable) { 95
    this.mRenderComponent = renderable; 96
  } 97

  getXform() { return this.mRenderComponent.getXform(); } 98

  getRenderable() { return this.mRenderComponent; } 99

  update() { } 100

  draw(aCamera) { 101
    this.mRenderComponent.draw(aCamera); 102
  } 103
} 104

export default GameObject; 105
```

With the assessors to the `Renderable` and `Transform` objects defined, all `GameObject` 106
 instances can be drawn and have defined locations and sizes. Note that the `update()` 107
 function is designed for subclasses to override with per object-specific behaviors, and 108
 thus, it is left empty. 109

Manage Game Objects in Sets

Because most games consist of many interacting objects, it is useful to define a utility class to support working with a set of `GameObject` instances:

1. Create a new file in the `src/engine/game_objects` folder and name it `game_object_set.js`. Define the `GameObjectSet` class and the constructor to initialize an array for holding `GameObject` instances.

```
117 class GameObjectSet {  
118     constructor() {  
119         this.mSet = [];  
120     }  
121     ... implementation to follow ...  
122     export default GameObjectSet;
```

2. Define functions for managing the set membership:

```
124     size() { return this.mSet.length; }  
125     getObjectAt(index) { return this.mSet[index]; }  
126     addToSet(obj) { this.mSet.push(obj); }  
127     removeFromSet(obj) {  
128         let index = this.mSet.indexOf(obj);  
129         if (index > -1)  
130             this.mSet.splice(index, 1);  
131     }
```

3. Define functions to update and draw each of the `GameObject` instances in the set:

```
134     update() {  
135         let i;  
136         for (i = 0; i < this.mSet.length; i++) {  
137             this.mSet[i].update();  
138         }  
139     }
```

```

draw(aCamera) { 140
    let i; 141
    for (i = 0; i < this.mSet.length; i++) { 142
        this.mSet[i].draw(aCamera); 143
    } 144
} 145

```

Export the Classes to the Client

The last step in integrating any new functionality into the engine involves modifying the engine access file, `index.js`. Edit `index.js` and add the following import and export statements to grant the client access to the `GameObject` and `GameObjectSet` classes:

```

... identical to previous code ... 150

// game objects 151
import GameObject from "./game_objects/game_object.js"; 152
import GameObjectSet from "./game_objects/game_object_set.js"; 153

... identical to previous code ... 154

export default { 155
    ... identical to previous code ... 156

    // Game Objects 157
    GameObject, GameObjectSet, 158
    ... identical to previous code ... 159
} 160

```

Note This process of import/export classes via the engine access file, `index.js`, must be repeated for every newly defined functionality. Henceforth, only a reminder will be provided and the straightforward code change will not be shown again.

164 Test the **GameObject** and **GameObjectSet**

165 The goals of this project are to ensure proper functioning of the new **GameObject** class,
 166 to demonstrate customization of behaviors by individual object types, and to observe
 167 a cleaner **MyGame** implementation clearly reflecting the in-game logic. To accomplish
 168 these goals, three object types are defined: **DyePack**, **Hero**, and **Minion**. Before you begin
 169 to examine the detailed implementation of these objects, follow good source code
 170 organization practice and create a new folder `src/my_game/objects` for storing the new
 171 object types.

172 The **DyePack** **GameObject**

173 The **DyePack** class derives from the **GameObject** class to demonstrate the most basic
 174 example of a **GameObject**: an object that has no behavior and is simply drawn to the
 175 screen.

176 Create a new file in the `src/my_game/objects` folder and name it `dye_pack.js`.
 177 Import from the engine access file, `index.js`, to gain access to all of the game engine
 178 functionality. Define **DyePack** as a subclass of **GameObject** and implement the
 179 constructor as follows:

```
180 import engine from "../../engine/index.js";
181 class DyePack extends engine.GameObject {
182   constructor(spriteTexture) {
183     super(null);
184     this.kRefWidth = 80;
185     this.kRefHeight = 130;
186     this.mRenderComponent =
187       new engine.SpriteRenderable(spriteTexture);
188     this.mRenderComponent.setColor([1, 1, 1, 0.1]);
189     this.mRenderComponent.getXform().setPosition(50, 33);
190     this.mRenderComponent.getXform().setSize(
191       this.kRefWidth / 50, this.kRefHeight / 50);
192     this.mRenderComponent.setElementPixelPositions(510, 595, 23, 153);
193   }
194 }
195 export default DyePack;
```

Notice that even without specific behaviors, the DyePack is implementing code that
 used to be found in the `init()` function of the MyGame level. In this way, the DyePack
 object hides specific geometric information and simplifies the MyGame level.

Note The need to import from the engine access file, `index.js`, is true for
 almost all client source code file and will not be repeated.

The Hero GameObject

The Hero class supports direct user keyboard control. This object demonstrates hiding of
 game object control logic from the `update()` function of MyGame.

1. Create a new file in the `src/my_game/objects` folder and name it
`hero.js`. Define Hero as a subclass of `GameObject`, and implement
 the constructor to initialize the sprite UV values, size, and
 position. Make sure to export and share this class.

```
class Hero extends engine.GameObject {  

    constructor(spriteTexture) {  

        super(null);  

        this.kDelta = 0.3;  

        this.mRenderComponent =  

            new engine.SpriteRenderable(spriteTexture);  

        this.mRenderComponent.setColor([1, 1, 1, 0]);  

        this.mRenderComponent.getXform().setPosition(35, 50);  

        this.mRenderComponent.getXform().setSize(9, 12);  

        this.mRenderComponent.setElementPixelPositions(0, 120, 0, 180);  

    }  

    ... implementation to follow ...  

    export default Hero;
```

2. Add a function to support the update of this object by user
 keyboard control. The Hero object moves at a `kDelta` rate based
 on WASD input from the keyboard.

```

224 update() {
225     // control by WASD
226     let xform = this.getXform();
227     if (engine.input.isKeyPressed(engine.input.keys.W)) {
228         xform.incYPosBy(this.kDelta);
229     }
230     if (engine.input.isKeyPressed(engine.input.keys.S)) {
231         xform.incYPosBy(-this.kDelta);
232     }
233     if (engine.input.isKeyPressed(engine.input.keys.A)) {
234         xform.incXPosBy(-this.kDelta);
235     }
236     if (engine.input.isKeyPressed(engine.input.keys.D)) {
237         xform.incXPosBy(this.kDelta);
238     }
239 }
```

240 The Minion GameObject

241 The Minion class demonstrates that simple autonomous behavior can also be hidden:

- 242 Create a new file in the `src/my_game/objects` folder and name
243 it `minion.js`. Define Minion as a subclass of `GameObject`, and
244 implement the constructor to initialize the sprite UV values, sprite
245 animation parameters, size, and position as follows:

```

246 class Minion extends engine.GameObject {
247     constructor(spriteTexture, atY) {
248         super(null);
249         this.kDelta = 0.2;
250
251         this.mRenderComponent =
252             new engine.SpriteAnimateRenderable(spriteTexture);
253
254         this.mRenderComponent.setColor([1, 1, 1, 0]);
255         this.mRenderComponent.getXform().setPosition(
256             Math.random() * 100, atY);
257         this.mRenderComponent.getXform().setSize(12, 9.6);
```

```

// first element pixel position: top-left 512 is top of image      256
// 0 is left of the image                                         257
this.mRenderComponent.setSpriteSequence(512, 0,                  258
    204, 164, // widthxheight in pixels                         259
    5,          // number of elements in this sequence                260
    0);          // horizontal padding in between                   261
this.mRenderComponent.setAnimationType(                           262
    engine.eAnimationType.eSwing);                                263
this.mRenderComponent.setAnimationSpeed(15);                      264
// show each element for mAnimSpeed updates                     265
}
... implementation to follow ...                                267
}

export default Minion;                                         268
269

```

2. Add a function to update the sprite animation, support the simple right-to-left movements, and provide the wrapping functionality:

```

update() {                                                 272
    // remember to update this.mRenderComponent's animation        273
    this.mRenderComponent.updateAnimation();                      274

    // move towards the left and wraps                          275
    let xform = this.getXform();
    xform.incXPosBy(-this.kDelta);

    // if fly off to the left, re-appear at the right           278
    if (xform.getXPos() < 0) {
        xform.setXPos(100);
        xform.setYPos(65 * Math.random());
    }
}

```

284 The MyGame Scene

285 As in all cases, the MyGame level is implemented in the `my_game.js` file. With the three
 286 specific `GameObject` subclasses defined, follow these steps:

- 287 1. In addition to the engine access file, `index.js`, in order to gain
 288 access to the newly defined objects, the corresponding source
 289 code must be imported:

```
290 import engine from "../engine/index.js";
291 // user stuff
292 import DyePack from "./objects/dye_pack.js";
293 import Minion from "./objects/minion.js";
294 import Hero from "./objects/hero.js";
```

295 **Note** As is the case for other import/export statements, unless there are other
 296 specific reasons, this reminder will not be shown again.

- 297 2. The constructor and the `load()`, `unload()`, and `draw()` functions
 298 are similar as in previous projects, so the details are not shown
 299 here.
- 300 3. Edit the `init()` function and add the following code:

```
301 init() {
302     ... identical to previous code ...
303
304     // Step B: The dye pack: simply another GameObject
305     this.mDyePack = new DyePack(this.kMinionSprite);
306
307     // Step C: A set of Minions
308     this.mMinionset = new engine.GameObjectSet();
309     let i = 0, randomY, aMinion;
310     // create 5 minions at random Y values
311     for (i = 0; i < 5; i++) {
312         randomY = Math.random() * 65;
```

```

aMinion = new Minion(this.kMinionSprite, randomY);          311
this.mMinionset.addToSet(aMinion);                          312
}
// Step D: Create the hero object                         314
this.mHero = new Hero(this.kMinionSprite);                  315

// Step E: Create and initialize message output          316
this.mMsg = new engine.FontRenderable("Status Message"); 317
this.mMsg.setColor([0, 0, 0, 1]);                          318
this.mMsg.getXform().setPosition(1, 2);                   319
this.mMsg.setTextHeight(3);                               320
}

```

The details of step A, the creation of the camera and initialization of the background color, are not shown because they are identical to previous projects. Steps B, C, and D show the instantiation of the three object types, with step C showing the creation and insertion of the right-to-left moving Minion objects into the mMinionset, an instance of the GameObjectSet class. Notice that the init() function is free from the clustering of setting each object's textures, geometries, and so on.

4. Edit the update() function to update the game state:

```

update() {                                              329
    this.mHero.update();                                330
    this.mMinionset.update();                           331
    this.mDyePack.update();                            332
}

```

With the well-defined behaviors for each object type abstracted, the clean update() function clearly shows that the game consists of three noninteracting objects.

Observation

You can now run the project and notice that the slightly more complex movements of six minions are accomplished with much cleaner init() and update() functions. The init() function consists of only logic and controls for placing created objects in the game world and does not include any specific settings for different object types. With the Minion object defining its motion behaviors in its own update() function, the logic in the

342 MyGame update() function can focus on the details of the level. Note that the structure of
343 this function clearly shows that the three objects are updated independently and do not
344 interact with each other.

345 **Note** Throughout this book, in almost all cases, MyGame classes are designed
346 to showcase the engine functionality. As a result, the source code organization
347 in most MyGame classes may not represent the best practices in implementing
348 games.

349 Creating a Chase Behavior

350 A closer examination of the previous project reveals that though there are quite a few
351 minions moving on the screen, their motions are simple and boring. Even though there
352 are variations in speed and direction, the motions are without purpose or awareness
353 of other game objects in the scene. To support more sophisticated or interesting
354 movements, a GameObject needs to be aware of the locations of other objects and
355 determine motion based on that information.

356 Chasing behavior is one such example. The goal of a chasing object is usually to catch
357 the game object that it is targeting. This requires programmatic manipulation of the front
358 direction and speed of the chaser such that it can hone in on its target. However, it is
359 generally important to avoid implementing a chaser that has perfect aim and always hits
360 its target—because if the player is unable to avoid being hit, the game becomes impossibly
361 difficult. Nonetheless, this does not mean you should not implement a perfect chaser if
362 your game design requires it. You will implement a chaser in the next project.

363 Vectors and the associated operations are the foundation for implementing object
364 movements and behaviors. Before programming with vectors, a quick review is provided.
365 As in the case of matrices and transform operators, the following discussion is not meant
366 to be a comprehensive coverage of vectors. Instead, the focus is on the application
367 of a small collection of concepts that are relevant to the implementation of the game
368 engine. This is not a study of the theories behind the mathematics. If you are interested
369 in the specifics of vectors and how they relate to games, please refer to the discussion in
370 Chapter 1 where you can learn more about these topics in depth by delving into relevant
371 books on linear algebra and games.

Vectors Review

372

Vectors are used across many fields of study, including mathematics, physics, computer science, and engineering. They are particularly important in games; nearly every game uses vectors in one way or another. Because they are used so extensively, this section is devoted to understanding and utilizing vectors in games.

373

374

375

376

Note For an introductory and comprehensive coverage of vectors, you can refer to www.storyofmathematics.com/vectors. For more detailed coverage of vector applications in games, you can refer to *Basic Math for Game Development with Unity 3D: A Beginner's Guide to Mathematical Foundations*, Apress, 2019.

377

378

379

380

One of the most common uses for vectors is to represent an object's displacement and direction or *velocity*. This can be done easily because a vector is defined by its size and direction. Using only this small amount of information, you can represent attributes such as the velocity or acceleration of an object. If you have the position of an object, its direction, and its velocity, then you have sufficient information to move it around the game world without user input.

381

382

383

384

385

386

Before going any further, it is important to review the concepts of a vector, starting with how you can define one. A vector can be specified using two points. For example, given the arbitrary positions $P_a = (x_a, y_a)$ and $P_b = (x_b, y_b)$, you can define the vector from P_a to P_b or \vec{V}_{ab} as $P_b - P_a$. You can see this represented in the following equations and Figure 6-3:

387

388

389

390

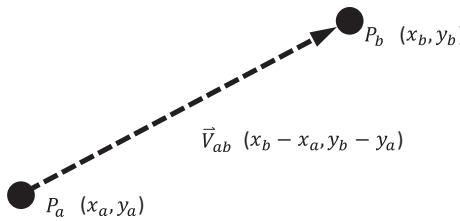
391

- $P_a = (x_a, y_a)$
- $P_b = (x_b, y_b)$
- $\vec{V}_{ab} = P_b - P_a = (x_b - x_a, y_b - y_a)$

392

393

394

**Figure 6-3.** A vector defined by two points

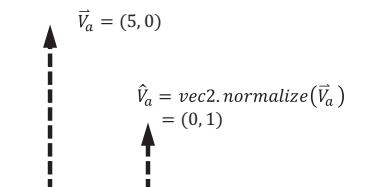
Now that you have a vector \vec{V}_{ab} , you can easily determine its length (or size) and direction. A vector's length is equal to the distance between the two points that created it. In this example, the length of \vec{V}_{ab} is equal to the distance between P_a and P_b , while the direction of \vec{V}_{ab} goes from P_a toward P_b .

Note The size of a vector is often referred to as its length or *magnitude*.

In the `gl-matrix` library, the `vec2` object implements the functionality of a 2D vector. Conveniently, you can also use the `vec2` object to represent 2D points or positions in space. In the preceding example, P_a , P_b , and \vec{V}_{ab} can all be implemented as instances of the `vec2` object. However, \vec{V}_{ab} is the only mathematically defined vector. P_a and P_b represent positions or points used to create a vector.

Recall that a vector can also be normalized. A *normalized* vector (also known as a *unit vector*) always has a size of 1. You can see a normalized vector by the following function, as shown in Figure 6-4. Notice that the mathematical symbol for a regular vector is \vec{V}_a and for a normalized vector is \hat{V}_a :

- `vec2.normalized(\vec{V}_a)`: Normalizes vector \vec{V}_a and stores the results to the `vec2` object

**Figure 6-4.** A vector being normalized

Vectors to a position can also be rotated. If, for example, the vector $\vec{V} = (x_v, y_v)$ represents the direction from the origin to the position (x_v, y_v) and you want to rotate it by θ , then, as illustrated in Figure 6-5, you can use the following equations to derive x_r and y_r :

- $x_r = x_v \cos \theta - y_v \sin \theta$

- $y_r = x_v \sin \theta + y_v \cos \theta$

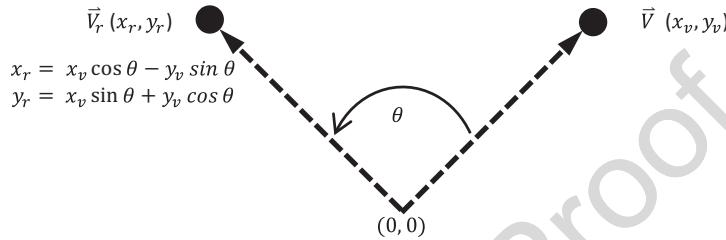


Figure 6-5. A vector from the origin to the position (x_v, y_v) being rotated by the angle theta

Note JavaScript trigonometric functions, including the `Math.sin()` and `Math.cos()` functions, assume input to be in radians and not degrees. Recall that 1 degree is equal to $\frac{\pi}{180}$ radians.

It is always important to remember that vectors are defined by their direction and size. In other words, two vectors can be equal to each other independent of the locations of the vectors. Figure 6-6 shows two vectors \vec{V}_a and \vec{V}_{bc} that are located at different positions but have the same direction and magnitude and thus are equal to each other. In contrast, the vector \vec{V}_d is not the same because its direction and magnitude are different from the others.

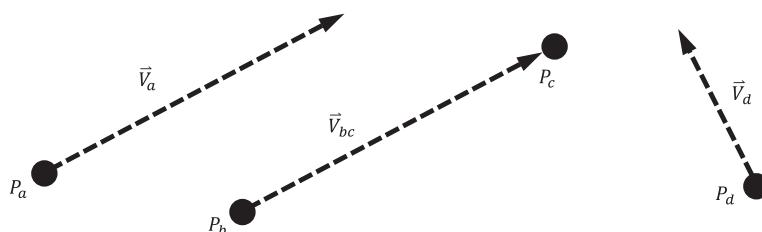


Figure 6-6. Three vectors represented in 2D space with two vectors equal to each other

425 The Dot Product

426 The dot product of two normalized vectors provides you with the means to find the angle
 427 between those vectors. For example, given the following:

428 • $\vec{V}_1 = (x_1, y_1)$

429 • $\vec{V}_2 = (x_2, y_2)$

430 Then the following is true:

431 • $\vec{V}_1 \cdot \vec{V}_2 = \vec{V}_2 \cdot \vec{V}_1 = x_1 x_2 + y_1 y_2$.

432 Additionally, if both vectors \vec{V}_1 and \vec{V}_2 are normalized, then

433 • $\hat{\vec{V}}_1 \cdot \hat{\vec{V}}_2 = \cos\theta$

434 Figure 6-7 depicts an example of the \vec{V}_1 and \vec{V}_2 vectors with an angle θ in between
 435 them. It is also important to recognize that if $\vec{V}_1 \cdot \vec{V}_2 = 0$, then the two vectors are
 436 perpendicular.

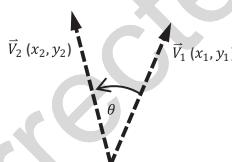


Figure 6-7. The angle between two vectors, which can be found through the dot product

this figure will be printed in b/w

437 **Note** If you need to review or refresh the concept of a dot product, please refer to
 438 www.mathsisfun.com/algebra/vectors-dot-product.html.

439 The Cross Product

440 The cross product of two vectors produces a vector that is *orthogonal*, or perpendicular,
 441 to both of the original vectors. In 2D games, where the 2D dimensions lie flat on the
 442 screen, the result of the cross product is a vector that points either inward (toward the
 443 screen) or outward (away from the screen). This may seem odd because it is not intuitive
 444 that crossing two vectors in 2D or the x/y plane results in a vector that lies in the third

dimension or along the z axis. However, the resulting vector in the third dimension carries crucial information. For example, the direction of this vector in the third dimension can be used to determine whether the game object needs to rotate in the clockwise or counterclockwise direction. Take a closer look at the following:

- $\vec{V}_1 = (x_1, y_1)$
- $\vec{V}_2 = (x_2, y_2)$

Given the previous, the following is true:

- $\vec{V}_3 = \vec{V}_1 \times \vec{V}_2$ is a vector perpendicular to both \vec{V}_1 and \vec{V}_2 .

Additionally, you know that the cross product of two vectors on the x/y plane results in a vector in the z direction. When $\vec{V}_1 \times \vec{V}_2 > 0$, you know that \vec{V}_1 is in the clockwise direction from \vec{V}_2 ; similarly, when $\vec{V}_1 \times \vec{V}_2 < 0$, you know that \vec{V}_1 is in the counterclockwise direction. Figure 6-8 should help clarify this concept.

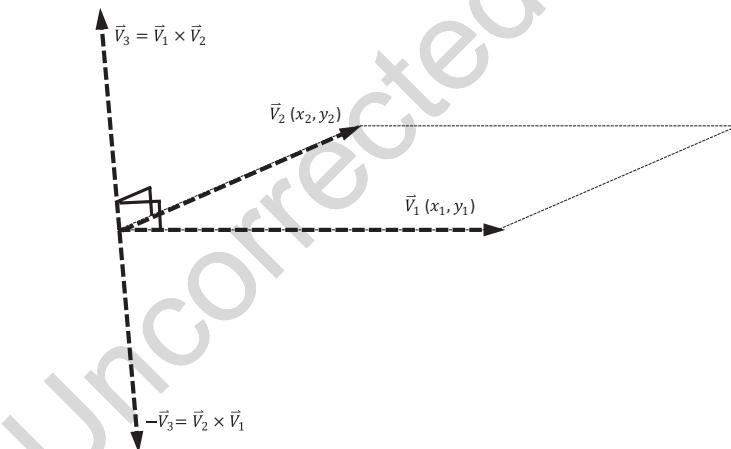


Figure 6-8. The cross product of two vectors

this figure will be printed in b/w

Note If you need to review or refresh the concept of a cross product, please refer to www.mathsisfun.com/algebra/vectors-cross-product.html.

The Front and Chase Project

457

458 This project implements more interesting and sophisticated behaviors based on the
459 vector concepts that have been reviewed. Instead of constant and aimless motions, you
460 will experience the process of defining and varying the front direction of an object and
461 guiding an object to chase after another object in the scene. You can see an example
462 of this project running in Figure 6-9. The source code to this project is defined in the
463 chapter6/6.2.front_and_chase folder.

this figure will be printed in b/w



Figure 6-9. Running the Front and Chase project

464 The controls of the project are as follows:

- 465 • **WASD keys:** Moves the Hero object
- 466 • **Left-/right-arrow keys:** Change the front direction of the Brain
467 object when it is under user control

- **Up-/down-arrow keys:** Increase/decrease the speed of the Brain object 468
object 469
- **H key:** Switches the Brain object to be under user arrow keys control 470
- **J key:** Switches the Brain object to always point at and move toward the current Hero object position 471
472
- **K key:** Switches the Brain object to turn and move gradually toward the current Hero object position 473
474

The goals of the project are as follows:

- To experience working with speed and direction 476
- To practice traveling along a predefined direction 477
- To implement algorithms with vector dot and cross products 478
- To examine and implement chasing behavior 479

You can find the same external resource files as in the previous project in the assets folder.

Add Vector Rotation to the gl-matrix Library

The gl-matrix library does not support rotating a position in 2D space. This can be rectified by adding the following code to the `gl-matrix.js` file in the `lib` folder:

```
vec2.rotate = function(out, a, c){  
    var r=[];  
    // perform rotation  
    r[0] = a[0]*Math.cos(c) - a[1]*Math.sin(c);  
    r[1] = a[0]*Math.sin(c) + a[1]*Math.cos(c);  
    out[0] = r[0];  
    out[1] = r[1];  
    return r;  
};
```

Note This modification to the gl-matrix library must be present in all projects from this point forward.

494

495

496 Modify GameObject to Support Interesting Behaviors

497 The GameObject class abstracts and implements the desired new object behaviors:

- 498 1. Edit the game_object.js file and modify the GameObject
- 499 constructor to define visibility, front direction, and speed:

```
500 constructor(renderable) {
501     this.mRenderComponent = renderable;
502     this.mVisible = true;
503     this.mCurrentFrontDir = vec2.fromValues(0, 1); // front direction
504     this.mSpeed = 0;
505 }
```

- 506 2. Add assessor and setter functions for the instance variables:

```
507 getXform() { return this.mRenderComponent.getXform(); }

508 setVisibility(f) { this.mVisible = f; }
509 isVisible() { return this.mVisible; }

510 setSpeed(s) { this.mSpeed = s; }
511 getSpeed() { return this.mSpeed; }
512 incSpeedBy(delta) { this.mSpeed += delta; }

513 setCurrentFrontDir(f) { vec2.normalize(this.mCurrentFrontDir, f); }
514 getCurrentFrontDir() { return this.mCurrentFrontDir; }

515 getRenderable() { return this.mRenderComponent; }
```

- 516 3. Implement a function to rotate the front direction toward a
- 517 position, p:

```
518 rotateObjPointTo(p, rate) {
519     // Step A: determine if reached the destination position p
520     let dir = [];
521     vec2.sub(dir, p, this.getXform().getPosition());
522     let len = vec2.length(dir);
523     if (len < Number.MIN_VALUE) {
524         return; // we are there.
525     }
```

```

vec2.scale(dir, dir, 1 / len);                                526

// Step B: compute the angle to rotate                         527
let fdir = this.getCurrentFrontDir();                          528
let cosTheta = vec2.dot(dir, fdir);                           529

if (cosTheta > 0.999999) { // almost exactly the same direction 530
    return;
}                                                               531

// Step C: clamp the cosTheta to -1 to 1                      532
// in a perfect world, this would never happen! BUT ...
if (cosTheta > 1) {                                         533
    cosTheta = 1;                                           534
} else {
    if (cosTheta < -1) {                                     535
        cosTheta = -1;                                         536
    }
}                                                               537

// Step D: compute whether to rotate clockwise, or counterclockwise 538
let dir3d = vec3.fromValues(dir[0], dir[1], 0);                539
let f3d = vec3.fromValues(fdir[0], fdir[1], 0);                540
let r3d = [];                                                 541
vec3.cross(r3d, f3d, dir3d);

let rad = Math.acos(cosTheta); // radian to roate            542
if (r3d[2] < 0) {                                         543
    rad = -rad;                                            544
}

// Step E: rotate the facing direction with the angle and rate 545
rad *= rate; // actual angle need to rotate from Obj's front 546
vec2.rotate(this.getCurrentFrontDir(),this.getCurrentFrontDir(),rad); 547
this.getXform().incRotationByRad(rad);                         548
}

```

CHAPTER 6 DEFINING BEHAVIORS AND DETECTING COLLISIONS

556 The `rotateObjPointTo()` function rotates the `mCurrentFrontDir` to point to the
557 destination position `p` at a rate specified by the parameter `rate`. Here are the details of
558 each operation:

- 559 a. Step A computes the distance between the current object
560 and the destination position `p`. If this value is small, it means
561 current object and the target position are close. The function
562 returns without further processing.
- 563 b. Step B, as illustrated in Figure 6-10, computes the dot product to
564 determine the angle θ between the current front direction of the
565 object (`fdir`) and the direction toward the destination position
566 `p` (`dir`). If these two vectors are pointing in the same direction
567 ($\cos\theta$ is almost 1 or θ almost zero), the function returns.

this figure will be printed in b/w

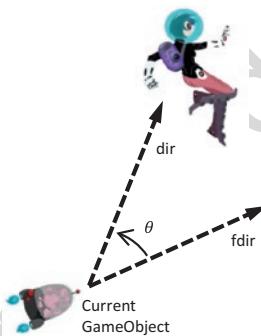


Figure 6-10. A `GameObject` (*Brain*) chasing a target (*Hero*)

- c. Step C checks for the range of $\cos\theta$. This is a step that must be performed because of the inaccuracy of floating-point operations in JavaScript.
- d. Step D uses the results of the cross product to determine whether the current `GameObject` should be turning clockwise or counterclockwise to face toward the destination position `p`.
- e. Step E rotates `mCurrentFrontDir` and sets the rotation in the `Transform` of the `Renderable` object. It is important to recognize the two separate object rotation controls. The `Transform` controls the rotation of what is being drawn, and `mCurrentFrontDir` controls the direction of travel. In this

case, the two are synchronized and thus must be updated
with the new value simultaneously.

4. Add a function to update the object's position with its direction
and speed. Notice that if the `mCurrentFrontDir` is modified by the
`rotateObjPointTo()` function, then this `update()` function will
move the object toward the target position `p`, and the object will
behave as though it is chasing the target.

```
update() {  
    // simple default behavior  
    let pos = this.getXform().getPosition();  
    vec2.scaleAndAdd(pos, pos, this.getCurrentFrontDir(), this.getSpeed());  
}
```

5. Add a function to draw the object based on the visibility setting:

```
draw(aCamera) {  
    if (this.isVisible()) {  
        this.mRenderComponent.draw(aCamera);  
    }  
}
```

Test the Chasing Functionality

The strategy and goals of this test case are to create a steerable Brain object to
demonstrate traveling along a predefined front direction and to direct the Brain to chase
after the Hero to demonstrate the chasing functionality.

Define the Brain GameObject

The Brain object will travel along its front direction under the control of the user's left-/
right-arrow keys for steering:

1. Create a new file in the `src/my_game/objects` folder and name
it `brain.js`. Define Brain as a subclass of `GameObject`, and
implement the constructor to initialize the appearance and
behavior parameters.

CHAPTER 6 DEFINING BEHAVIORS AND DETECTING COLLISIONS

```
597 class Brain extends engine.GameObject {  
598     constructor(spriteTexture) {  
599         super(null);  
600         this.kDeltaDegree = 1;  
601         this.kDeltaRad = Math.PI * this.kDeltaDegree / 180;  
602         this.kDeltaSpeed = 0.01;  
603         this.mRenderComponent =  
604             new engine.SpriteRenderable(spriteTexture);  
605         this.mRenderComponent.setColor([1, 1, 1, 0]);  
606         this.mRenderComponent.getXform().setPosition(50, 10);  
607         this.mRenderComponent.getXform().setSize(3, 5.4);  
608         this.mRenderComponent.setElementPixelPositions(600, 700, 0, 180);  
609         this.setSpeed(0.05);  
610     }  
611     ... implementation to follow ...  
612 }  
613 export default Brain;
```

- 614 2. Override the update() function to support the user steering and
615 controlling the speed. Notice that the default update() function in
616 the GameObject must be called to support the basic traveling of the
617 object along the front direction according to its speed.

```
618 update() {  
619     super.update();  
620     let xf = this.getXform();  
621     let fdir = this.getCurrentFrontDir();  
622     if (engine.input.isKeyPressed(engine.input.keys.Left)) {  
623         xf.incRotationByDegree(this.kDeltaDegree);  
624         vec2.rotate(fdir, fdir, this.kDeltaRad);  
625     }  
626     if (engine.input.isKeyPressed(engine.input.keys.Right)) {  
627         xf.incRotationByRad(-this.kDeltaRad);  
628         vec2.rotate(fdir, fdir, -this.kDeltaRad);  
629     }
```

```

if (engine.input.isKeyClicked(engine.input.keys.Up)) {           630
    this.incSpeedBy(this.kDeltaSpeed);                            631
}
if (engine.input.isKeyClicked(engine.input.keys.Down)) {          632
    this.incSpeedBy(-this.kDeltaSpeed);                           633
}
}                                                               634
}                                                               635
}                                                               636

```

The MyGame Scene

Modify the MyGame scene to test the Brain object movement. In this case, except for the update() function, the rest of the source code in my_game.js is similar to previous projects. For this reason, only the details of the update() function are shown:

```

update() {                                         641
    let msg = "Brain [H:keys J:imm K:gradual]: ";   642
    let rate = 1;                                     643
    this.mHero.update();                             644
    switch (this.mMode) {                           645
        case 'H':
            this.mBrain.update(); // player steers with arrow keys 646
            break;                                647
        case 'K':
            rate = 0.02; // gradual rate           648
            // In gradual mode, the following should also be executed 649
        case 'J':
            this.mBrain.rotateObjPointTo(          650
                this.mHero.getXform().getPosition(), rate); 651
                // the default GameObject: only move forward           652
                engine.GameObject.prototype.update.call(this.mBrain); 653
                break;                                654
    }                                                 655
    if (engine.input.isKeyClicked(engine.input.keys.H)) { 656
        this.mMode = 'H';                            657
    }                                                 658
}
```

```
662     if (engine.input.isKeyPressed(engine.input.keys.J)) {  
663         this.mMode = 'J';  
664     }  
665     if (engine.input.isKeyPressed(engine.input.keys.K)) {  
666         this.mMode = 'K';  
667     }  
668     this.mMsg.setText(msg + this.mMode);  
669 }
```

670 In the update() function, the switch statement uses mMode to determine how to
671 update the Brain object. In the cases of J and K modes, the Brain object turns toward the
672 Hero object position with the rotateObjPointTo() function call. While in the H mode,
673 the Brain object's update() function is called for the user to steer the object with the
674 arrow keys. The final three if statements simply set the mMode variable according to user
675 input.

676 Note that in the cases of J and K modes, in order to bypass the user control logic after
677 the rotateObjPointTo(), the update() function being called is the one defined by the
678 GameObject and not by the Brain.

679 **Note** The JavaScript syntax, `ClassName.prototype.FunctionName.
680 call(anObj)`, calls `FunctionName` defined by `ClassName`, where `anObj` is a
681 subclass of `ClassName`.

682 Observation

683 You can now try running the project. Initially, the Brain object is under the user's
684 control. You can use the left- and right-arrow keys to change the front direction of the
685 Brain object and experience steering the object. Pressing the J key causes the Brain
686 object to immediately point and move toward the Hero object. This is a result of the
687 default turn rate value of 1.0. The K key causes a more natural behavior, where the Brain
688 object continues to move forward and gradually turns to move toward the Hero object.
689 Feel free to change the values of the rate variable or modify the control value of the
690 Brain object. For example, change the kDeltaRad or kDeltaSpeed to experiment with
691 different settings for the behavior.

Collisions Between GameObjects

In the previous project, the Brain object would never stop traveling. Notice that under the J and K modes, the Brain object would orbit or rapidly flip directions when it reaches the target position. The Brain object is missing the critical ability to detect that it has collided with the Hero object, and as a result, it never stops moving. This section describes axis-aligned bounding boxes (AABBs), one of the most straightforward tools for approximating object collisions, and demonstrates the implementation of collision detection based on AABB.

Axis-Aligned Bounding Box (AABB)

An AABB is an x/y axis-aligned rectangular box that bounds a given object. The term *x/y axis aligned* refers to the fact that the four sides of an AABB are parallel either to the horizontal x axis or to the vertical y axis. Figure 6-11 shows an example of representing the bounds to the Hero object by the lower-left corner (`mLL`), width, and height. This is a fairly common way to represent an AABB because it uses only one position and two floating-point numbers to represent the dimensions.

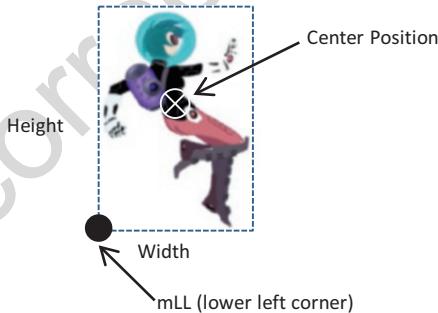


Figure 6-11. The lower-left corner and size of the bounds for an object

It is interesting to note that in addition to representing the bounds of an object, bounding boxes can be used to represent the bounds of any given rectangular area. For example, recall that the WC visible through the Camera is a rectangular area with the camera's position located at the center and the WC width/height defined by the game developer. An AABB can be defined to represent the visible WC rectangular area, or the WC window, and used for detecting collision between the WC window and GameObject instances in the game world.

707

Note In this book, AABB and “bounding box” are used interchangeably.

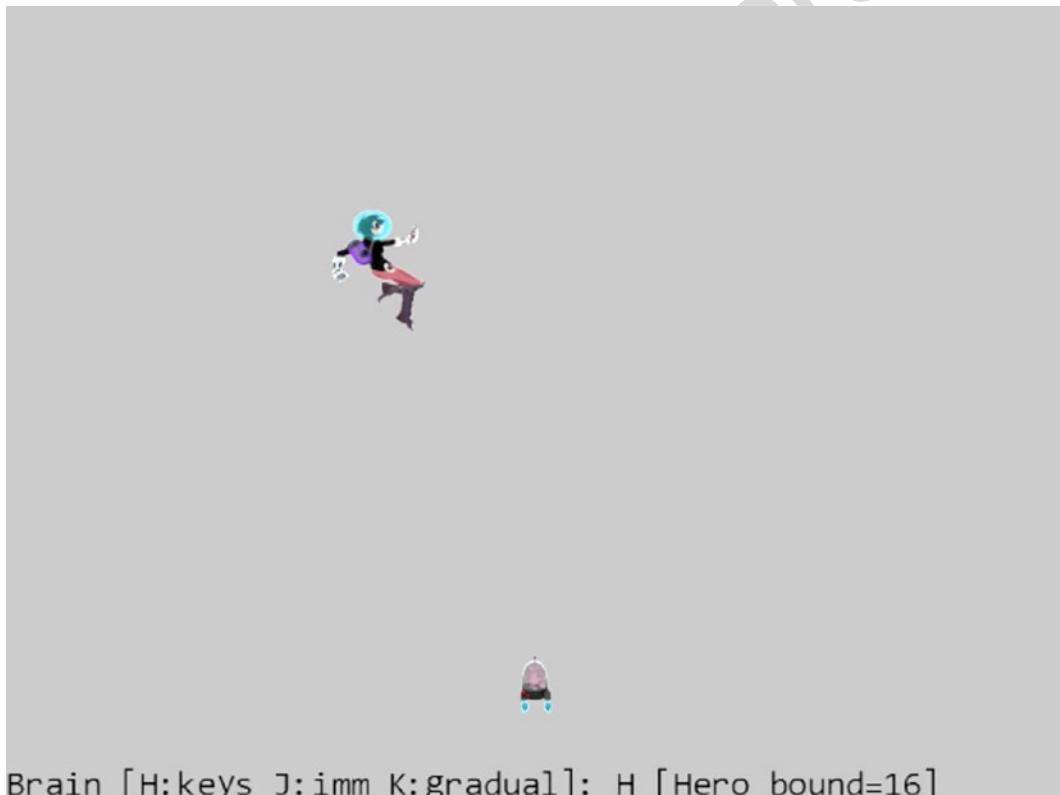
708

The Bounding Box and Collisions Project

709

This project demonstrates how to define a bounding box for a `GameObject` instance and detect collisions between two `GameObject` instances based on their bounding boxes. It is important to remember that bounding boxes are axes aligned, and thus, the solution presented in this section does not support collision detections between rotated objects. You can see an example of this project running in Figure 6-12. The source code to this project is defined in the `chapter6/6.3.bbox_and_collisions` folder.

this figure will be printed in b/w



```
Brain [H:keys J:imm K:gradual]: H [Hero bound=16]
```

Figure 6-12. Running the Bounding Box and Collisions project

The controls of the project are identical to the previous project:

- **WASD keys:** Moves the Hero object
- **Left-/right-arrow keys:** Change the front direction of the Brain object when it is under user control
- **Up-/down-arrow keys:** Increase/decrease the speed of the Brain object
- **H key:** Switches the Brain object to be under user arrow keys control
- **J key:** Switches the Brain object to always point at and move toward the current Hero object position
- **K key:** Switches the Brain object to turn and move gradually toward the current Hero object position

The goals of the project are as follows:

- To understand the implementation of the bounding box class
- To experience working with the bounding box of a GameObject instance
- To compute and work with the bounds of a Camera WC window
- To program with object collisions and object and camera WC window collisions

You can find the same external resource files as in the previous project in the assets folder.

Define a Bounding Box Class

Define a BoundingBox class to represent the bounds of a rectangular area:

1. Create a new file in the `src/engine` folder; name it `bounding_box.js`. First, define an enumerated data type with values that identify the colliding sides of a bounding box.

```
const eBoundCollideStatus = Object.freeze({
  eCollideLeft: 1,
  eCollideRight: 2,
  eCollideTop: 4,
```

CHAPTER 6 DEFINING BEHAVIORS AND DETECTING COLLISIONS

```
715     eCollideBottom: 8,  
716     eInside: 16,  
717     eOutside: 0  
718 });


```

719 Notice that each enumerated value has only one nonzero bit. This allows the
720 enumerated values to be combined with the bitwise-or operator to represent a
721 multisided collision. For example, if an object collides with both the top and left sides of
722 a bounding box, the collision status will be `eCollideLeft | eCollideTop = 1 | 4 = 5`.

- 723 2. Now, define the `BoundingBox` class and the constructor with
724 instance variables to represent a bound, as illustrated in Figure 6-11.
725 Notice that the `eBoundCollideStatus` must also be exported such
726 that the rest of the engine, including the client, can also have
727 access.

```
728 class BoundingBox {  
729     constructor(centerPos, w, h) {  
730         this.mLL = vec2.fromValues(0, 0);  
731         this.setBounds(centerPos, w, h);  
732     }  
733     ... implementation to follow ...  
734 }  
735 export {eBoundCollideStatus}  
736 export default BoundingBox;
```

- 737 3. The `setBounds()` function computes and sets the instance
738 variables of the bounding box:

```
739 setBounds(centerPos, w, h) {  
740     this.mWidth = w;  
741     this.mHeight = h;  
742     this.mLL[0] = centerPos[0] - (w / 2);  
743     this.mLL[1] = centerPos[1] - (h / 2);  
744 }
```

4. Define a function to determine whether a given position, (x, y) , is within the bounds of the box:

```
containsPoint(x, y) {
    return ((x > this.minX()) && (x < this maxX()) &&
        (y > this.minY()) && (y < this.maxY()));
}
```

5. Define a function to determine whether a given bound intersects with the current one:

```
intersectsBound(otherBound) {
    return ((this.minX() < otherBound.maxX()) &&
        (this.maxX() > otherBound.minX()) &&
        (this.minY() < otherBound.maxY()) &&
        (this.maxY() > otherBound.minY()));
}
```

6. Define a function to compute the intersection status between a given bound and the current one:

```
boundCollideStatus(otherBound) {
    let status = eBoundCollideStatus.eOutside;

    if (this.intersectsBound(otherBound)) {
        if (otherBound minX() < this.minX()) {
            status |= eBoundCollideStatus.eCollideLeft;
        }
        if (otherBound maxX() > this.maxX()) {
            status |= eBoundCollideStatus.eCollideRight;
        }
        if (otherBound minY() < this.minY()) {
            status |= eBoundCollideStatus.eCollideBottom;
        }
        if (otherBound maxY() > this.maxY()) {
            status |= eBoundCollideStatus.eCollideTop;
        }
    }
}
```

```

745     // if the bounds intersects and yet none of the sides overlaps
746     // otherBound is completely inside thisBound
747     if (status === eBoundCollideStatus.eOutside) {
748         status = eBoundCollideStatus.eInside;
749     }
750 }
751 return status;
752 }
```

Notice the subtle yet important difference between the `intersectsBound()` and `boundCollideStatus()` functions where the former is capable of returning only a true or false condition while the latter function encodes the colliding sides in the returned `status`.

7. Implement the functions that return the X/Y values to the min and max bounds of the bounding box:

```

759 minX() { return this.mLL[0]; }
760 maxX() { return this.mLL[0] + this.mWidth; }
761 minY() { return this.mLL[1]; }
762 maxY() { return this.mLL[1] + this.mHeight; }
```

Lastly, remember to update the engine access file, `index.js`, to forward the newly defined functionality to the client.

765 Use the BoundingBox in the Engine

766 The newly defined functionality will be used to detect collisions between objects and
767 between objects and the WC bounds. In order to accomplish this, the `GameObject` and
768 `Camera` classes must be modified.

769 1. Edit `game_object.js` to import the newly defined functionality
770 and modify the `GameObject` class; implement the `getBBox()`
771 function to return the bounding box of the unrotated `Renderable`
772 object:

```

773 import BoundingBox from "../bounding_box.js";
774 class GameObject {
775     ... identical to previous code ...
```

```

getBBox() { 776
    let xform = this.getXform(); 777
    let b = new BoundingBox( 778
        xform.getPosition(), 779
        xform.getWidth(), 780
        xform.getHeight()); 781
    return b; 782
} 783
... identical to previous code ... 784
} 785

```

2. Edit camera.js to import from bounding box, and modify the Camera class to compute the collision status between the bounds of a Transform object (typically defined in a Renderable object) and that of the WC window:

```

import BoundingBox from "./bounding_box.js"; 790
class Camera { 791
    ... identical to previous code ... 792
    collideWCBound(aXform, zone) { 793
        let bbox = new BoundingBox( 794
            aXform.getPosition(), 795
            aXform.getWidth(), 796
            aXform.getHeight()); 797
        let w = zone * this.getWCWidth(); 798
        let h = zone * this.getWCHeight(); 799
        let cameraBound = new BoundingBox(this.getWCCenter(), w, h); 800
        return cameraBound.boundCollideStatus(bbox); 801
    } 802
} 803

```

Notice that the zone parameter defines the relative size of WC that should be used in the collision computation. For example, a zone value of 0.8 would mean computing for intersection status based on 80 percent of the current WC window size. Figure 6-13 shows how the camera collides with an object.

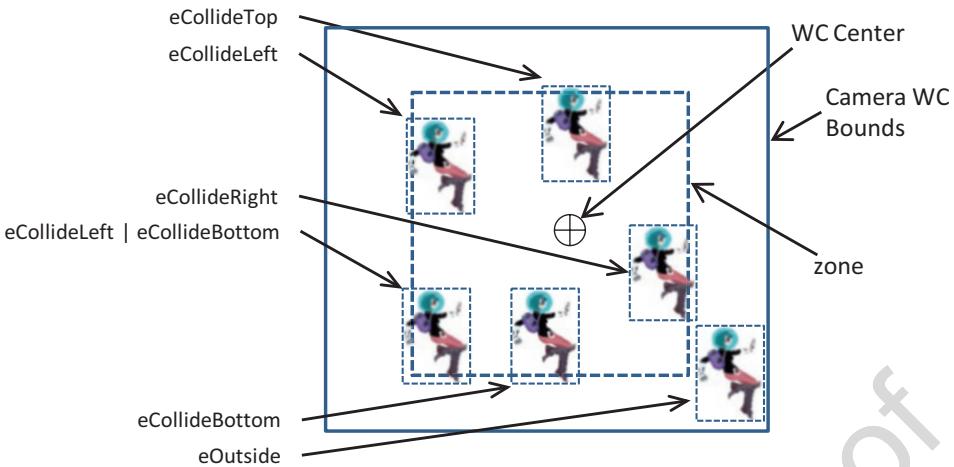


Figure 6-13. Camera WC bounds colliding with the bounds defining a Transform object

808 Test Bounding Boxes with MyGame

809 The goal of this test case is to verify the correctness of the bounding box implementation
 810 in detecting object-object and object-camera intersections. Once again, with the
 811 exception of the update() function, the majority of the code in the my_game.js file
 812 is similar to the previous projects and is not repeated here. The update() function is
 813 modified from the previous project to test for bounding box intersections.

```
814 update() {
815     ... identical to previous code ...
816     switch (this.mMode) {
817         case 'H':
818             this.mBrain.update(); // player steers with arrow keys
819             break;
820         case 'K':
821             rate = 0.02; // graduate rate
822             // no break here on purpose
823         case 'J':
824             // stop the brain when it touches hero bound
825             if (!hBbox.intersectsBound(bBbox)) {
826                 this.mBrain.rotateObjPointTo(
```

```

        this.mHero.getXform().getPosition(), rate);
    // the default GameObject: only move forward
    engine.GameObject.prototype.update.call(this.mBrain);
}
break;
}

// Check for hero going outside 80% of the WC Window bound
let status = this.mCamera.collideWCBound(this.mHero.getXform(), 0.8);
... identical to previous code ...

this.mMsg.setText(msg + this.mMode + " [Hero bound=" + status + "]");
}

```

In the switch statement's J and K cases, the modification tests for bounding box collision between the Brain and Hero objects before invoking Brain. `rotateObjPointTo()` and `update()` to cause the chasing behavior. In this way, the Brain object will stop moving as soon as it touches the bound of the Hero object. In addition, the collision results between the Hero object and 80 percent of the camera WC window are computed and displayed.

Observation

You can now run the project and observe that the Brain object, when in autonomous mode (J or K keys), stops moving as soon as it touches the Hero object. When you move the Hero object around, observe the Hero_bound output message begins to echo WC window collisions before the Hero object actually touches the WC window bounds. This is a result of the 0.8, or 80 percent, parameter passed to the `mCamera.collideWCBound()` function, configuring the collision computation to 80 percent of the current WC window size. When the Hero object is completely within 80 percent of the WC window bounds, the output Hero_bound value is 16 or the value of `eboundcollideStatus.eInside`. Try moving the Hero object to touch the top 20 percent of the window bound, and observe the Hero_bound value of 4 or the value of `eboundcollideStatus.eCollideTop`. Now move the Hero object toward the top-left corner of the window, and observe the Hero_bound value of 5 or `eboundcollideStatus.eCollideTop | eboundcollideStatus.eCollideLeft`. In this way, the collision status is a bitwise-or result of all the colliding bounds.

827 Per-Pixel Collisions

828 In the previous example, you saw the results of bounding box collision approximation.
 829 Namely, the Brain object's motion stops as soon as its bounds overlap that of the Hero
 830 object. This is much improved over the original situation where the Brain object never
 831 stops moving. However, as illustrated in Figure 6-14, there are two serious limitations to
 832 the bounding box-based collisions.

- 833 1. The BoundingBox object introduced in the previous example does
 834 not account for rotation. This is a well-known limitation for AABB:
 835 although the approach is computationally efficient, it does not
 836 support rotated objects.
- 837 2. The two objects do not actually collide. The fact that the bounds
 838 of two objects overlap does not automatically equate to the two
 839 objects colliding.

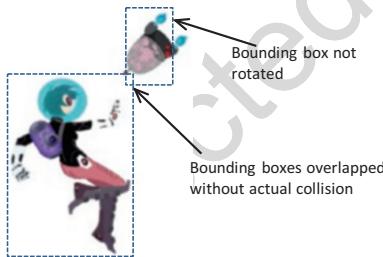


Figure 6-14. Limitation with bounding box-based collision

840 In this project, you will implement per-pixel-accurate collision detection, pixel-
 841 accurate collision detection, or per-pixel collision detection, to detect the overlapping
 842 of nontransparent pixels of two colliding objects. However, keep in mind that this is
 843 *not* an end-all solution. While the per-pixel collision detection is precise, the trade-off
 844 is potential performance costs. As an image becomes larger and more complex, it also
 845 has more pixels that need to be checked for collisions. This is in contrast to the constant
 846 computation cost required for bounding box collision detection.

The Per-Pixel Collisions Project

847

This project demonstrates how to detect collision between a large textured object, the Collector minion and a small textured object, the Portal minion. Both of the textures contain transparent and nontransparent areas. A collision occurs only when the nontransparent pixels overlap. In this project, when a collision occurs, a yellow DyePack appears at the collision point. You can see an example of this project running in Figure 6-15. The source code to this project is defined in the chapter6/6.4.per_pixel_collisions folder.

848

849

850

851

852

853

854

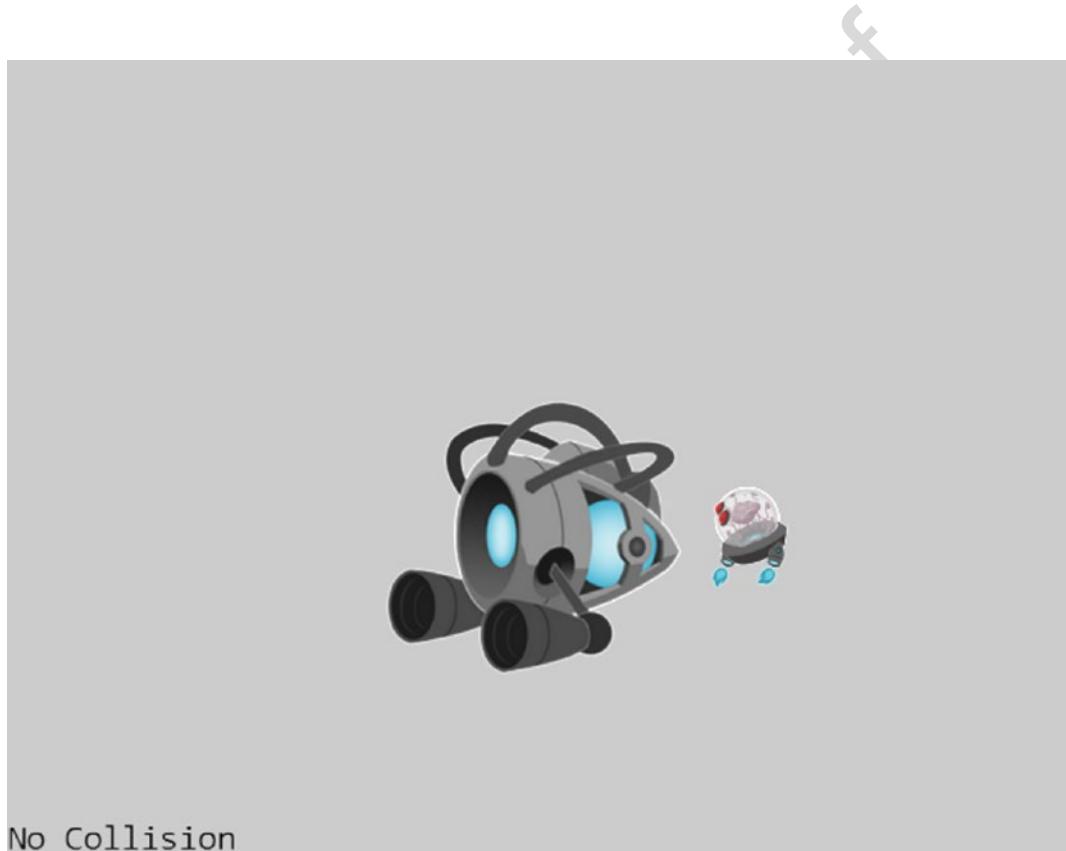


Figure 6-15. Running the Per-Pixel Collisions project

The controls of the project are as follows:

855

- **Arrow keys:** Move the small textured object, the Portal minion
- **WASD keys:** Move the large textured object, the Collector minion

856

857

858 The goals of the project are as follows:

- 859 • To demonstrate how to detect nontransparent pixel overlap
860 • To understand the pros and cons of using per-pixel-accurate collision
861 detection
-

862 **Note** A “transparent” pixel is one you can see through completely and, in
863 the case of this engine, has an alpha value of 0. A “nontransparent” pixel has
864 a greater than 0 alpha value, or the pixel does not completely block what is
865 behind it; it may or may not occlude. An “opaque” pixel occludes what is behind
866 it, is “nontransparent,” and has an alpha value of 1. For example, notice that
867 you can “see through” the top region of the Portal object. These pixels are
868 nontransparent but not opaque and should cause a collision when an overlap
869 occurs based on the parameters defined by the project.

870 You can find the following external resources in the assets folder: the fonts folder
871 that contains the default system fonts, `minion_collector.png`, `minion_portal.png`, and
872 `minion_sprite.png`. Note that `minion_collector.png` is a large, 1024x1024 image, while
873 `minion_portal.png` is a small, 64x64 image; `minion_sprite.png` defines the DyePack
874 sprite element.

875 Overview of Per-Pixel Collision Algorithm

876 Before moving forward, it is important to identify the requirements for detecting
877 a collision between two textured objects. Foremost is that the texture itself needs
878 to contain an area of transparency in order for this type of collision detection to
879 provide any increase in accuracy. Without transparency in the texture, you can and
880 should use a simple bounding box collision detection. If one or both of the textures
881 contain transparent areas, then you’ll need to handle two cases of collision. The
882 first case is to check whether the bounds of the two objects collide. You can see this
883 reflected in Figure 6-16. Notice how the bounds of the objects overlap, yet none of the
884 nontransparent colored pixels are touching.

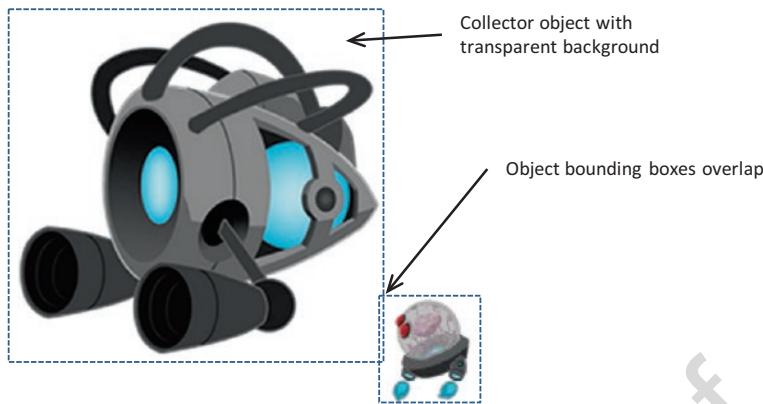


Figure 6-16. Overlapping bounding boxes without actual collision

The next case is to check whether the nontransparent pixels of the textures overlap.
Take a look at Figure 6-17. Nontransparent pixels from the textures of the Collector and
Portal objects are clearly in contact with one another.

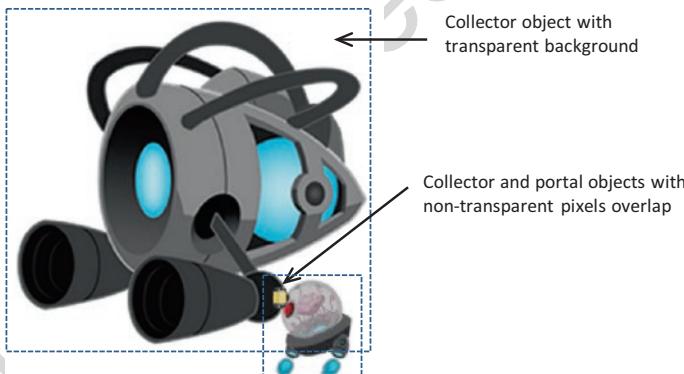
885
886
887

Figure 6-17. Pixel collision occurring between the large texture and the small texture

Now that the problem is clearly defined, here is the logic or pseudocode for per-pixel-
accurate collision detection:

888
889

Given two images, Image-A and Image-B

890

If the bounds of the two collide then

891

For each Pixel-A in Image-A

892

If Pixel-A is not completely transparent

893

```

894     pixelCameraSpace = Pixel-A position in camera space
895     Transform pixelCameraSpace to Image-B space
896     Read Pixel-B from Image-B
897     If Pixel-B is not completely transparent then
898         A collision has occurred

```

899 The per-pixel transformation to Image-B space from pixelCameraSpace is required
900 because collision checking must be carried out within the same coordinate space.

901 Notice that in the algorithm Image-A and Image-B are exchangeable. That is, when
902 testing for collision between two images, it does not matter which image is Image-A
903 or Image-B. The collision result will be the same. Either the two images do overlap,
904 or they do not. Additionally, pay attention to the runtime of this algorithm. Each pixel
905 within Image-A must be processed; thus, the runtime is $O(N)$, where N is the number of
906 pixels in Image-A or Image-A's resolution. For this reason, for performance reason, it is
907 important to choose the smaller of the two images (the Portal minion in this case) as
908 Image-A.

909 At this point, you can probably see why the performance of pixel-accurate collision
910 detection is concerning. Checking for these collisions during every update with many
911 high-resolution textures can quickly bog down performance. You are now ready to
912 examine the implementation of per-pixel-accurate collision.

913 **Modify Texture to Load a Texture as an Array of Colors**

914 Recall that the Texture component reads image files from the server file system, loads
915 the images to the GPU memory, and processes the images into WebGL textures. In
916 this way, texture images are stored on the GPU and are not accessible by the game
917 engine which is running on the CPU. To support per-pixel collision detection, the
918 color information must be retrieved from the GPU and stored in the CPU. The Texture
919 component can be modified to support this requirement.

- 920 1. In the texture.js file, expand the TextureInfo object to include
921 a new variable for storing the color array of a file texture:

```

922 class TextureInfo {
923     constructor(w, h, id) {
924         this.mWidth = w;
925         this.mHeight = h;

```

```

this.mGLTexID = id;                                926
this.mColorArray = null;                         927
}
}                                                       928
}                                                       929
2. Define and export a function to retrieve the color array from the 930
   GPU memory:                                         931

function getColorArray(textureName) {                932
  let gl = glSys.get();                            933
  let texInfo = get(textureName);                  934
  if (texInfo.mColorArray === null) {               935
    // create framebuffer bind to texture and read the color content
    let fb = gl.createFramebuffer();                 936
    gl.bindFramebuffer(gl.FRAMEBUFFER, fb);         937
    gl.framebufferTexture2D(gl.FRAMEBUFFER,          938
      gl.COLOR_ATTACHMENT0,                          939
      gl.TEXTURE_2D, texInfo.mGLTexID, 0);           940
    if (gl.checkFramebufferStatus(gl.FRAMEBUFFER) === 941
        gl.FRAMEBUFFER_COMPLETE) {
      let pixels = new Uint8Array(                      942
        texInfo.mWidth * texInfo.mHeight * 4);        943
      gl.readPixels(0, 0, texInfo.mWidth, texInfo.mHeight, 944
        gl.RGBA, gl.UNSIGNED_BYTE, pixels);
      texInfo.mColorArray = pixels;                   945
    } else {                                         946
      throw new Error("...");                        947
      return null;                                  948
    }
    gl.bindFramebuffer(gl.FRAMEBUFFER, null);        949
    gl.deleteFramebuffer(fb);                        950
  }
  return texInfo.mColorArray;
}

```

```
958 export {has, get, load, unload,  
959     TextureInfo,  
960     activate, deactivate,  
961     getColorArray  
962 }
```

963 The `getColorArray()` function creates a WebGL FRAMEBUFFER, fills the buffer with
964 the desired texture, and retrieves the buffer content into the CPU memory referenced by
965 `texInfo.mColorArray`.

966 **Modify TextureRenderable to Support Per-Pixel Collision**

967 The `TextureRenderable` is the most appropriate class for implementing the per-pixel
968 collision functionality. This is because `TextureRenderable` is the base class for all classes
969 that render textures. Implementation in this base class means all subclasses can inherit
970 the functionality with minimal additional changes.

971 As the functionality of the `TextureRenderable` class increases, so will the complexity
972 and size of the implementation source code. For readability and expandability, it is
973 important to maintain the size of source code files. An effective approach is to separate
974 the source code of a class into multiple files according to their functionality.

975 **Organize the Source Code**

976 In the following steps, the `TextureRenderable` class will be separated into three source
977 code files: `texture_renderable_main.js` for implementing the basic functionality from
978 previous projects, `texture_renderable_pixel_collision.js` for implementing the
979 newly introduced per-pixel-accurate collision, and `texture_renderable.js` for serving
980 as the class access point.

- 981 1. Rename `texture_renderable.js` to `texture_renderable_`
982 `main.js`. This file defines the basic functionality of the
983 `TextureRenderable` class.
- 984 2. Create a new file in `src/engine/renderables` and name it
985 `texture_renderable_pixel_collision.js`. This file will be used
986 to extend the `TextureRenderable` class functionality in supporting
987 per-pixel-accurate collision. Add in the following code to import

from the Texture module and the basic TextureRenderable class, 988
 and reexport the TextureRenderable class. For now, this file does 989
 not serve any purpose; you will add in the appropriate extending 990
 functions in the following subsection. 991

```
"use strict";
import TextureRenderable from "./texture_renderable_main.js";
import * as texture from "../resources/texture.js";
... implementation to follow ...
export default TextureRenderable;
```

3. Create a new texture_renderable.js file to serve as the 997
 TextureRenderable access point by adding the following code: 998

```
"use strict";
import TextureRenderable from "./ texture_renderable_pixel_collision.js";
export default TextureRenderable;
```

With this structure, the texture_renderable_main.js file implements all the basic 1002 functionality and exports to texture_renderable_pixel_collision.js, which appends 1003 additional functionality to the TextureRenderable class. Finally, texture_renderable.js 1004 imports the extended functions from texture_renderable_pixel_collision.js. The 1005 users of the TextureRenderable class can simply import from texture_renderable.js 1006 and will have access to all of the defined functionality. 1007

In this way, from the perspective of the game developer, texture_renderable.js 1008 serves as the access point to the TextureRenderable class and hides the details of the 1009 implementation source code structure. At the same time, from the perspective of you as 1010 the engine developer, complex implementations are separated into source code files with 1011 names indicating the content achieving readability of each individual file. 1012

Define Access to the Texture Color Array

Recall that you began this project by first editing the Texture module to retrieve, 1014
 from the GPU to the CPU, the color array that represents a texture. You must now edit 1015
 TextureRenderable to gain access to this color array. 1016

CHAPTER 6 DEFINING BEHAVIORS AND DETECTING COLLISIONS

- 1017 1. Edit the `texture_renderable_main.js` file, and modify the
1018 constructor to add instance variables to hold texture information,
1019 including a reference to the retrieved color array, for supporting
1020 per-pixel collision detection and for later subclass overrides:

```
1021 class TextureRenderable extends Renderable {  
1022     constructor(myTexture) {  
1023         super();  
1024         // Alpha of 0: switch off tinting of texture  
1025         super.setColor([1, 1, 1, 0]);  
1026         super._setShader(shaderResources.getTextureShader());  
1027         this.mTexture = null;  
1028         // these two instance variables are to cache texture information  
1029         // for supporting per-pixel accurate collision  
1030         this.mTextureInfo = null;  
1031         this.mColorArray = null;  
1032         // defined for subclass to override  
1033         this.mElmWidthPixels = 0;  
1034         this.mElmHeightPixels = 0;  
1035         this.mElmLeftIndex = 0;  
1036         this.mElmBottomIndex = 0;  
1037         // texture for this object, cannot be a "null"  
1038         this.setTexture(myTexture);  
1039     }  
1040 }
```

- 1040 2. Modify the `setTexture()` function to initialize the instance
1041 variables accordingly:

```
1042     setTexture(newTexture) {  
1043         this.mTexture = newTexture;  
1044         // these two instance variables are to cache texture information  
1045         // for supporting per-pixel accurate collision  
1046         this.mTextureInfo = texture.get(newTexture);  
1047         this.mColorArray = null;  
1048         // defined for one sprite element for subclass to override  
1049         // For texture_renderable, one sprite element is the entire texture
```

```

this.mElmWidthPixels = this.mTextureInfo.mWidth;          1050
this.mElmHeightPixels = this.mTextureInfo.mHeight;        1051
this.mElmLeftIndex = 0;                                  1052
this.mElmBottomIndex = 0;                                1053
}

```

Note that by default, the `mColorArry` is initialized to `null`. For CPU memory optimization, the color array is fetched from the GPU only for textures that participate in per-pixel collision. The `mElmWidthPixels` and `mElmHeightPixels` variables are the width and height of the texture. These variables are defined for later subclass overrides such that the algorithm can support the collision of sprite elements.

Implement Per-Pixel Collision

You are now ready to implement the per-pixel collision algorithm in the newly created `texture_renderable_pixel_collision.js` file.

1. Edit the `texture_renderable_pixel_collision.js` file, and define a new function for the `TextureRenderable` class to set the `mColorArray`:

```

TextureRenderable.prototype.setColorArray = function() {
  if (this.mColorArray === null) {
    this.mColorArray = texture.getColorArray(this.mTexture);
  }
}

```

Note JavaScript classes are implemented based on prototype chains. After class construction, instance methods can be accessed and defined via the prototype of the class or `aClass.prototype.method`. For more information on JavaScript classes and prototypes, please refer to https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain.

CHAPTER 6 DEFINING BEHAVIORS AND DETECTING COLLISIONS

- 1076 2. Define a new function to return the alpha value, or the
1077 transparency, of any given pixel (x, y):

```
1078 TextureRenderable.prototype._pixelAlphaValue = function(x, y) {  
1079     x = x * 4;  
1080     y = y * 4;  
1081     return this.mColorArray[(y * this.mTextureInfo.mWidth) + x + 3];  
1082 }
```

1083 Notice that mColorArray is a 1D array where colors of pixels are stored as four floats
1084 and organized by rows.

- 1085 3. Define a function to compute the WC position (returnWCPos) of a
1086 given pixel (i, j):

```
1087 TextureRenderable.prototype._indexToWCPosition =  
1088 function(returnWCPos, i, j) {  
1089     let x = i * this.mXform.getWidth() / this.mElmWidthPixels;  
1090     let y = j * this.mXform.getHeight() / this.mElmHeightPixels;  
1091     returnWCPos[0] = this.mXform.getXPos() +  
1092                     (x - (this.mXform.getWidth() * 0.5));  
1093     returnWCPos[1] = this.mXform.getYPos() +  
1094                     (y - (this.mXform.getHeight() * 0.5));  
1095 }
```

- 1096 4. Now, implement the inverse of the previous function, and use
1097 a WC position (wcPos) to compute the texture pixel indices
1098 (returnIndex):

```
1099 TextureRenderable.prototype._wcPositionToIndex =  
1100 function(returnIndex, wcPos) {  
1101     // use wcPos to compute the corresponding returnIndex[0 and 1]  
1102     let delta = [];  
1103     vec2.sub(delta, wcPos, this.mXform.getPosition());  
1104     returnIndex[0] = this.mElmWidthPixels *  
1105                     (delta[0] / this.mXform.getWidth());  
1106     returnIndex[1] = this.mElmHeightPixels *  
1107                     (delta[1] / this.mXform.getHeight());
```

```

// recall that xForm.getPosition() returns center, yet           1108
// Texture origin is at lower-left corner!                     1109
returnIndex[0] += this.mElmWidthPixels / 2;                  1110
returnIndex[1] += this.mElmHeightPixels / 2;                 1111

returnIndex[0] = Math.floor(returnIndex[0]);                  1112
returnIndex[1] = Math.floor(returnIndex[1]);                 1113
}

```

5. Now it is possible to implement the outlined per-pixel collision
algorithm:

```

TextureRenderable.prototype.pixelTouches = function(other, wcTouchPos) {    1117
    let pixelTouch = false;                                         1118
    let xIndex = 0, yIndex;                                         1119
    let otherIndex = [0, 0];                                         1120

    while ((!pixelTouch) && (xIndex < this.mElmWidthPixels)) {        1121
        yIndex = 0;
        while ((!pixelTouch) && (yIndex < this.mElmHeightPixels)) {    1122
            if (this._pixelAlphaValue(xIndex, yIndex) > 0) {
                this._indexToWCPosition(wcTouchPos, xIndex, yIndex);
                other._wcPositionToIndex(otherIndex, wcTouchPos);
                if ((otherIndex[0] >= 0) &&
                    (otherIndex[0] < other.mElmWidthPixels) &&
                    (otherIndex[1] >= 0) &&
                    (otherIndex[1] < other.mElmHeightPixels)) {
                    pixelTouch = other._pixelAlphaValue(
                        otherIndex[0], otherIndex[1]) > 0;
                }
            }
            yIndex++;
        }
        xIndex++;
    }
    return pixelTouch;
}

```

1141 The parameter `other` is a reference to the other `TextureRenderable` object that is
 1142 being tested for collision. If pixels do overlap between the objects, the returned value
 1143 of `wcTouchPos` is the first detected colliding position in the WC space. Notice that the
 1144 nested loops terminate as soon as one-pixel overlap is detected or when `pixelTouch`
 1145 becomes true. This is an important feature for efficiency concerns. However, this also
 1146 means that the returned `wcTouchPos` is simply one of the many potentially colliding
 1147 points.

1148 **Support Per-Pixel Collision in GameObject**

1149 Edit the `game_object.js` file to add the `pixelTouches()` function to the `GameObject`
 1150 class:

```
1151 pixelTouches(otherObj, wcTouchPos) {
  1152     // only continue if both objects have getColorArray defined
  1153     // if defined, should have other texture intersection support!
  1154     let pixelTouch = false;
  1155     let myRen = this.getRenderable();
  1156     let otherRen = otherObj.getRenderable();

  1157     if ((typeof myRen.pixelTouches === "function") &&
  1158         (typeof otherRen.pixelTouches === "function")) {
  1159         let otherBbox = otherObj.getBBox();
  1160         if (otherBbox.intersectsBound(this.getBBox())) {
  1161             myRen.setColorArray();
  1162             otherRen.setColorArray();
  1163             pixelTouch = myRen.pixelTouches(otherRen, wcTouchPos);
  1164         }
  1165         return pixelTouch;
  1166     }
  1167 }
```

1168 This function checks to ensure that the objects are colliding and delegates the actual
 1169 per-pixel collision to the `TextureRenderable` objects. Notice the `intersectsBound()`
 1170 function for a bounding box intersection check before invoking the potentially expensive
 1171 `TextureRenderable.pixelTouches()` function.

Test the Per-Pixel Collision in MyGame

As illustrated in Figure 6-15, the testing of per-pixel collision is rather straightforward, involving three instances of `GameObject`: the large `Collector` minion, the small `Portal` minion, and the `DyePack`. The `Collector` and `Portal` minions are controlled by the arrow and WASD keys, respectively. The details of the implementation of `MyGame` are similar to the previous projects and are not shown.

The noteworthy code fragment is the collision testing in the `update()` function, as shown here:

```

update() {
    let msg = "No Collision";
    this.mCollector.update(engine.input.keys.W, engine.input.keys.S,
        engine.input.keys.A, engine.input.keys.D);
    this.mPortal.update(engine.input.keys.Up, engine.input.keys.Down,
        engine.input.keys.Left, engine.input.keys.Right);

    let h = [];
    // Portal's resolution is 1/16 x 1/16 that of Collector!
    // VERY EXPENSIVE!!
    // if (this.mCollector.pixelTouches(this.mPortal, h)) {
    if (this.mPortal.pixelTouches(this.mCollector, h)) {
        msg = "Collided!: (" + h[0].toPrecision(4) + " " +
            h[1].toPrecision(4) + ")";
        this.mDyePack.setVisibility(true);
        this.mDyePack.getXform().setXPos(h[0]);
        this.mDyePack.getXform().setYPos(h[1]);
    } else {
        this.mDyePack.setVisibility(false);
    }
    this.mMsg.setText(msg);
}

```

Observation

You can now test the collision accuracy by moving the two minions and intersecting them at different locations (e.g., top colliding with the bottom, left colliding with the right) or moving them such that there are large overlapping areas. Notice that it is rather difficult, if not impossible, to predict the actual reported intersection position (position of the DyePack). It is important to remember that the per-pixel collision function is mainly a function that returns true or false indicating whether there is a collision. You cannot rely on this function to compute the actual collision positions.

Lastly, try switching to calling the `Collector.pixelTouches()` function to detect collisions. Notice the less than real-time performance! In this case, the computation cost of the `Collector.pixelTouches()` function is $16 \times 16 = 256$ times that of the `Portal.pixelTouches()` function.

Generalized Per-Pixel Collisions

In the previous section, you saw the basic operations required to achieve per-pixel-accurate collision detection. However, as you may have noticed, the previous project applies only when the textures are aligned along the x/y axes. This means that your implementation does not support collisions between rotated objects.

This section explains how you can achieve per-pixel-accurate collision detection when objects are rotated. The fundamental concepts of this project are the same as in the previous project; however, this version involves working with vector decomposition, and a quick review can be helpful.

Vector Review: Components and Decomposition

Recall that two perpendicular directions can be used to decompose a vector into corresponding components. For example, Figure 6-18 contains two normalized vectors, or the component vectors, that can be used to decompose the vector $\vec{V} = (2, 3)$: the normalized component vectors $\hat{i} = (1, 0)$ and $\hat{j} = (0, 1)$ decompose the vector \vec{V} into the components $2\hat{i}$ and $3\hat{j}$.

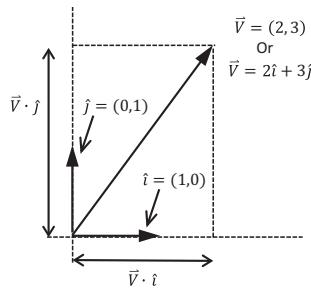


Figure 6-18. The decomposition of vector \vec{V}

In general, as illustrated in Figure 6-19, given the normalized perpendicular component vectors \hat{L} and \hat{M} and any vector \vec{V} , the following formulae are always true:

$$\vec{V} = (\vec{V} \cdot \hat{i}) \hat{i} + (\vec{V} \cdot \hat{j}) \hat{j}$$

$$\vec{V} = (\vec{V} \cdot \hat{L}) \hat{L} + (\vec{V} \cdot \hat{M}) \hat{M}$$

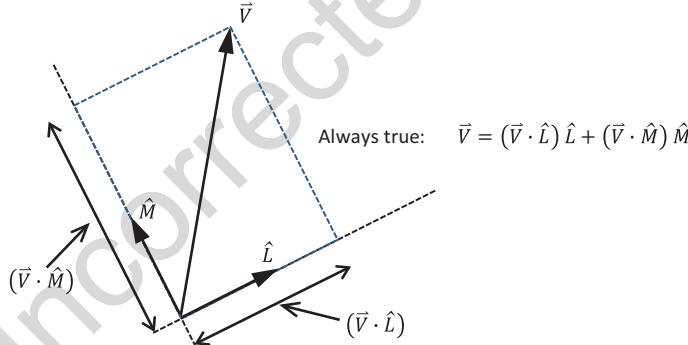


Figure 6-19. Decomposing a vector by two normalized component vectors

Vector decomposition is relevant to this project because of the rotated image axes. Without rotation, an image can be referenced by the familiar normalized perpendicular set of vectors along the default x axis (\hat{i}) and y axis (\hat{j}). You handled this case in the previous project. You can see an example of this in Figure 6-20.

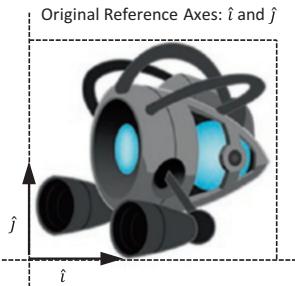


Figure 6-20. An axes-aligned texture

1201 However, after the image has been rotated, the reference vector set no longer resides
 1202 along the x/y axes. Therefore, the collision computation must take into account the
 1203 newly rotated axes \hat{L} and \hat{M} , as shown in Figure 6-21.

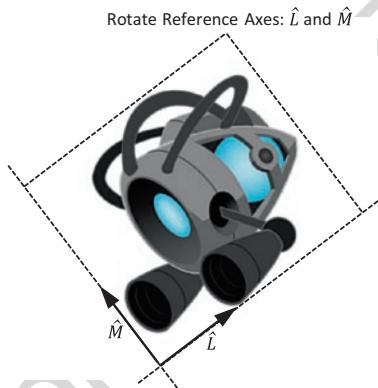
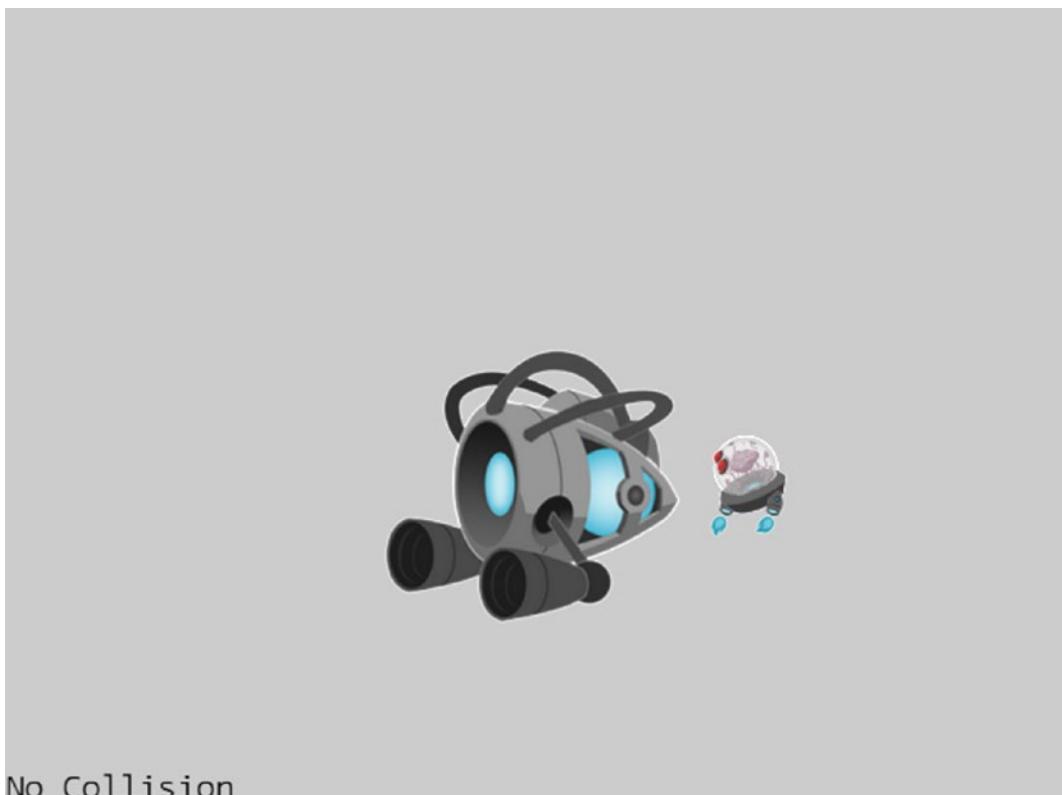


Figure 6-21. A rotated texture and its component vectors

1204 The General Pixel Collisions Project

1205 This project demonstrates how to detect a collision between two rotated
 1206 TextureRenderable objects with per-pixel accuracy. Similar to the previous project, a
 1207 yellow DyePack object (as a test confirmation) will be displayed at the detected colliding
 1208 position. You can see an example of this project running in Figure 6-22. The source code
 1209 to this project is defined in the chapter6/6.5.general_pixel_collisions folder.



this figure will be printed in b/w

Figure 6-22. Running the General Pixel Collisions project

The controls of the project are as follows:

- **Arrow keys:** Move the small textured object, the Portal minion 1210
- **P key:** Rotates the small textured object, the Portal minion 1211
- **WASD keys:** Move the large textured object, the Collector minion 1212
- **E key:** Rotates the large textured object, the Collector minion 1213

The goals of the project are as follows:

- To access pixels of a rotated image via vector decomposition 1214
- To support per-pixel-accurate collision detection between two rotated 1215
- textured objects 1216

1219 You can find the same external resource files as in the previous project in the assets
 1220 folder.

1221 Modify Pixel Collision to Support Rotation

1222 1. Edit the `texture_renderable_pixel_collision.js` file and
 1223 modify the `_indexToWCPosition()` function:

```
1224 TextureRenderable.prototype._indexToWCPosition =
1225   function (returnWCPos, i, j, xDir, yDir) {
1226     let x = i * this.mXform.getWidth() / this.mElmWidthPixels;
1227     let y = j * this.mXform.getHeight() / this.mElmHeightPixels;
1228     let xDisp = x - (this.mXform.getWidth() * 0.5);
1229     let yDisp = y - (this.mXform.getHeight() * 0.5);
1230     let xDirDisp = [];
1231     let yDirDisp = [];
1232
1233     vec2.scale(xDirDisp, xDir, xDisp);
1234     vec2.scale(yDirDisp, yDir, yDisp);
1235     vec2.add(returnWCPos, this.mXform.getPosition(), xDirDisp);
1236     vec2.add(returnWCPos, returnWCPos, yDirDisp);
1237 }
```

1237 In the listed code, `xDir` and `yDir` are the \hat{L} and \hat{M} normalized component vectors.
 1238 The variables `xDisp` and `yDisp` are the displacements to be offset along `xDir` and `yDir`,
 1239 respectively. The returned value of `returnWCPos` is a simple displacement from the
 1240 object's center position along the `xDirDisp` and `yDirDisp` vectors. Note that `xDirDisp`
 1241 and `yDirDisp` are the scaled `xDir` and `yDir` vectors.

1242 2. In a similar fashion, modify the `_wcPositionToIndex()` function
 1243 to support the rotated normalized vector components:

```
1244 TextureRenderable.prototype._wcPositionToIndex =
1245   function (returnIndex, wcPos, xDir, yDir) {
1246     // use wcPos to compute the corresponding returnIndex[0 and 1]
1247     let delta = [];
1248     vec2.sub(delta, wcPos, this.mXform.getPosition());
1249     let xDisp = vec2.dot(delta, xDir);
```

```

let yDisp = vec2.dot(delta, yDir);          1250
returnIndex[0] = this.mElmWidthPixels *      1251
    (xDisp / this.mXform.getWidth());        1252
returnIndex[1] = this.mElmHeightPixels *      1253
    (yDisp / this.mXform.getHeight());        1254

// recall that xForm.getPosition() returns center, yet
// Texture origin is at lower-left corner!          1255
returnIndex[0] += this.mElmWidthPixels / 2;      1256
returnIndex[1] += this.mElmHeightPixels / 2;       1257
returnIndex[0] = Math.floor(returnIndex[0]);        1258
returnIndex[1] = Math.floor(returnIndex[1]);        1259
}                                              1260
}                                              1261

```

3. The `pixelTouches()` function needs to be modified to compute
the rotated normalized component vectors:

```

TextureRenderable.prototype.pixelTouches = function (other, wcTouchPos) { 1262
    let pixelTouch = false;          1263
    let xIndex = 0, yIndex;          1264
    let otherIndex = [0, 0];          1265

    let xDir = [1, 0];              1266
    let yDir = [0, 1];              1267
    let otherXDir = [1, 0];          1268
    let otherYDir = [0, 1];          1269
    vec2.rotate(xDir, xDir, this.mXform.getRotationInRad());           1270
    vec2.rotate(yDir, yDir, this.mXform.getRotationInRad());           1271
    vec2.rotate(otherXDir, otherXDir, other.mXform.getRotationInRad()); 1272
    vec2.rotate(otherYDir, otherYDir, other.mXform.getRotationInRad()); 1273

    while ((!pixelTouch) && (xIndex < this.mElmWidthPixels)) {          1274
        yIndex = 0;                1275
        while ((!pixelTouch) && (yIndex < this.mElmHeightPixels)) {          1276
            if (this._pixelAlphaValue(xIndex, yIndex) > 0) {          1277
                this._indexToWCPosition(wcTouchPos,          1278
                    xIndex, yIndex, xDir, yDir);          1279
            }
        }
    }
}

```

```

1282         other._wcPositionToIndex(otherIndex, wcTouchPos,
1283                                     otherXDir, otherYDir);
1284         if ((otherIndex[0] >= 0) &&
1285             (otherIndex[0] < other.mElmWidthPixels) &&
1286             (otherIndex[1] >= 0) &&
1287             (otherIndex[1] < other.mElmHeightPixels)) {
1288             pixelTouch = other._pixelAlphaValue(
1289                         otherIndex[0], otherIndex[1]) > 0;
1290         }
1291     }
1292     yIndex++;
1293 }
1294 xIndex++;
1295 }
1296 return pixelTouch;
1297 }
```

1298 The variables `xDir` and `yDir` are the rotated normalized component vectors \hat{L} and
 1299 \hat{M} of this `TextureRenderable` object, while `otherXDir` and `otherYDir` are those of the
 1300 colliding object. These vectors are used as references for computing transformations
 1301 from texture index to WC and from WC to texture index.

1302 Modify GameObject to Support Rotation

1303 Recall that the `GameObject` class first tests for the bounding-box collision between
 1304 two objects before it actually invokes the much more expensive per-pixel collision
 1305 computation. As illustrated in Figure 6-14, the `BoundingBox` object does not support
 1306 object rotation correctly, and the following code remedies this shortcoming:

```

1307 pixelTouches(otherObj, wcTouchPos) {
1308     // only continue if both objects have getColorArray defined
1309     // if defined, should have other texture intersection support!
1310     let pixelTouch = false;
1311     let myRen = this.getRenderable();
1312     let otherRen = otherObj.getRenderable();
1313     if ((typeof myRen.pixelTouches === "function") &&
```

```

(typeof otherRen.pixelTouches === "function")) {
    1314
if ((myRen.getXform().getRotationInRad() === 0) &&
    1315
    (otherRen.getXform().getRotationInRad() === 0)) {
    1316
    // no rotation, we can use bbox ...
    1317
    let otherBbox = otherObj.getBBox();
    1318
    if (otherBbox.intersectsBound(this.getBBox())) {
    1319
        myRen.setColorArray();
    1320
        otherRen.setColorArray();
    1321
        pixelTouch = myRen.pixelTouches(otherRen, wcTouchPos);
    1322
    }
    1323
} else {
    1324
    // One or both are rotated, compute an encompassing circle
    1325
    // by using the hypotenuse as radius
    1326
    let mySize = myRen.getXform().getSize();
    1327
    let otherSize = otherRen.getXform().getSize();
    1328
    let myR = Math.sqrt(0.5*mySize[0]*0.5*mySize[0] +
    1329
                        0.5*mySize[1]*0.5*mySize[1]);
    1330
    let otherR = Math.sqrt(0.5*otherSize[0]*0.5*otherSize[0] +
    1331
                        0.5*otherSize[1]*0.5*otherSize[1]);
    1332
    let d = [];
    1333
    vec2.sub(d, myRen.getXform().getPosition(),
    1334
              otherRen.getXform().getPosition());
    1335
    if (vec2.length(d) < (myR + otherR)) {
    1336
        myRen.setColorArray();
    1337
        otherRen.setColorArray();
    1338
        pixelTouch = myRen.pixelTouches(otherRen, wcTouchPos);
    1339
    }
    1340
}
    1341
}
    1342
return pixelTouch;
}
    1343
}
    1344

```

The listed code shows that if either of the colliding objects is rotated, then two encompassing circles are used to determine whether the objects are sufficiently close for the expensive per-pixel collision computation. The two circles are defined with radii equal to the hypotenuse of the x/y size of the corresponding TextureRenderable objects.

1349 The per-pixel collision detection is invoked only if the distance between these two circles
1350 is less than the sum of the radii.

1351 **Test Generalized Per-Pixel Collision**

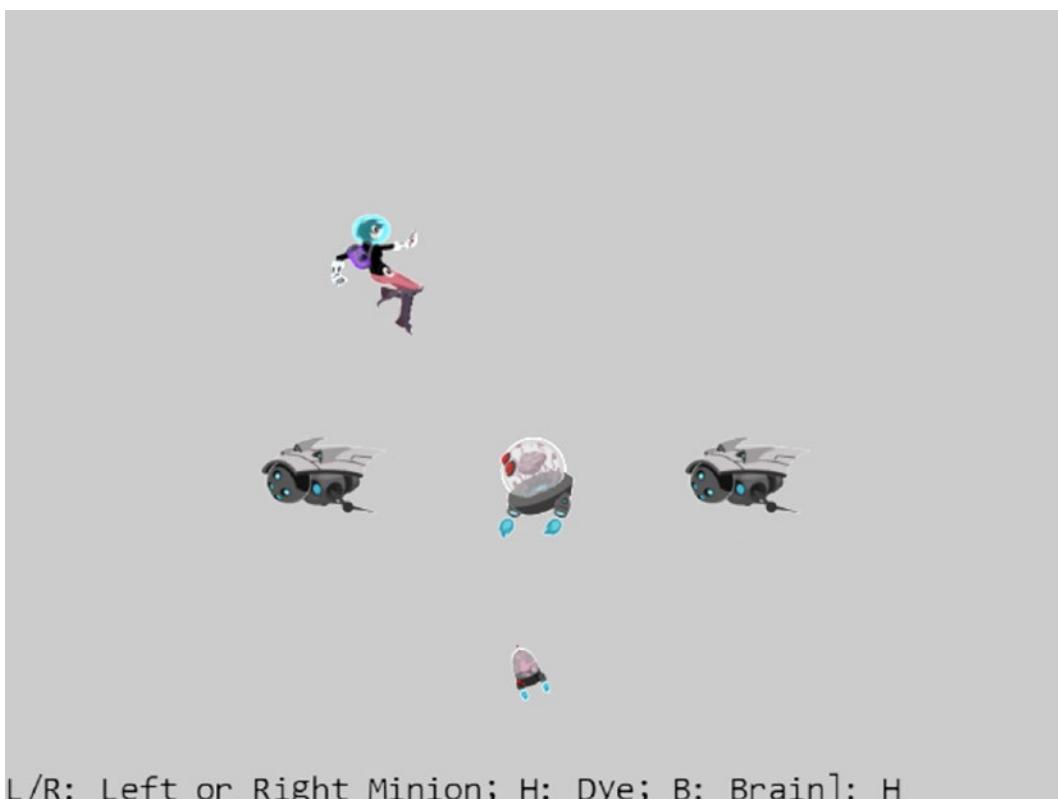
1352 The code for testing the rotated TextureRenderable objects is essentially identical to that
1353 from the previous project, with the exception of the two added controls for rotations. The
1354 details of the implementation are not shown. You can now run the project, rotate the two
1355 objects, and observe the accurate collision results.

1356 **Per-Pixel Collisions for Sprites**

1357 The previous project implicitly assumes that the Renderable object is covered by the
1358 entire texture map. This assumption means that the per-pixel collision implementation
1359 does not support sprite or animated sprite objects. In this section, you will remedy this
1360 deficiency.

1361 **The Sprite Pixel Collisions Project**

1362 This project demonstrates how to move an animated sprite object around the screen
1363 and perform per-pixel collision detection with other objects. The project tests for
1364 the correctness between collisions of TextureRenderable, SpriteRenderable, and
1365 SpriteAnimateRenderable objects. You can see an example of this project running in
1366 Figure 6-23. The source code to this project is defined in the chapter6/6.6.sprite_
1367 pixel_collisions folder.



L/R: Left or Right Minion; H: Dye; B: Brain]: H

Figure 6-23. Running the Sprite Pixel Collisions project

The controls of the project are as follows:

- **Arrow and P keys:** Move and rotate the Portal minion
- **WASD keys:** Move the Hero
- **L, R, H, B keys:** Select the target for colliding with the Portal minion

The goal of the project is as follows:

- To generalize the per-pixel collision implementation for sprite and animated sprite objects

You can find the following external resource files in the assets folder: the fonts folder that contains the default system fonts, `minion_sprite.png`, and `minion_portal.png`.

1377 Implement Per-Pixel Collision for SpriteRenderable

1378 Edit `sprite_renderable.js` to implement the per-pixel-specific support for
 1379 `SpriteRenderable` objects:

- 1380 1. Modify the `SpriteRenderable` constructor to call the `_
 1381 setTexInfo()` function to initialize per-pixel collision parameters;
 1382 this function is defined in the next step:

```
1383 constructor(myTexture) {
1384     super(myTexture);
1385     super._setShader(shaderResources.getSpriteShader());
1386     // sprite coordinate
1387     // bounds of texture coordinate (0 is left, 1 is right)
1388     this.mElmLeft = 0.0;
1389     this.mElmRight = 1.0;
1390     this.mElmTop = 1.0;    // 1 is top and 0 is bottom of image
1391     this.mElmBottom = 0.0; // 
1392     // sets info to support per-pixel collision
1393     this._setTexInfo();
1394 }
```

- 1395 2. Define the `_setTexInfo()` function to override instance variables
 1396 defined in the `TextureRenderable` superclass. Instead of the
 1397 entire texture image, the instance variables now identify the
 1398 currently active sprite element.

```
1399 _setTexInfo() {
1400     let imageW = this.mTextureInfo.mWidth;
1401     let imageH = this.mTextureInfo.mHeight;
1402     this.mElmLeftIndex = this.mElmLeft * imageW;
1403     this.mElmBottomIndex = this.mElmBottom * imageH;
1404     this.mElmWidthPixels = ((this.mElmRight - this.mElmLeft)*imageW)+1;
1405     this.mElmHeightPixels = ((this.mElmTop - this.mElmBottom)*imageH)+1;
1406 }
```

Notice that instead of the dimension of the entire texture map, `mElmWidthPixel` and `mElmHeightPixel` now contain pixel values that correspond to the dimension of a single sprite element in the sprite sheet.

3. Remember to call the `_setTexInfo()` function when the current sprite element is updated in the `setElementUVCoordinate()` and `setElementPixelPositions()` functions:

```
setElementUVCoordinate(left, right, bottom, top) {  
    this.mElmLeft = left;  
    this.mElmRight = right;  
    this.mElmBottom = bottom;  
    this.mElmTop = top;  
    this._setTexInfo();  
}  
  
setElementPixelPositions(left, right, bottom, top) {  
    // entire image width, height  
    let imageW = this.mTextureInfo.mWidth;  
    let imageH = this.mTextureInfo.mHeight;  
  
    this.mElmLeft = left / imageW;  
    this.mElmRight = right / imageW;  
    this.mElmBottom = bottom / imageH;  
    this.mElmTop = top / imageH;  
    this._setTexInfo();  
}
```

Support Accesses to Sprite Pixels in TextureRenderable

Edit the `texture_renderable_pixel_collision.js` file, and modify the `_pixelAlphaValue()` function to support pixel accesses with a sprite element index offset:

```
TextureRenderable.prototype._pixelAlphaValue = function (x, y) {  
    y += this.mElmBottomIndex;  
    x += this.mElmLeftIndex;  
    x = x * 4;  
    y = y * 4;  
    return this.mColorArray[(y * this.mTextureInfo.mWidth) + x + 3];  
}
```

1440 Test Per-Pixel Collision for Sprites in MyGame

1441 The code for testing this project is a simple modification from previous projects, and the
1442 details are not listed. It is important to note the different object types in the scene.

- 1443 • **Portal minion:** A simple TextureRenderable object
- 1444 • **Hero and Brain:** SpriteRenderable objects where the textures shown
1445 on the geometries are sprite elements defined in the `minion_sprite.`
1446 `.png` sprite sheet
- 1447 • **Left and Right minions:** SpriteAnimateRenderable objects with
1448 sprite elements defined in the top two rows of the `minion_sprite.`
1449 `.png` animated sprite sheet

1450 Observation

1451 You can now run this project and observe the correct results from the collisions of the
1452 different object types:

- 1453 1. Try moving the Hero object and observe how the Brain object
1454 constantly seeks out and collides with it. This is the case of
1455 collision between two SpriteRenderable objects.
- 1456 2. Press the L/R keys and then move the Portal minion with the
1457 WASD keys to collide with the Left or Right minions. Remember
1458 that you can rotate the Portal minion with the P key. This
1459 is the case of collision between TextureRenderable and
1460 SpriteAnimatedRenderable objects.
- 1461 3. Press the H key and then move the Portal minion to collide
1462 with the Hero object. This is the case of collision between
1463 TextureRenderable and SpriteRenderable objects.
- 1464 4. Press the B key and then move the Portal minion to collide with
1465 the Brain object. This is the case of collision between rotated
1466 TextureRenderable and SpriteRenderable objects.

Summary

This chapter showed you how to encapsulate common behaviors of objects in games and demonstrated the benefits of the encapsulation in the forms of a simpler and better organized control logic in the client's MyGame test levels. You reviewed vectors in 2D space. A vector is defined by its direction and magnitude. Vectors are convenient for describing displacements (velocities). You reviewed some foundational vector operations, including normalization of a vector and how to calculate both dot and cross products. You worked with these operators to implement the front-facing direction capability and create simple autonomous behaviors such as pointing toward a specific object and chasing.

The need for detecting object collisions became a prominent omission as the behaviors of objects increased in sophistication. The axis-aligned bounding boxes, or AABBs, were introduced as a crude, yet computationally efficient solution for approximating object collisions. You learned the algorithm for per-pixel-accurate collision detection and that its accuracy comes at the cost of performance. You now understand how to mitigate the computational cost in two ways. First, you invoke the pixel-accurate procedure only when the objects are sufficiently close to each other, such as when their bounding boxes collide. Second, you invoke the pixel iteration process based on the texture with a lower resolution.

When implementing pixel-accurate collision, you began with tackling the basic case of working with axis-aligned textures. After that implementation, you went back and added support for collision detection between rotated textures. Finally, you generalized the implementation to support collisions between sprite elements. Solving the easiest case first lets you test and observe the results and helps define what you might need for the more advanced problems (rotation and subregions of a texture in this case).

At the beginning of this chapter, your game engine supported interesting sophistication in drawing ranging from the abilities to define WC space, to view the WC space with the Camera object, and to draw visually pleasing textures and animations on objects. However, there was no infrastructure for supporting the behaviors of the objects. This shortcoming resulted in clustering of initialization and control logic in the client-level implementations. With the object behavior abstraction, mathematics, and collision algorithms introduced and implemented in this chapter, your game engine functionality is now better balanced. The clients of your game engine now have tools for encapsulating specific behaviors and detecting collisions. The next chapter reexamines

1501 and enhances the functionality of the Camera object. You will learn to control and
1502 manipulate the Camera object and work with multiple Camera objects in the same game.

1503 Game Design Considerations

1504 Chapters 1–5 introduced foundation techniques for drawing, moving, and animating
1505 objects on the screen. The Scene Objects project from Chapter 4 described a simple
1506 interaction behavior and showed you how to change the game screen based on the
1507 location of a rectangle: recall that moving the rectangle to the left boundary caused
1508 the level to visually change, while the Audio Support project added contextual sound
1509 to reinforce the overall sense of presence. Although it's possible to build an intriguing
1510 (albeit simple) puzzle game using only the elements from Chapters 1 to 5, things get
1511 much more interesting when you can integrate object detection and collision triggers;
1512 these behaviors form the basis for many common game mechanics and provide
1513 opportunities to design a wide range of interesting gameplay scenarios.

1514 Starting with the Game Objects project, you can see how the screen elements start
1515 working together to convey the game setting; even with the interaction in this project
1516 limited to character movement, the setting is beginning to resolve into something that
1517 conveys a sense of place. The hero character appears to be flying through a moving scene
1518 populated by a number of mechanized robots, and there's a small object in the center of
1519 the screen that you might imagine could become some kind of special pickup.

1520 Even at this basic stage of development, it's possible to brainstorm game mechanics
1521 that could potentially form the foundation for a full game. If you were designing a
1522 simple game mechanic based on only the screen elements found in the Game Objects
1523 project, what kind of behaviors would you choose and what kind of actions would you
1524 require the player to perform? As one example, imagine that the hero character must
1525 avoid colliding with the flying robots and that perhaps some of the robots will detect
1526 and pursue the hero in an attempt to stop the player's progress; maybe the hero is also
1527 penalized in some way if they come into contact with a robot. Imagine perhaps that the
1528 small object in the center of the screen allows the hero to be invincible for a fixed period
1529 of time and that we've designed the level to require temporary invincibility to reach the
1530 goal, thus creating a more complex and interesting game loop (e.g., avoid the pursuing
1531 robots to reach the power up, activate the power up and become temporarily invincible,
1532 use invincibility to reach the goal). With these few basic interactions, we've opened
1533 opportunities to explore mechanics and level designs that will feel very familiar from

many different kinds of games, all with just the inclusion of the object detection, chase, and collision behaviors covered in Chapter 6. Try this design exercise yourself using just the elements shown in the Game Objects project: What kinds of simple conditions and behaviors might you design to make your experience unique? How many ways can you think of to use the small object in the center of the screen? The final design project in Chapter 12 will explore these themes in greater detail.

This is also a good opportunity to brainstorm some of the other nine elements of game design discussed in Chapter 1. What if the game wasn't set in space with robots? Perhaps the setting is in a forest, or under water, or even something completely abstract. How might you incorporate audio to enhance the sense of presence and reinforce the game setting? You'll probably be surprised by the variety of settings and scenarios you come up with. Limiting yourself to just the elements and interactions covered through Chapter 6 is actually a beneficial exercise as design constraints often help the creative process by shaping and guiding your ideas. Even the most advanced video games typically have a fairly basic set of core game loops as their foundation.

The Vectors: Front and Chase project is interesting from both a game mechanic and presence perspective. Many games, of course, require objects in the game world to detect the hero character and will either chase or try to avoid the player (or both if the object has multiple states). The project also demonstrates two different approaches to chase behavior, instant and smooth pursuit, and the game setting will typically influence which behavior you choose to implement. The choice between instant and smooth pursuit is a great example of subtle behaviors that can significantly influence the sense of presence. If you were designing a game where ships were interacting on the ocean, for example, you would likely want their pursuit behavior to take real-world inertia and momentum into consideration because ships can't instantly turn and respond to changes in movement; rather, they move smoothly and gradually, demonstrating a noticeable delay in how quickly they can respond to a moving target. Most objects in the physical world will display the same inertial and momentum constraint to some degree, but there are also situations where you may want game objects to respond directly to path changes (or, perhaps, you want to intentionally flout real-world physics and create a behavior that isn't based on the limitations of physical objects). The key is to always be intentional about your design choices, and it's good to remember that virtually no implementation details are too small to be noticed by players.

The Bounding Box and Collisions project introduces the key element of detection to your design arsenal, allowing you to begin including more robust cause-and-effect

CHAPTER 6 DEFINING BEHAVIORS AND DETECTING COLLISIONS

mechanics that form the basis for many game interactions. Chapter 6 discusses the trade-offs of choosing between the less precise but more performant bounding box collision detection method and the precise but resource-intensive per-pixel detection method. There are many situations where the bounding-box approach is sufficient, but if players perceive collisions to be arbitrary because the bounding boxes are too different from the actual visual objects, it can negatively impact the sense of presence. Detection and collision are even more powerful design tools when coupled with the result from the Per-Pixel Collisions project. Although the dye pack in this example was used to indicate the first point of collision, you can imagine building interesting causal chains around a new object being produced as the result of two objects colliding (e.g., player pursues object, player collides with object, object “drops” a new object that enables the player to do something they couldn’t do before). Game objects that move around the game screen will typically be animated, of course, so the Sprite Pixel Collisions project describes how to implement collision detection when the object boundaries aren’t stationary.

With the addition of the techniques in Chapter 6, you now have a critical mass of behaviors that can be combined to create truly interesting game mechanics covering the spectrum from action games to puzzlers. Of course, game mechanic behaviors are only one of the nine elements of game design and typically aren’t sufficient on their own to create a magical gameplay experience: the setting, visual style, meta-game elements, and the like all have something important to contribute. The good news is that creating a memorable game experience need not be as elaborate as you often believe and great games continue being produced based on relatively basic combinations of the behaviors and techniques covered in Chapters 1–6. The games that often shine the brightest aren’t always the most complex, but rather they’re often the games where every aspect of each of the nine elements of design is intentional and working together in harmony. If you give the appropriate attention and focus to all aspects of the game design, you’re on a great track to produce something great whether you’re working on your own or you’re part of a large team.

Author Query

Chapter No.: 6 0005244099

Queries	Details Required	Author's Response
AU1	All occurrences of “two-dimensional,” “2-dimensional,” and the like have been changed to “2D,” etc., for consistency. Please check if okay.	

Uncorrected Proof

CHAPTER 7

1

Manipulating the Camera

2

After completing this chapter, you will be able to

- Implement operations that are commonly employed in manipulating a camera
- Interpolate values between old and new to create a smooth transition
- Understand how some motions or behaviors can be described by simple mathematical formulations
- Build games with multiple camera views
- Transform positions from the mouse-clicked pixel to the World Coordinate (WC) position
- Program with mouse input in a game environment with multiple cameras

3

4

5

6

7

8

9

10

11

12

13

Introduction

14

Your game engine is now capable of representing and drawing objects. With the basic abstraction mechanism introduced in the previous chapter, the engine can also support the interactions and behaviors of these objects. This chapter refocuses the attention on controlling and interacting with the Camera object that abstracts and facilitates the presentation of the game objects on the canvas. In this way, your game engine will be able to control and manipulate the presentation of visually appealing game objects with well-structured behaviors.

15

16

17

18

19

20

21

Figure 7-1 presents a brief review of the Camera object abstraction that was introduced in Chapter 3. The Camera object allows the game programmer to define a World Coordinate (WC) window of the game world to be displayed into a viewport on the HTML canvas. The WC window is the bounds defined by a WC center and a

22

23

24

25

dimension of $W_{wc} \times H_{wc}$. A viewport is a rectangular area on the HTML canvas with the lower-left corner located at (V_x, V_y) and a dimension of $W_v \times H_v$. The Camera object's `setViewAndCameraMatrix()` function encapsulates the details and enables the drawing of all game objects inside the WC window bounds to be displayed in the corresponding viewport.

this figure will be printed in b/w

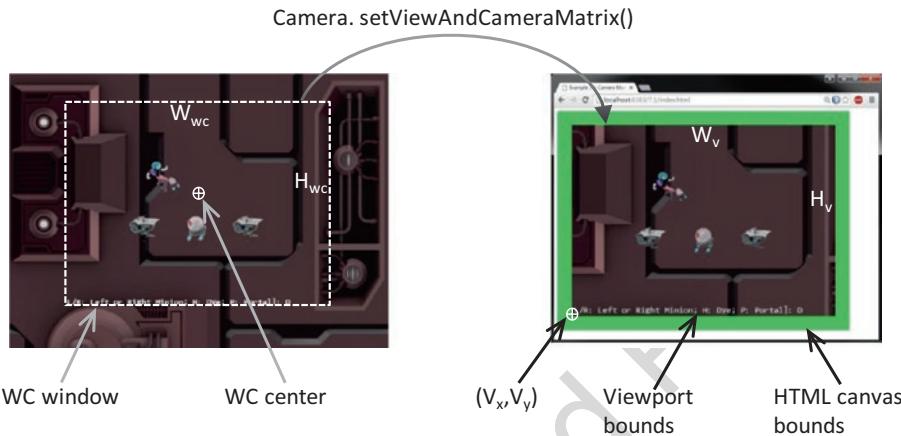


Figure 7-1. Review of WC parameters that define a Camera object

31 **Note** In this book, the WC window or WC bounds are used to refer to the WC
32 window bounds.

33 The Camera object abstraction allows the game programmer to ignore the details
34 of WC bounds and the HTML canvas and focus on designing a fun and entertaining
35 gameplay experience. Programming with a Camera object in a game level should reflect
36 the use of a physical video camera in the real world. For example, you may want to pan
37 the camera to show your audiences the environment, you may want to attach the camera
38 on an actress and share her journey with your audience, or you may want to play the role
39 of a director and instruct the actors in your scene to stay within the visual ranges of the
40 camera. The distinct characteristics of these examples, such as panning or following a
41 character's view, are the high-level functional specifications. Notice that in the real world
42 you do not specify coordinate positions or bounds of windows.

This chapter introduces some of the most commonly encountered camera manipulation operations including clamping, panning, and zooming. Solutions in the form of interpolation will be derived to alleviate annoying or confusing abrupt transitions resulting from the manipulation of cameras. You will also learn about supporting multiple camera views in the same game level and working with mouse input.

43
44
45
46
47

Camera Manipulations

48

In a 2D world, you may want to clamp or restrict the movements of objects to be within the bounds of a camera, to pan or move the camera, or to zoom the camera into or away from specific areas. These high-level functional specifications can be realized by strategically changing the parameters of the Camera object: the WC center and the $W_{wc} \times H_{wc}$ of the WC window. The key is to create convenient functions for the game developers to manipulate these values in the context of the game. For example, instead of increasing/decreasing the width/height of the WC windows, zoom functions can be defined for the programmer.

49
50
51
52
53
54
55
56

The Camera Manipulations Project

57
58
59
60
61

This project demonstrates how to implement intuitive camera manipulation operations by working with the WC center, width, and height of the Camera object. You can see an example of this project running in Figure 7-2. The source code to this project is defined in the chapter7/7.1.camera_manipulations folder.



Figure 7-2. Running the Camera Manipulations project

62 The controls of the project are as follows:

- 63 • **WASD keys:** Move the Dye character (the Hero object). Notice that
64 the camera WC window updates to follow the Hero object when it
65 attempts to move beyond 90 percent of the WC bounds.
- 66 • **Arrow keys:** Move the Portal object. Notice that the Portal object
67 cannot move beyond 80 percent of the WC bounds.
- 68 • **L/R/P/H keys:** Select the Left minion, Right minion, Portal object,
69 or Hero object to be the object in focus; the L/R keys also set the
70 camera to center on the Left or Right minion.
- 71 • **N/M keys:** Zoom into or away from the center of the camera.

- **J/K keys:** Zoom into or away while ensuring the constant relative position of the currently in-focus object. In other words, as the camera zooms, the positions of all objects will change except that of the in-focus object. 72
73
74
75

The goals of the project are as follows: 76

- To experience some of the common camera manipulation operations 77
- To understand the mapping from manipulation operations to the corresponding camera parameter values that must be altered 78
79
- To implement camera manipulation operations 80

You can find the following external resources in the assets folder: the fonts folder that contains the default system fonts and three texture images (`minion_portal.png`, `minion_sprite.png`, and `bg.png`). The Portal object is represented by the first texture image, the remaining objects are sprite elements of `minion_sprite.png`, and the background is a large TextureRenderable object texture mapped with `bg.png`. 81
82
83
84
85

Organize the Source Code

To accommodate the increase in functionality and the complexity of the Camera class you will create a separate folder for storing the related source code files. Similar to the case of dividing the complicated source code of TextureRenderable into multiple files, in this project the Camera class implementation will be separated into three files. 86
87
88
89
90

- `camera_main.js` for implementing the basic functionality from previous projects 91
92
- `camera_manipulation.js` for supporting the newly introduced manipulation operations 93
94
- `camera.js` for serving as the class access point. 95

The implementation steps are as follows: 96

1. Create a new folder called `cameras` in `src/engine`. Move the `camera.js` file into this folder and rename it to `camera_main.js`. 97
98

99 2. Create a new file in `src/engine/cameras` and name it `camera_`
 100 `manipulation.js`. This file will be used to extend the Camera class
 101 functionality in supporting manipulations. Add in the following
 102 code to import and export the basic Camera class functionality.
 103 For now, this file does not contain any useful source code and
 104 thus does not serve any purpose. You will define the appropriate
 105 extension functions in the following subsection.

```
106 import Camera from "./camera_main.js";  

107 // new functionality to be defined here in the next subsection  

108 export default Camera;
```

109 3. Create a new `camera.js` to serve as the Camera access point by
 110 adding the following code:

```
111 import Camera from "./camera_manipulation.js";  

112 export default Camera;
```

113 With this structure of the source code files, `camera_main.js` implements all the
 114 basic functionality and exports to `camera_manipulation.js` that defines additional
 115 functionality for the Camera class. Finally, `camera.js` imports the extended functions
 116 from `camera_manipulation.js`. The users of the Camera class can simply import from
 117 `camera.js` and will have access to all of the defined functionality. This allows `camera.`
 118 `js` to serve as the access point to the Camera class while hiding the details of the
 119 implementation source code structure.

120 Support Clamping to Camera WC Bounds

121 Edit `camera_main.js` to import bounding box functionality and define a function to
 122 clamp the bounds associated with a Transform object to the camera WC bound:

```
123 import * as glSys from "../core/gl.js";  

124 import BoundingBox from "../bounding_box.js";  

125 import { eBoundCollideStatus } from "../bounding_box.js";  

126 ... identical to previous code ...  

127 clampAtBoundary(aXform, zone) {
```

```

let status = this.collideWCBound(aXform, zone);          128
if (status !== eBoundCollideStatus.eInside) {           129
    let pos = aXform.getPosition();                      130
    if ((status & eBoundCollideStatus.eCollideTop) !== 0) { 131
        pos[1] = (this.getWCCenter())[1] +                132
            (zone * this.getWCHeight() / 2) -             133
            (aXform.getHeight() / 2);                     134
    }
    if ((status & eBoundCollideStatus.eCollideBottom) !== 0) { 135
        pos[1] = (this.getWCCenter())[1] -                136
            (zone * this.getWCHeight() / 2) +             137
            (aXform.getHeight() / 2);                     138
    }
    if ((status & eBoundCollideStatus.eCollideRight) !== 0) { 139
        pos[0] = (this.getWCCenter())[0] +                140
            (zone * this.getWCWidth() / 2) -              141
            (aXform.getWidth() / 2);                     142
    }
    if ((status & eBoundCollideStatus.eCollideLeft) !== 0) { 143
        pos[0] = (this.getWCCenter())[0] -                144
            (zone * this.getWCWidth() / 2) +              145
            (aXform.getWidth() / 2);                     146
    }
}
return status;
}

```

The `aXform` object can be the `Transform` of a `GameObject` or `Renderable` object. The `clampAtBoundary()` function ensures that the bounds of the `aXform` remain inside the WC bounds of the camera by clamping the `aXform` position. The `zone` variable defines a percentage of clamping for the WC bounds. For example, a 1.0 would mean clamping to the exact WC bounds, while a 0.9 means clamping to a bound that is 90 percent of the current WC window size. It is important to note that the `clampAtBoundary()` function operates only on bounds that collide with the camera WC bounds. For example, if the `aXform` object has its bounds that are completely outside of the camera WC bounds, it will remain outside.

Define Camera Manipulation Operations in camera_manipulation.js File

- 163 1. Edit `camera_manipulate.js`. Ensure you are adding code between
164 the initial import and final export of the Camera class functionality.
- 165 2. Import the bounding box collision status, and define the
166 `panWidth()` function to pan the camera based on the bounds
167 of a `Transform` object. This function is complementary to
168 the `clampAtBoundary()` function, where instead of changing
169 the `aXform` position, the camera is moved to ensure the
170 proper inclusion of the `aXform` bounds. As in the case of the
171 `clampAtBoundary()` function, the camera will not be changed if
172 the `aXform` bounds are completely outside the tested WC bounds
173 area.

```

179
180     import { eBoundCollideStatus } from "../bounding_box.js";
181
182     Camera.prototype.panWith = function (aXform, zone) {
183         let status = this.collideWCBound(aXform, zone);
184         if (status !== eBoundCollideStatus.eInside) {
185             let pos = aXform.getPosition();
186             let newC = this.getWCCenter();
187
188             if ((status & eBoundCollideStatus.eCollideTop) !== 0) {
189                 newC[1] = pos[1]+(aXform.getHeight() / 2) -
190                             (zone * this.getWCHeight() / 2);
191             }
192             if ((status & eBoundCollideStatus.eCollideBottom) !== 0) {
193                 newC[1] = pos[1] - (aXform.getHeight() / 2) +
194                             (zone * this.getWCHeight() / 2);
195             }
196             if ((status & eBoundCollideStatus.eCollideRight) !== 0) {

```

```

        newC[0] = pos[0] + (aXform.getWidth() / 2) -
194          (zone * this.getWCWidth() / 2);
195      }
196      if ((status & eBoundCollideStatus.eCollideLeft) !== 0) {
197          newC[0] = pos[0] - (aXform.getWidth() / 2) +
198            (zone * this.getWCWidth() / 2);
199      }
200  }
201 }
202

```

3. Define camera panning functions `panBy()` and `panTo()` by
appending to the `Camera` class prototype. These two functions
change the camera WC center by adding a delta to it or moving it
to a new location.

```

Camera.prototype.panBy = function (dx, dy) {
207    this.mWCCenter[0] += dx;
208    this.mWCCenter[1] += dy;
209}
210
Camera.prototype.panTo = function (cx, cy) {
211    this.setWCCenter(cx, cy);
212}
213

```

4. Define functions to zoom the camera with respect to the center or
a target position:

```

Camera.prototype.zoomBy = function (zoom) {
216    if (zoom > 0) {
217        this.setWCWidth(this.getWCWidth() * zoom);
218    }
219}
220
Camera.prototype.zoomTowards = function (pos, zoom) {
221    let delta = [];
222    vec2.sub(delta, pos, this.mWCCenter);
223    vec2.scale(delta, delta, zoom - 1);
224    vec2.sub(this.mWCCenter, this.mWCCenter, delta);
225    this.zoomBy(zoom);
226

```

```

227 }
228     The zoomBy() function zooms with respect to the center of the camera, and the
229     zoomTowards() function zooms with respect to a world coordinate position. If the zoom
230     variable is greater than 1, the WC window size becomes larger, and you will see more of
231     the world in a process we intuitively know as zooming out. A zoom value of less than 1
232     zooms in. Figure 7-3 shows the results of zoom=0.5 for zooming with respect to the center
233     of WC and with respect to the position of the Hero object.
234

```

this figure will be printed in b/w

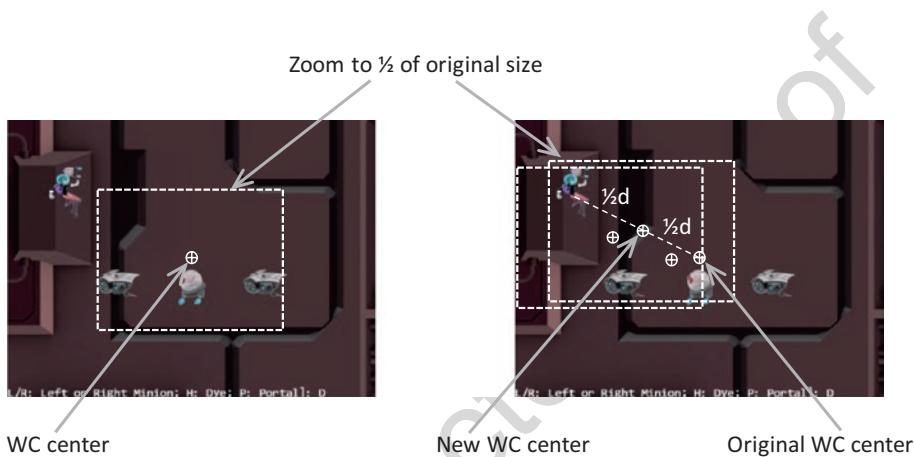


Figure 7-3. Zooming toward the WC Center and toward a target position

235 Manipulating the Camera in MyGame

```

236 There are two important functionalities to be tested: panning and zooming. The only
237 notable changes to the MyGame class are in the update() function. The init(), load(),
238 unload(), and draw() functions are similar to previous projects and can be found in the
239 project source code.

```

```

240 update() {
241     let zoomDelta = 0.05;
242     let msg = "L/R: Left or Right Minion; H: Dye; P: Portal]: ";
243     // ... code to update each object not shown
244     // Brain chasing the hero
245     let h = [];

```

```
if (!this.mHero.pixelTouches(this.mBrain, h)) { 246
    this.mBrain.rotateObjPointTo( 247
        this.mHero.getXform().getPosition(), 0.01);
    engine.GameObject.prototype.update.call(this.mBrain); 248
}
250

// Pan camera to object
251
if (engine.input.isKeyClicked(engine.input.keys.L)) {
    this.mFocusObj = this.mLMinion; 252
    this.mChoice = 'L'; 253
    this.mCamera.panTo(this.mLMinion.getXform().getXPos(),
        this.mLMinion.getXform().getYPos()); 254
}
255

if (engine.input.isKeyClicked(engine.input.keys.R)) {
    this.mFocusObj = this.mRMinion; 256
    this.mChoice = 'R'; 257
    this.mCamera.panTo(this.mRMinion.getXform().getXPos(),
        this.mRMinion.getXform().getYPos()); 258
}
259

if (engine.input.isKeyClicked(engine.input.keys.P)) {
    this.mFocusObj = this.mPortal; 260
    this.mChoice = 'P'; 261
}
262

if (engine.input.isKeyClicked(engine.input.keys.H)) {
    this.mFocusObj = this.mHero; 263
    this.mChoice = 'H';
}
264

// zoom
265
if (engine.input.isKeyClicked(engine.input.keys.N)) {
    this.mCamera.zoomBy(1 - zoomDelta); 266
}
267

if (engine.input.isKeyClicked(engine.input.keys.M)) {
    this.mCamera.zoomBy(1 + zoomDelta); 268
}
269

if (engine.input.isKeyClicked(engine.input.keys.J)) { 270
}
271

}
272

if (engine.input.isKeyClicked(engine.input.keys.N)) { 273
    this.mCamera.zoomBy(1 - zoomDelta); 274
}
275

if (engine.input.isKeyClicked(engine.input.keys.M)) { 276
    this.mCamera.zoomBy(1 + zoomDelta); 277
}
278

if (engine.input.isKeyClicked(engine.input.keys.J)) { 279
}
```

CHAPTER 7 MANIPULATING THE CAMERA

```
280     this.mCamera.zoomTowards(  
281             this.mFocusObj.getXform().getPosition(),  
282             1 - zoomDelta);  
283 }  
284 if (engine.input.isKeyClicked(engine.input.keys.K)) {  
285     this.mCamera.zoomTowards(  
286             this.mFocusObj.getXform().getPosition(),  
287             1 + zoomDelta);  
288 }  
289 // interaction with the WC bound  
290 this.mCamera.clampAtBoundary(this.mBrain.getXform(), 0.9);  
291 this.mCamera.clampAtBoundary(this.mPortal.getXform(), 0.8);  
292 this.mCamera.panWith(this.mHero.getXform(), 0.9);  
293 this.mMsg.setText(msg + this.mChoice);  
294 }
```

295 In the listed code, the first four if statements select the in-focus object, where L and
296 R keys also re-center the camera by calling the panTo() function with the appropriate
297 WC positions. The second set of four if statements control the zoom, either toward the
298 WC center or toward the current in-focus object. Then the function clamps the Brain
299 and Portal objects to within 90 percent and 80 percent of the WC bounds, respectively.
300 The function finally ends by panning the camera based on the transform (or position) of
301 the Hero object.

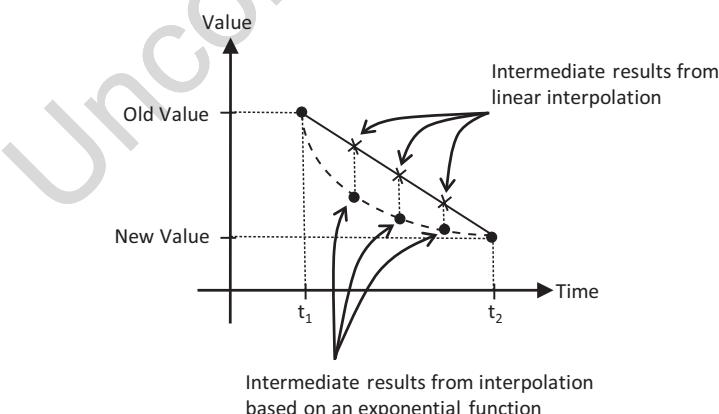
302 You can now run the project and move the Hero object with the WASD keys. Move
303 the Hero object toward the WC bounds to observe the camera being pushed. Continue
304 pushing the camera with the Hero object; notice that because of the clampAtBoundary()
305 function call, the Portal object will in turn be pushed such that it never leaves the
306 camera WC bounds. Now press the L/R key to observe the camera center switching to
307 the center on the Left or Right minion. The N/M keys demonstrate straightforward
308 zooming with respect to the center. To experience zooming with respect to a target, move
309 the Hero object toward the top left of the canvas and then press the H key to select it as
310 the zoom focus. Now, with your mouse pointer pointing at the head of the Hero object,
311 you can press the K key to zoom out first and then the J key to zoom back in. Notice that
312 as you zoom, all objects in the scene change positions except the areas around the Hero

object. Zooming into a desired region of a world is a useful feature for game developers
with many applications. You can experience moving the Hero object around while
zooming into/away from it.

Interpolation

It is now possible to manipulate the camera based on high-level functions such as pan or zoom. However, the results are often sudden or visually incoherent changes to the rendered image, which may result in annoyance or confusion. For example, in the previous project, the L or R key causes the camera to re-center with a simple assignment of new WC center values. The abrupt change in camera position results in the sudden appearance of a seemingly new game world. This is not only visually distracting but can also confuse the player as to what has happened.

When new values for camera parameters are available, instead of assigning them and causing an abrupt change, it is desirable to morph the values gradually from the old to the new over time or *interpolate* the values. For example, as illustrated in Figure 7-4, at time t_1 , a parameter with the old value is to be assigned a new one. In this case, instead of updating the value abruptly, interpolation will change the value gradually over time. It will compute the intermediate results with decreasing values and complete the change to the new value at a later time t_2 .



this figure will be printed in b/w

Figure 7-4. Interpolating values based on linear and exponential functions

331 Figure 7-4 shows that there are multiple ways to interpolate values over time. For
332 example, linear interpolation computes intermediate results according to the slope of
333 the line connecting the old and new values. In contrast, an exponential function may
334 compute intermediate results based on percentages from previous values. In this way,
335 with linear interpolation, a camera position would move from an old to new position
336 with a constant speed similar to a moving (or panning) a camera at some constant
337 speed. In comparison, the interpolation based on the given exponential function would
338 move the camera position rapidly at first and then slow down quickly over time giving a
339 sensation of moving and focusing the camera on a new target.

340 Human motions and movements typically follow the exponential interpolation
341 function. For example, try turning your head from facing the front to facing the right
342 or moving your hand to pick up an object on your desk. Notice that in both cases,
343 you began with a relatively quick motion and slowed down significantly when the
344 destination is in close proximity. That is, you probably started by turning your head
345 quickly and slowed down rapidly as your view approaches your right side, and it is likely
346 your hand started moving quickly toward the object and slowed down significantly when
347 the hand is almost reaching the object. In both of these examples, your displacements
348 followed the exponential interpolation function as depicted in Figure 7-4, quick changes
349 followed by a rapid slow down as the destination approaches. This is the function you
350 will implement in the game engine because it mimics human movements and is likely to
351 seem natural to human players.

352 **Note** Linear interpolation is often referred to as *LERP* or *lerp*. The result of lerp is
353 the linear combination of an initial and a final value. In this chapter, and in almost
354 all cases, the exponential interpolation depicted in Figure 7-4 is approximated by
355 repeatedly applying the lerp function where in each invocation, the initial value is
356 the result of the previous lerp invocation. In effect, approximating the exponential
357 function with a piece-wise linear function.

This section introduces the Lerp and LerpVec2 utility classes to support smooth and gradual camera movements resulting from camera manipulation operations. 358

The Camera Interpolations Project 360

This project demonstrates the smoother and visually more appealing interpolated results from camera manipulation operations. You can see an example of this project running in Figure 7-5. The source code to this project is defined in the chapter7/7.2.camera_interpolations folder. 361

362
363
364
this figure will be printed in b/w



L/R: Left or Right Minion; H: Dye; P: Portal]: D

Figure 7-5. Running the Camera Interpolations project

CHAPTER 7 MANIPULATING THE CAMERA

365 The controls of the project are identical to the previous project:

- 366 • **WASD keys:** Move the Dye character (the Hero object). Notice that
367 the camera WC window updates to follow the Hero object when it
368 attempts to move beyond 90 percent of the WC bounds.
- 369 • **Arrow keys:** Move the Portal object. Notice that the Portal object
370 cannot move beyond 80 percent of the WC bounds.
- 371 • **L/R/P/H keys:** Select the Left minion, Right minion, Portal object,
372 or Hero object to be the object in focus. The L/R keys also set the
373 camera to focus on the Left or Right minion.
- 374 • **N/M keys:** Zoom into or away from the center of the camera.
- 375 • **J/K keys:** Zoom into or away while ensuring constant relative
376 position of the currently in-focus object. In other words, as the
377 camera zooms, the positions of all objects will change except that of
378 the in-focus object.

379 The goals of the project are as follows:

- 380 • To understand the concept of interpolation between given values
- 381 • To implement interpolation supporting gradual camera parameter
382 changes
- 383 • To experience interpolated changes in camera parameters

384 As in previous projects, you can find external resource files in the assets folder.

385 Interpolation as a Utility

386 Similar to the `Transform` class supporting transformation functionality and the
387 `BoundingBox` class supporting collision detection, a `Lerp` class can be defined to support
388 interpolation of values. To keep the source code organized, a new folder should be
389 defined to store these utilities.

390 Create the `src/engine/utils` folder and move the `transform.js` and `bounding_`
391 `box.js` files into this folder.

The Lerp Class

Define the Lerp class to compute interpolation between two values:

1. Create a new file in the `src/engine/utils` folder, name it `lerp.`

`js`, and define the constructor. This class is designed to interpolate values from `mCurrentValue` to `mFinalValue` in the duration of `mCycles`. During each update, intermediate results are computed based on the `mRate` increment on the difference between `mCurrentValue` and `mFinalValue`, as shown next.

```
class Lerp {
    constructor(value, cycles, rate) {
        this.mCurrentValue = value;      // begin value of interpolation
        this.mFinalValue = value;       // final value of interpolation
        this.mCycles = cycles;
        this.mRate = rate;

        // Number of cycles left for interpolation
        this.mCyclesLeft = 0;
    }

    ... implementation to follow ...
}
```

2. Define the function that computes the intermediate results:

```
// subclass should override this function for non-scalar values
_interpolateValue() {
    this.mCurrentValue = this.mCurrentValue + this.mRate *
        (this.mFinalValue - this.mCurrentValue);
}
```

Note that the `_interpolateValue()` function computes a result that linearly interpolates between `mCurrentValue` and `mFinalValue`. In this way, `mCurrentValue` will be set to the intermediate value during each iteration approximating an exponential curve as it approaches the value of the `mFinalValue`.

CHAPTER 7 MANIPULATING THE CAMERA

- 422 3. Define a function to configure the interpolation. The `mRate`
423 variable defines how quickly the interpolated result approaches
424 the final value. A `mRate` of 0.0 will result in no change at all, where
425 1.0 causes instantaneous change. The `mCycle` variable defines the
426 duration of the interpolation process.

```
427 config(stiffness, duration) {  
428     this.mRate = stiffness;  
429     this.mCycles = duration;  
430 }
```

- 431 4. Define relevant getter and setter functions. Note that the
432 `setFinal()` function both sets the final value and triggers a new
433 round of interpolation computation.

```
434     get() { return this.mCurrentValue; }  
  
435     setFinal(v) {  
436         this.mFinalValue = v;  
437         this.mCyclesLeft = this.mCycles; // will trigger interpolation  
438     }
```

- 439 5. Define the function to trigger the computation of each
440 intermediate result:

```
441     update() {  
442         if (this.mCyclesLeft <= 0) { return; }  
  
443         this.mCyclesLeft--;  
444         if (this.mCyclesLeft === 0) {  
445             this.mCurrentValue = this.mFinalValue;  
446         } else {  
447             this._interpolateValue();  
448         }  
449     }
```

- 450 6. Finally, make sure to export the defined class:

```
451     export default Lerp;
```

The LerpVec2 Class

452

Since many of the camera parameters are vec2 objects (e.g., the WC center position), it is
453 important to generalize the Lerp class to support the interpolation of vec2 objects:
454

1. Create a new file in the `src/engine/utils` folder, name it `lerp_vec2.js`, and define its constructor:
455
456

```
class LerpVec2 extends Lerp {  
    constructor(value, cycle, rate) {  
        super(value, cycle, rate);  
    }  
    ... implementation to follow ...  
}
```

2. Override the `_interpolateValue()` function to compute
463 intermediate results for vec2:
464

```
_interpolateValue() {  
    vec2.lerp(this.mCurrentValue, this.mCurrentValue,  
              this.mFinalValue, this.mRate);  
}
```

The `vec2.lerp()` function defined in the `gl-matrix.js` file computes the
469 vec2 components for x and y. The computation involved is identical to the
470 `_interpolateValue()` function in the Lerp class.
471

Lastly, remember to update the engine access file, `index.js`, to forward the newly
472 defined Lerp and LerpVec2 functionality to the client.
473

Represent Interpolated Intermediate Results with CameraState

474

The state of a Camera object must be generalized to support gradual changes of
475 interpolated intermediate results. The CameraState class is introduced to accomplish
476 this purpose.
477

1. Create a new file in the `src/engine/cameras` folder, name it
478 `camera_state.js`, import the defined Lerp functionality, and
479 define the constructor:
480

CHAPTER 7 MANIPULATING THE CAMERA

```
481 import Lerp from "../utils/lerp.js";
482 import LerpVec2 from "../utils/lerp_vec2.js";
483
484 class CameraState {
485     constructor(center, width) {
486         this.kCycles = 300; // cycles to complete the transition
487         this.kRate = 0.1; // rate of change for each cycle
488         this.mCenter = new LerpVec2(center, this.kCycles, this.kRate);
489         this.mWidth = new Lerp(width, this.kCycles, this.kRate);
490     }
491 }
492
493 export default CameraState;
```

Observe that `mCenter` and `mWidth` are the only variables required to support camera panning (changing of `mCenter`) and zooming (changing of `mWidth`). Both of these variables are instances of the corresponding `Lerp` classes and are capable of interpolating and computing intermediate results to achieve gradual changes.

2. Define the getter and setter functions:

```
499 getCenter() { return this.mCenter.get(); }
500 getWidth() { return this.mWidth.get(); }
501 setCenter(c) { this.mCenter.setFinal(c); }
502 setWidth(w) { this.mWidth.setFinal(w); }
```

3. Define the update function to trigger the interpolation computation:

```
505 update() {
506     this.mCenter.update();
507     this.mWidth.update();
508 }
```

4. Define a function to configure the interpolation:

```
config(stiffness, duration) {
    this.mCenter.config(stiffness, duration);
    this.mWidth.config(stiffness, duration);
}
```

The stiffness variable is the mRate of Lerp. It defines how quickly the interpolated intermediate results should converge to the final value. As discussed in the Lerp class definition, this is a number between 0 and 1, where 0 means the convergence will never happen and a 1 means instantaneous convergence. The duration variable is the mCycle of Lerp. It defines the number of update cycles it takes for the results to converge. This must be a positive integer value.

Note that as the sophistication of the engine increases, so does the complexity of the supporting code. In this case, you have designed an internal utility class, CameraState, for storing the internal state of a Camera object to support interpolation. This is an internal engine operation. There is no reason for the game programmer to access this class, and thus, the engine access file, index.js, should not be modified to forward the definition.

Integrate Interpolation into Camera Manipulation Operations

The Camera class in camera_main.js must be modified to represent the WC center and width using the newly defined CameraState:

1. Edit the camera_main.js file and import the newly defined CameraState class:

```
import CameraState from "./camera_state.js";
```

2. Modify the Camera constructor to replace the center and width variables with an instance of CameraState:

```
constructor(wcCenter, wcWidth, viewportArray) {
    this.mCameraState = new CameraState(wcCenter, wcWidth);
    ... identical to previous code ...
}
```

CHAPTER 7 MANIPULATING THE CAMERA

- 539 3. Now, edit the `camera_manipulation.js` file to define the functions
540 to update and configure the interpolation functionality of the
541 `CameraState` object:

```
542 Camera.prototype.update = function () {  
543     this.mCameraState.update();  
544 }  
  
545 // For LERP function configuration  
546 Camera.prototype.configLerp = function (stiffness, duration) {  
547     this.mCameraState.config(stiffness, duration);  
548 }
```

- 549 4. Modify the `panBy()` camera manipulation function to support the
550 `CameraState` object as follows:

```
551 Camera.prototype.panBy = function (dx, dy) {  
552     let newC = vec2.clone(this.getWCCenter());  
553     newC[0] += dx;  
554     newC[1] += dy;  
555     this.mCameraState.setCenter(newC);  
556 }
```

- 557 5. Update `panWith()` and `zoomTowards()` functions to receive and
558 set WC center to the newly defined `CameraState` object:

```
559 Camera.prototype.panWith = function (aXform, zone) {  
560     let status = this.collideWCBound(aXform, zone);  
561     if (status !== eBoundCollideStatus.eInside) {  
562         let pos = aXform.getPosition();  
563         let newC = vec2.clone(this.getWCCenter());  
564         if ((status & eBoundCollideStatus.eCollideTop) !== 0)  
565             ... identical to previous code ...  
566         this.mCameraState.setCenter(newC);  
567     }  
568 }
```

```

Camera.prototype.zoomTowards = function (pos, zoom) {
    ... identical to previous code ...
    this.zoomBy(zoom);
this.mCameraState.setCenter(newC);
}

```

Testing Interpolation in MyGame

Recall that the user controls of this project are identical to that from the previous project. The only difference is that in this project, you can expect gradual and smooth transitions between different camera settings. To observe the proper interpolated results, the camera update() function must be invoked at each game scene update.

```

update() {
    let zoomDelta = 0.05;
    let msg = "L/R: Left or Right Minion; H: Dye; P: Portal]: ";
    this.mCamera.update(); // for smoother camera movements
    ... identical to previous code ...
}

```

The call to update the camera for computing interpolated intermediate results is the only change in the my_game.js file. You can now run the project and experiment with the smooth and gradual changes resulting from camera manipulation operations. Notice that the interpolated results do not change the rendered image abruptly and thus maintain the sense of continuity in space from before and after the manipulation commands. You can try changing the stiffness and duration variables to better appreciate the different rates of interpolation convergence.

Camera Shake and Object Oscillation Effects

In video games, shaking the camera can be a convenient way to convey the significance or mightiness of events, such as the appearance of an enemy boss or the collisions between large objects. Similar to the interpolation of values, the camera shake movement can also be modeled by straightforward mathematical formulations.

597 Consider how a camera shake may occur in a real-life situation. For instance, while
 598 shooting with a video camera, say you are surprised or startled by someone or that
 599 something collided with you. Your reaction will probably be slight disorientation followed
 600 by quickly refocusing on the original targets. From the perspective of the camera, this
 601 reaction can be described as an initial large displacement from the original camera center
 602 followed by quick adjustments to re-center the camera. Mathematically, as illustrated
 603 in Figure 7-6, damped simple harmonic motions, which can be represented with the
 604 damping of trigonometric functions, can be used to describe these types of displacements.

605 Note that straight mathematical formulation is precise with perfect predictability.
 606 Such formulation can be suitable for describing regular, normal, or expected behaviors,
 607 for example, the bouncing of a ball or the oscillation of a pendulum. A shaking
 608 effect should involve slight chaotic and unpredictable randomness, for example, the
 609 stabilization of the coffee-carrying hand after an unexpected collision or, as in the
 610 previous example, the stabilization of the video camera after being startled. Following
 611 this reasoning, in this section, you will define a general damped oscillation function and
 612 then inject pseudo-randomness to simulate the slight chaos to achieve the shake effect.

this figure will be printed in b/w

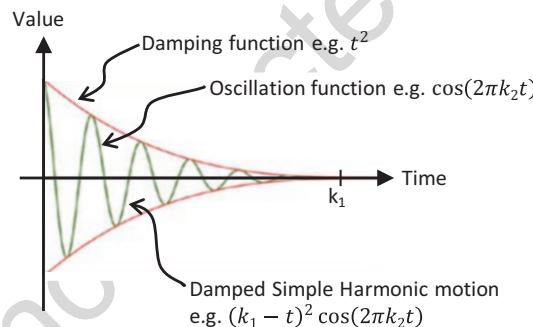


Figure 7-6. The displacements of a damped simple harmonic motion

613 The Camera Shake and Object Oscillate Project

614 This project demonstrates how to implement damped simple harmonic motion to
 615 simulate object oscillation and the injection of pseudo-randomness to create a camera
 616 shake effect. You can see an example of this project running in Figure 7-7. This project
 617 is identical to the previous project except for an additional command to create object
 618 oscillation and camera shake effects. The source code to this project is defined in the
 619 chapter7/7.3.camera_shake_and_object_oscillate folder.



L/R: Left or Right Minion; H: Dye; P: Portal]: D

Figure 7-7. Running the Camera Shake and Object Oscillate project

The following is the new control of this project:

- **Q key:** Initiates the positional oscillation of the Dye character and the camera shake effects.

The following controls are identical to the previous project:

- **WASD keys:** Move the Dye character (the Hero object). Notice that the camera WC window updates to follow the Hero object when it attempts to move beyond 90 percent of the WC bounds.
- **Arrow keys:** Move the Portal object. Notice that the Portal object cannot move beyond 80 percent of the WC bounds.
- **L/R/P/H keys:** Select the Left minion, Right minion, Portal object, or Hero object to be the object in focus. The L/R keys also set the camera to focus on the Left or Right minion.

CHAPTER 7 MANIPULATING THE CAMERA

- **N/M keys:** Zoom into or away from the center of the camera.
- **J/K keys:** Zoom into or away while ensuring constant relative position of the currently in-focus object. In other words, as the camera zooms, the positions of all objects will change except that of the in-focus object.

The goals of the project are as follows:

- To gain some insight into modeling displacements with simple mathematical functions
- To experience the oscillate effect on an object
- To experience the shake effect on a camera
- To implement oscillations as damped simple harmonic motion and to introduce pseudo-randomness to create the camera shake effect

As in previous projects, you can find external resource files in the assets folder.

Abstract the Shake Behavior

The ability to shake the camera is a common and dynamic behavior in many games. However, it is important to recognize that the shake behavior can be applied to more than just the camera. That is, the shaking effect can be abstracted as the perturbation (shaking) of numerical value(s) such as a size, a point, or a position. In the case of camera shake, it just so happened that the numerical values being shaken represent the x and y positions of the camera. For this reason, the shake and associated supports should be general utility functions of the game engine so that they can be applied by the game developer to any numerical values. The following are the new utilities that will be defined:

- **Oscillate:** The base class that implements simple harmonic oscillation of a value over time
- **Shake:** An extension of the Oscillate class that introduces randomness to the magnitudes of the oscillations to simulate slight chaos of the shake effect on a value
- **ShakeVec2:** An extension of the Shake class that expands the Shake behavior to two values such as a position

Create the Oscillate Class to Model Simple Harmonic Motion

662

Because all of the described behaviors depend on simple oscillation, this should be implemented first;

1. Create a new file in the `src/engine/utils` folder and name it `oscillate.js`. Define a class named `Oscillate` and add the following code to construct the object:

```
class Oscillate {
    constructor(delta, frequency, duration) {
        this.mMag = delta;

        this.mCycles = duration; // cycles to complete the transition
        this.mOmega = frequency * 2 * Math.PI; // Converts to radians
        this.mNumCyclesLeft = duration;
    }

    ... implementation to follow ...
}

export default Oscillate;
```

The `delta` variable represents the initial displacements before damping, in WC space. The `frequency` parameter specifies how much to oscillate with a value of 1 representing one complete period of a cosine function. The `duration` parameter defines how long to oscillate in units of game loop updates.

2. Define the damped simple harmonic motion:

```
_nextDampedHarmonic() {
    // computes (Cycles) * cos(Omega * t)
    let frac = this.mNumCyclesLeft / this.mCycles;
    return frac * Math.cos((1 - frac) * this.mOmega);
}
```

Refer to Figure 7-8. `mNumCyclesLeft` is the number of cycles left in the oscillation, or $k-t$, and the `frac` variable, $\frac{k-t}{k}$, is the damping factor. This function returns a value between -1 and 1 and can be scaled as needed.

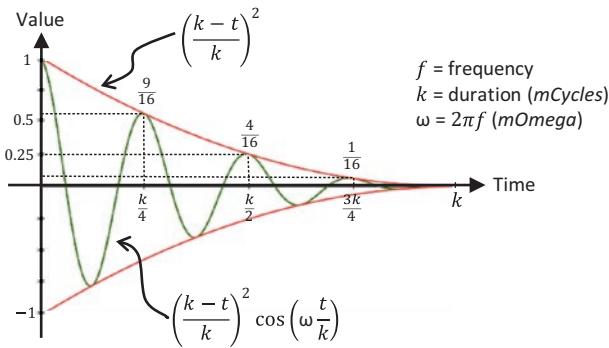


Figure 7-8. The damped simple harmonic motion that specifies value oscillation

- 693 3. Define a protected function to retrieve the value of the next damped
 694 harmonic motion. This function may seem trivial and unnecessary.
 695 However, as you will observe in the next subsection, this function
 696 allows a shake subclass to overwrite and inject randomness.

```

697 // local/protected methods
698 _nextValue() {
699     return (this._nextDampedHarmonic());
700 }
701
702     4. Define functions to check for the end of the oscillation and for
703       restarting the oscillation:
704 done() { return (this.mNumCyclesLeft <= 0); }
705 reStart() { this.mNumCyclesLeft = this.mCycles; }
706
707     5. Lastly, define a public function to trigger the calculation of
708       oscillation. Notice that the computed oscillation result must be
709       scaled by the desired magnitude, mMag:
```

```

710
711     getNext() {
712         this.mNumCyclesLeft--;
713         let v = 0;
714         if (!this.done()) {
715             v = this._nextValue();
716         }
717         return (v * this.mMag);
718     }
719 }
```

Create the Shake Class to Randomize an Oscillation

716

You can now extend the oscillation behavior to convey a sense of shaking by introducing pseudo-randomness into the effect.

1. Create a new file, `shake.js`, in the `src/engine/utils` folder.

719

Define the `Shake` class to extend `Oscillate` and add the following code to construct the object:

720

721

```
import Oscillate from "./oscillate.js";
```

722

```
class Shake extends Oscillate {
```

723

```
    constructor(delta, frequency, duration) {
```

724

```
        super(delta, frequency, duration);
```

725

```
    }
```

726

```
    ... implementation to follow ...
```

727

```
}
```

728

```
export default Shake;
```

729

2. Overwrite the `_nextValue()` to randomize the sign of the oscillation results as follows. Recall that the `_nextValue()` function is called from the public `getNext()` function to retrieve the oscillating value. While the results from the damped simple harmonic oscillation continuously and predictably decrease in magnitude, the associated signs of the values are randomized causing sudden and unexpected discontinuities conveying a sense of chaos from the results of a shake.

730

731

732

733

734

735

736

737

```
_nextValue() {
```

738

```
    let v = this._nextDampedHarmonic();
```

739

```
    let fx = (Math.random() > 0.5) ? -v : v;
```

740

```
    return fx;
```

741

```
}
```

742

Create the ShakeVec2 Class to Model the Shaking of a vec2, or a Position

You can now generalize the shake effect to support the shaking of two values simultaneously. This is a useful utility because positions in 2D games are two-value entities and positions are convenient targets for shake effects. For example, in this project, the shaking of the camera position, a two-value entity, simulates the camera shake effect.

The ShakeVec2 class extends the Shake class to support the shaking of a vec2 object, shaking the values in both the x and y dimensions. The x-dimension shaking is supported via an instance of the Shake object, while the y dimension is supported via the Shake class functionality that is defined in the super class.

1. Create a new file, `shake_vec2.js`, in the `src/engine/utils` folder.

Define the ShakeVec2 class to extend the Shake class. Similar to the constructor parameters of the Shake super classes, the `deltas` and `freqs` parameters are 2D, or `vec2`, versions of magnitude and frequency for shaking in the x and y dimensions. In the constructor, the `xShake` instance variable keeps track of shaking effect in the x dimension. Note the y-component parameters, array indices of 1, in the `super()` constructor invocation. The Shake super class keeps track of the shaking effect in the y dimension.

```

761 class ShakeVec2 extends Shake {
762     constructor(deltas, freqs, duration) {
763         super(deltas[1], freqs[1], duration); // super in y-direction
764         this.xShake = new Shake(deltas[0], freqs[0], duration);
765     }
766     ... implementation to follow ...
767 }
768 export default ShakeVec2;

```

2. Extend the `restart()` and `getNext()` functions to support the second dimension:

```

771 reStart() {
772     super.reStart();
773     this.xShake.reStart();
774 }

```

```

getNext() {
    let x = this.xShake.getNext();
    let y = super.getNext();
    return [x, y];
}

```

Lastly, remember to update the engine access file, `index.js`, to forward the newly defined `Oscillate`, `Shake`, and `ShakeVec2` functionality to the client.

Define the CameraShake Class to Abstract the Camera Shaking Effect

With the defined `ShakeVec2` class, it is convenient to apply the displacements of a pseudo-random damped simple harmonic motion on the position of the Camera. However, the `Camera` object requires an additional abstraction layer.

1. Create a new file, `camera_shake.js`, in the `src/engine/cameras` folder, and define the constructor to receive the camera state, the state parameter, and shake configurations: `deltas`, `freqs`, and `shakeDuration`. The parameter `state` is of datatype `CameraState`, consisting of the camera center position and width.

```

import ShakeVec2 from "../utils/shake_vec2.js";

class CameraShake {
    // state is the CameraState to be shaken
    constructor(state, deltas, freqs, shakeDuration) {
        this.mOrgCenter = vec2.clone(state.getCenter());
        this.mShakeCenter = vec2.clone(this.mOrgCenter);
        this.mShake = new ShakeVec2(deltas, freqs, shakeDuration);
    }
    ... implementation to follow ...
}
export default CameraShake;

```

2. Define the function that triggers the displacement computation for accomplishing the shaking effect. Notice that the shake results are offsets from the original position. The given code adds this offset to the original camera center position.

```

806 update() {
807     let delta = this.mShake.getNext();
808     vec2.add(this.mShakeCenter, this.mOrgCenter, delta);
809 }

```

810 3. Define utility functions: inquire if shaking is done, restart the
 811 shaking, and getter/setter functions.

```

812 done() { return this.mShake.done(); }
813 reShake() {this.mShake.reStart();}
814 getCenter() { return this.mShakeCenter; }
815 setRefCenter(c) {
816     this.mOrgCenter[0] = c[0];
817     this.mOrgCenter[1] = c[1];
818 }

```

819 Similar to CameraState, CameraShake is also a game engine internal utility and
 820 should not be exported to the client game programmer. The engine access file, index.js,
 821 should not be updated to export this class.

822 Modify the Camera to Support Shake Effect

823 With the proper CameraShake abstraction, supporting the shaking of the camera simply
 824 means initiating and updating the shake effect:

825 1. Modify camera_main.js and camera_manipulation.js to import
 826 camera_shake.js as shown:

```
827 import CameraShake from "./camera_shake.js";
```

828 2. In camera_main.js, modify the Camera constructor to initialize a
 829 CameraShake object:

```

830 constructor(wcCenter, wcWidth, viewportArray) {
831     this.mCameraState = new CameraState(wcCenter, wcWidth);
832     this.mCameraShake = null;
833     ... identical to previous code ...
834 }
```

3. Modify step B of the `setViewAndCameraMatrix()` function to use
the `CameraShake` object's center if it is defined: 835
836

```
setViewAndCameraMatrix() {  
    ... identical to previous code ...  
  
    // Step B: Compute the Camera Matrix  
    let center = [];  
    if (this.mCameraShake !== null) {  
        center = this.mCameraShake.getCenter();  
    } else {  
        center = this.getWCCenter();  
    }  
    ... identical to previous code ...  
}
```

4. Modify the `camera_manipulation.js` file to add support to initiate
and restart the shake effect: 848
849

```
Camera.prototype.shake = function (deltas, freqs, duration) {  
    this.mCameraShake = new CameraShake(this.mCameraState,  
                                         deltas, freqs, duration);  
}  
  
// Restart the shake  
Camera.prototype.reShake = function () {  
    let success = (this.mCameraShake !== null);  
    if (success)  
        this.mCameraShake.reShake();  
    return success;  
}
```

5. Continue working with the `camera_manipulation.js` file, and
modify the `update()` function to trigger a camera shake update if
one is defined: 861
862
863

```
Camera.prototype.update = function () {  
    if (this.mCameraShake !== null) {  
        if (this.mCameraShake.done()) {  
            this.mCameraShake.start();  
        }  
    }  
}
```

```

867         this.mCameraShake = null;
868     } else {
869         this.mCameraShake.setRefCenter(this.getWCCenter());
870         this.mCameraShake.update();
871     }
872 }
873 this.mCameraState.update();
874 }
```

Testing the Camera Shake and Oscillation Effects in MyGame

The `my_game.js` file only needs to be modified slightly in the `init()` and `update()` functions to support triggering the oscillation and camera shake effects with the Q key:

1. Define a new instance variable for creating oscillation or bouncing effect on the Dye character:

```

880 init() {
881     ... identical to previous code ...
882
883     // create an Oscillate object to simulate motion
884     this.mBounce = new engine.Oscillate(2, 6, 120);
885         // delta, freq, duration
886 }
```

2. Modify the `update()` function to trigger the bouncing and camera shake effects with the Q key. In the following code, note the advantage of well-designed abstraction. For example, the camera shake effect is opaque where the only information a programmer needs to specify is the actual shake behavior, that is, the shake magnitude, frequency, and duration. In contrast, the oscillating or bouncing effect of the Dye character position is accomplished by explicitly inquiring and using the `mBounce` results.

```

894 update() {
895     ... identical to previous code ...
896
897     if (engine.input.isKeyPressed(engine.input.keys.Q)) {
898         if (!this.mCamera.reShake())
900             this.mCamera.shake([6, 1], [10, 3], 60);
```

```

    // also re-start bouncing effect
    this.mBounce.reStart();
}

if (!this.mBounce.done()) {
    let d = this.mBounce.getNext();
    this.mHero.getXform().incXPosBy(d);
}

this.mMsg.setText(msg + this.mChoice);
}

```

You can now run the project and experience the pseudo-random damped simple harmonic motion that simulates the camera shake effect. You can also observe the oscillation of the Dye character's x position. Notice that the displacement of the camera center position will undergo interpolation and thus result in a smoother final shake effect. You can try changing the parameters when creating the mBounce object or when calling the mCamera.shake() function to experiment with different oscillation and shake configurations. Recall that in both cases the first two parameters control the initial displacements and the frequency (number of cosine periods) and the third parameter is the duration of how long the effects should last.

Multiple Cameras

Video games often present the players with multiple views into the game world to communicate vital or interesting gameplay information, such as showing a mini-map to help the player navigate the world or providing a view of the enemy boss to warn the player of what is to come.

In your game engine, the Camera class abstracts the graphical presentation of the game world according to the source and destination areas of drawing. The source area of the drawing is the WC window of the game world, and the destination area is the viewport region on the canvas. This abstraction already effectively encapsulates and supports the multiple view idea with multiple Camera instances. Each view in the game can be handled with a separate instance of the Camera object with distinct WC window and viewport configurations.

The Multiple Cameras Project

This project demonstrates how to represent multiple views in the game world with multiple Camera objects. You can see an example of this project running in Figure 7-9. The source code to this project is defined in the chapter7/7.4.multiple_cameras folder.



Figure 7-9. Running the Multiple Cameras project

The controls of the project are identical to the previous project:

- **Q key:** Initiates the positional oscillation of the Dye character and the camera shake effects.
- **WASD keys:** Move the Dye character (the Hero object). Notice that the camera WC window updates to follow the Hero object when it attempts to move beyond 90 percent of the WC bounds.

- **Arrow keys:** Move the Portal object. Notice that the Portal object cannot move beyond 80 percent of the WC bounds. 939
940
- **L/R/P/H keys:** Select the Left minion, Right minion, Portal object, or Hero object to be the object in focus. The L/R keys also set the camera to focus on the Left or Right minion. 941
942
943
- **N/M keys:** Zoom into or away from the center of the camera. 944
- **J/K keys:** Zoom into or away while ensuring the constant relative position of the currently in-focus object. In other words, as the camera zooms, the positions of all objects will change except that of the in-focus object. 945
946
947
948

The goals of the project are as follows:

- To understand the camera abstraction for presenting views into the game world 950
951
- To experience working with multiple cameras in the same game level 952
- To appreciate the importance of interpolation configuration for cameras with specific purposes 953
954

As in previous projects, you can find external resource files in the assets folder.

Modify the Camera

The Camera object will be slightly modified to allow the drawing of the viewport with a bound. This will allow easy differentiation of camera views on the canvas.

1. Edit `camera_main.js` and modify the Camera constructor to allow programmers to define a bound number of pixels to surround the viewport of the camera: 959
960
961

```
constructor(wcCenter, wcWidth, viewportArray, bound) { 962
    this.mCameraState = new CameraState(wcCenter, wcWidth); 963
    this.mCameraShake = null; 964

    this.mViewport = []; // [x, y, width, height] 965
    this.mViewportBound = 0; 966
    if (bound !== undefined) { 967
```

```
968     this.mViewportBound = bound;  
969 }  
970 this.mScissorBound = []; // use for bounds  
971 this.setViewport(viewportArray, this.mViewportBound);  
972 // Camera transform operator  
973 this.mCameraMatrix = mat4.create();  
974 // background color  
975 this.mBGColor = [0.8, 0.8, 0.8, 1]; // RGB and Alpha  
976 }
```

977 Please refer to the `setViewport()` function that follows. By
978 default, bound is assumed to be zero, and the camera will draw to
979 the entire `mViewport`. When being nonzero, the bound number of
980 pixels that surround the `mViewport` will be left as the background
981 color, thereby allowing easy differentiation of multiple viewports
982 on the canvas.

983 2. Define the `setViewport()` function:

```
984 setViewport(viewportArray, bound) {  
985     if (bound === undefined) {  
986         bound = this.mViewportBound;  
987     }  
988     // [x, y, width, height]  
989     this.mViewport[0] = viewportArray[0] + bound;  
990     this.mViewport[1] = viewportArray[1] + bound;  
991     this.mViewport[2] = viewportArray[2] - (2 * bound);  
992     this.mViewport[3] = viewportArray[3] - (2 * bound);  
993     this.mScissorBound[0] = viewportArray[0];  
994     this.mScissorBound[1] = viewportArray[1];  
995     this.mScissorBound[2] = viewportArray[2];  
996     this.mScissorBound[3] = viewportArray[3];  
997 }
```

Recall that when setting the camera viewport, you invoke the
 gl.scissor() function to define an area to be cleared and the
 gl.viewport() function to identify the target area for drawing.
 Previously, the scissor and viewport bounds are identical, whereas
 in this case, notice that the actual mViewport bounds are the bound
 number of pixels smaller than the mScissorBound. These settings
 allow the mScissorBound to identify the area to be cleared to
 background color, while the mviewport bounds define the actual
 canvas area for drawing. In this way, the bound number of pixels
 around the viewport will remain the background color.

3. Define the getViewport() function to return the actual bounds that
 are reserved for this camera. In this case, it is the mScissorBound
 instead of the potentially smaller viewport bounds.

```
getViewport() {
  let out = [];
  out[0] = this.mScissorBound[0];
  out[1] = this.mScissorBound[1];
  out[2] = this.mScissorBound[2];
  out[3] = this.mScissorBound[3];
  return out;
}
```

4. Modify the setViewAndCameraMatrix() function to bind scissor
 bounds with mScissorBound instead of the viewport bounds:

```
setViewAndCameraMatrix() {
  let gl = glSys.get();
  ... identical to previous code ...
  // Step A2: set up corresponding scissor area to limit clear area
  gl.scissor(this.mScissorBound[0], // x of bottom-left corner
  this.mScissorBound[1], // y position of bottom-left corner
  this.mScissorBound[2], // width of the area to be drawn
  this.mScissorBound[3]); // height of the area to be drawn
  ... identical to previous code ...
}
```

1031 Testing Multiple Cameras in MyGame

1032 The MyGame level must create multiple cameras, configure them properly, and draw
 1033 each independently. For ease of demonstration, two new Camera objects will be created,
 1034 one to focus on the Hero object and one to focus on the chasing Brain object. As in the
 1035 previous examples, the implementation of the MyGame level is largely identical. In this
 1036 example, some portions of the init(), draw(), and update() functions are modified to
 1037 handle the multiple Camera objects and are highlighted.

- 1038 1. Modify the init() function to define three Camera objects. Both
 1039 the mHeroCam and mBrainCam define a two-pixel boundary for
 1040 their viewports, with the mHeroCam boundary defined to be gray
 1041 (the background color) and with mBrainCam white. Notice the
 1042 mBrainCam object's stiff interpolation setting informing the camera
 1043 interpolation to converge to new values in ten cycles.

```
1044 init() {
1045     // Step A: set up the cameras
1046     this.mCamera = new engine.Camera(
1047         vec2.fromValues(50, 36), // position of the camera
1048         100, // width of camera
1049         [0, 0, 640, 480] // viewport (orgX, orgY, width, height)
1050     );
1051     this.mCamera.setBackgroundColor([0.8, 0.8, 0.8, 1]);
1052     // sets the background to gray
1053
1054     this.mHeroCam = new engine.Camera(
1055         vec2.fromValues(50, 30), // update each cycle to point to hero
1056         20,
1057         [490, 330, 150, 150], // viewport bounds
1058     );
1059     this.mHeroCam.setBackgroundColor([0.5, 0.5, 0.5, 1]);
1060
1061     this.mBrainCam = new engine.Camera(
1062         vec2.fromValues(50, 30), // update each cycle to point to brain
1063         10,
```

```

[0, 330, 150, 150],          1063
    2                         // viewport bounds
);
this.mBrainCam.setBackgroundColor([1, 1, 1, 1]);      1066
this.mBrainCam.configLerp(0.7, 10);                 1067
... identical to previous code ...                  1068
}

```

2. Define a helper function to draw the world that is common to all three cameras:

```

_drawCamera(camera) {           1072
    camera.setViewAndCameraMatrix(); 1073
    this.mBg.draw(camera);          1074
    this.mHero.draw(camera);        1075
    this.mBrain.draw(camera);       1076
    this.mPortal.draw(camera);      1077
    this.mLMinion.draw(camera);     1078
    this.mRMinion.draw(camera);     1079
}

```

3. Modify the MyGame object draw() function to draw all three cameras. Take note of the mMsg object only being drawn to mCamera, the main camera. For this reason, the echo message will appear only in the viewport of the main camera.

```

draw() {                      1085
    // Step A: clear the canvas
    engine.clearCanvas([0.9, 0.9, 0.9, 1.0]); // clear to light gray 1087
    // Step B: Draw with all three cameras
    this._drawCamera(this.mCamera);           1088
    this.mMsg.draw(this.mCamera); // only draw status in main camera 1090
    this._drawCamera(this.mHeroCam);          1091
    this._drawCamera(this.mBrainCam);         1092
}

```

- 1094 4. Modify the update() function to pan the mHeroCam and mBrainCam
1095 with the corresponding objects and to move the mHeroCam
1096 viewport continuously:

1097 **Note** Viewports typically do not change their positions during gameplays. For
1098 testing purposes, the following code moves the mHeroCam viewport continuously
1099 from left to right in the canvas.

```
1100 update() {  
1101     let zoomDelta = 0.05;  
1102     let msg = "L/R: Left or Right Minion; H: Dye; P: Portal];";  
1103     this.mCamera.update(); // for smoother camera movements  
1104     this.mHeroCam.update();  
1105     this.mBrainCam.update();  
1106     ... identical to previous code ...  
1107     // set the hero and brain cams  
1108     this.mHeroCam.panTo(this.mHero.getXform().getXPos(),  
1109         this.mHero.getXform().getYPos());  
1110     this.mBrainCam.panTo(this.mBrain.getXform().getXPos(),  
1111         this.mBrain.getXform().getYPos());  
1112     // Move the hero cam viewport just to show it is possible  
1113     let v = this.mHeroCam.setViewport();  
1114     v[0] += 1;  
1115     if (v[0] > 500) {  
1116         v[0] = 0;  
1117     }  
1118     this.mHeroCam.setViewport(v);  
1119     this.mMsg.setText(msg + this.mChoice);  
1120 }
```

1121 You can now run the project and notice the three different viewports displayed on
1122 the HTML canvas. The two-pixel-wide bounds around the mHeroCam and mBrainCam
1123 viewports allow easy visual parsing of the three views. Observe that the mBrainCam

viewport is drawn on top of the `mHeroCam`. This is because in the `MyGame.draw()` function, the `mBrainCam` is drawn last. The last drawn object always appears on the top. You can move the Hero object to observe that `mHeroCam` follows the hero and experience the smooth interpolated results of panning the camera.

Now try changing the parameters to the `mBrainCam.configLerp()` function to generate smoother interpolated results, such as by setting the stiffness to 0.1 and the duration to 100 cycles. Note how it appears as though the camera is constantly trying to catch up to the Brain object. In this case, the camera needs a stiff interpolation setting to ensure the main object remains in the center of the camera view. For a much more drastic and fun effect, you can try setting `mBrainCam` to have much smoother interpolated results, such as with a stiffness value of 0.01 and a duration of 200 cycles. With these values, the camera can never catch up to the Brain object and will appear as though it is wandering aimlessly around the game world.

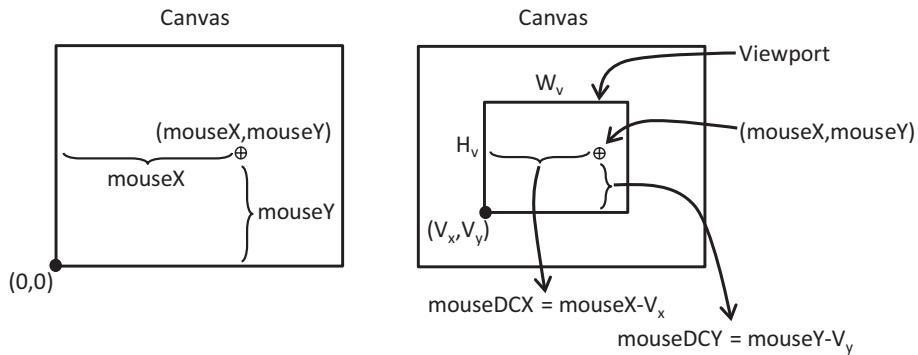
Mouse Input Through Cameras

The mouse is a pointing input device that reports position information in the Canvas Coordinate space. Recall from the discussion in Chapter 3 that the Canvas Coordinate space is simply a measurement of pixel offsets along the x/y axes with respect to the lower-left corner of the canvas. Remember that the game engine defines and works with the WC space where all objects and measurements are specified in WC. For the game engine to work with the reported mouse position, this position must be transformed from Canvas Coordinate space to WC.

The drawing on the left side of Figure 7-10 shows an example of a mouse position located at $(\text{mouseX}, \text{mouseY})$ on the canvas. The drawing on the right side of Figure 7-10 shows that when a viewport with the lower-left corner located at (V_x, V_y) and a dimension of $W_v \times H_v$ is defined within the canvas, the same $(\text{mouseX}, \text{mouseY})$ position can be represented as a position in the viewport as $(\text{mouseDCX}, \text{mouseDCY})$ where

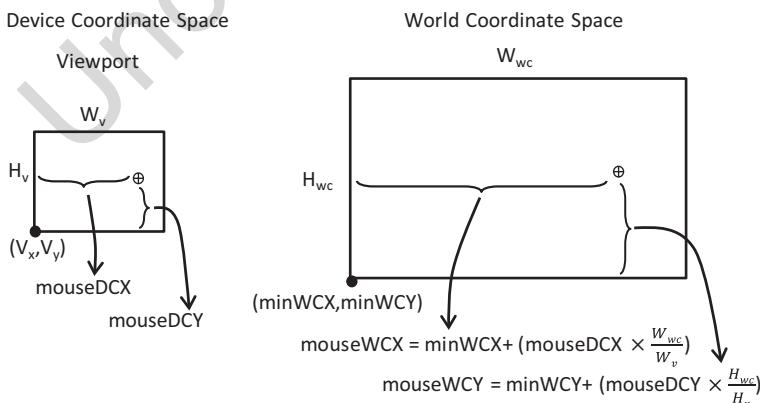
- $\text{mouseDCX} = \text{mouseX} - V_x$
- $\text{mouseDCY} = \text{mouseY} - V_y$

In this way, $(\text{mouseDCX}, \text{mouseDCY})$ is the offset from the (V_x, V_y) , the lower-left corner of the viewport.

**Figure 7-10.** Mouse position on canvas and viewport

1154 The left drawing in Figure 7-11 shows that the Device Coordinate (DC) space defines
 1155 a pixel position within a viewport with offsets measured with respect to the lower-left
 1156 corner of the viewport. For this reason, the DC space is also referred to as the pixel space.
 1157 The computed $(mouseDCX, mouseDCY)$ position is an example of a position in DC space.
 1158 The right drawing in Figure 7-11 shows that this position can be transformed into the
 1159 WC space with the lower-left corner located at $(minWCX, minWCY)$ and a dimension of
 1160 $W_{wc} \times H_{wc}$ according to these formulae:

- 1161 • $mouseWCX = minWCX + \left(mouseDCX \times \frac{W_{wc}}{W_v} \right)$
- 1162 • $mouseWCY = minWCY + \left(mouseDCY \times \frac{H_{wc}}{H_v} \right)$

**Figure 7-11.** Mouse position in viewport DC space and WC space

With the knowledge of how to transform positions from the Canvas Coordinate space to the WC space, it is now possible to implement mouse input support in the game engine.

1163

1164

1165

The Mouse Input Project

1166

This project demonstrates mouse input support in the game engine. You can see an example of this project running in Figure 7-12. The source code to this project is defined in the chapter7/7.5.mouse_input folder.

1167

1168

1169

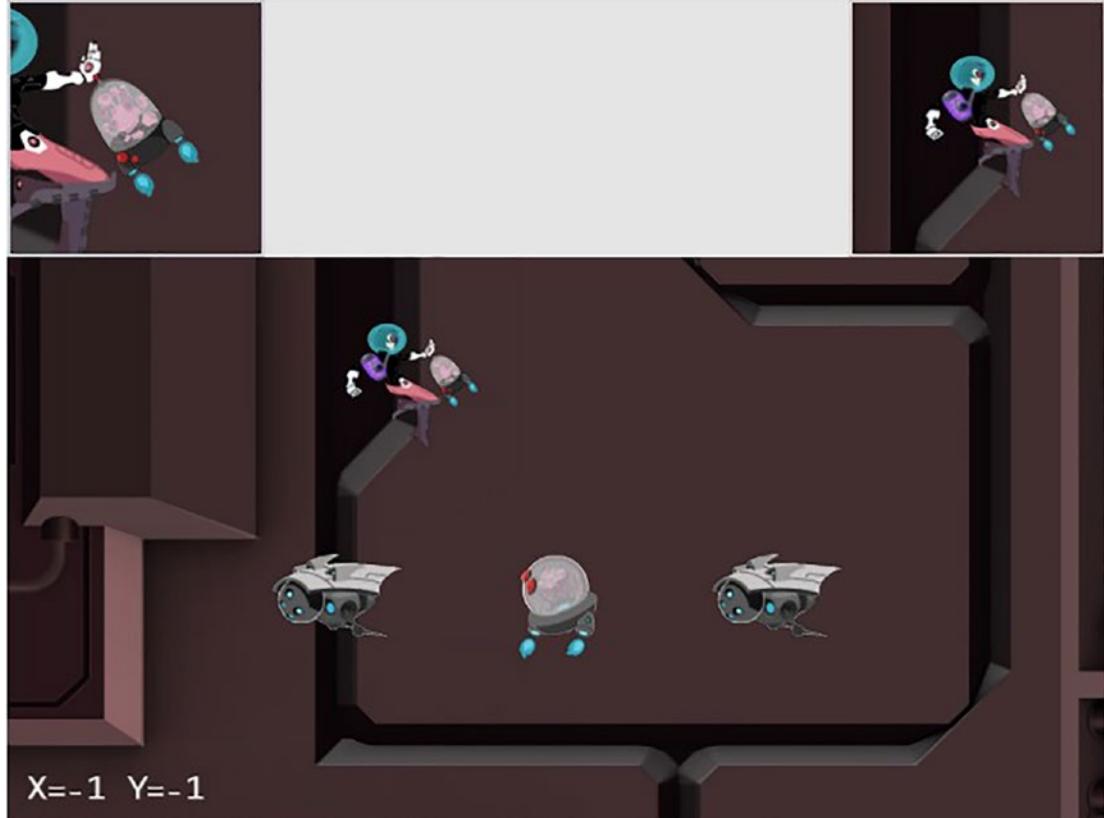


Figure 7-12. Running the Mouse Input project

CHAPTER 7 MANIPULATING THE CAMERA

1170 The new controls of this project are as follows:

- 1171 • **Left mouse button clicked in the main Camera view:** Drags the
1172 Portal object
- 1173 • **Middle mouse button clicked in the HeroCam view:** Drags the Hero
1174 object
- 1175 • **Right/middle mouse button clicked in any view:** Hides/shows the
1176 Portal object

1177 The following controls are identical to the previous project:

- 1178 • **Q key:** Initiates the positional oscillation of the Dye character and the
1179 camera shake effects
- 1180 • **WASD keys:** Move the Dye character (the Hero object) and push the
1181 camera WC bounds
- 1182 • **Arrow keys:** Move the Portal object
- 1183 • **L/R/P/H keys:** Select the in-focus object with L/R keys refocusing the
1184 camera to the Left or Right minion
- 1185 • **N/M and J/K keys:** Zoom into or away from the center of the camera
1186 or the in-focus object

1187 The goals of the project are as follows:

- 1188 • To understand the Canvas Coordinate space to WC space transform
- 1189 • To appreciate the importance of differentiating between viewports for
1190 mouse events
- 1191 • To implement transformation between coordinate spaces
- 1192 • To support and experience working with mouse input

1193 As in previous projects, you can find external resource files in the assets folder.

1194 **Modify index.js to Pass Canvas ID to Input Component**

1195 To receive mouse input information, the input component needs to have access to the
1196 HTML canvas. Edit `index.js` and modify the `init()` function to pass the `htmlCanvasID`
1197 to the input component during initialization.

```

... identical to previous code ... 1198

// general engine utilities 1199
function init(htmlCanvasID) { 1200
    glSys.init(htmlCanvasID); 1201
    vertexBuffer.init(); 1202
    input.init(htmlCanvasID); 1203
    audio.init(); 1204
    shaderResources.init(); 1205
    defaultResources.init(); 1206
} 1207
... identical to previous code ... 1208

```

Implement Mouse Support in input.js

Similar to the keyboard input, you should add mouse support to the input module by editing `input.js`:

1. Edit `input.js` and define the constants to represent the three mouse buttons:

```

// mouse button enums 1214
const eMouseButton = Object.freeze({ 1215
    eLeft: 0, 1216
    eMiddle: 1, 1217
    eRight: 2 1218
}); 1219

```

2. Define the variables to support mouse input. Similar to keyboard input, mouse button states are arrays of three boolean elements, each representing the state of the three mouse buttons.

```

let mCanvas = null; 1223
let mButtonPreviousState = []; 1224
let mIsButtonPressed = []; 1225
let mIsButtonClicked = []; 1226
let mMousePosX = -1; 1227
let mMousePosY = -1; 1228

```

1229 3. Define the mouse movement event handler:

```
1230 function onMouseMove(event) {  
1231   let inside = false;  
1232   let bBox = mCanvas.getBoundingClientRect();  
1233   // In Canvas Space now. Convert via ratio from canvas to client.  
1234   let x = Math.round((event.clientX - bBox.left) *  
1235                       (mCanvas.width / bBox.width));  
1236   let y = Math.round((event.clientY - bBox.top) *  
1237                       (mCanvas.height / bBox.height));  
1238  
1239   if ((x >= 0) && (x < mCanvas.width) &&  
1240       (y >= 0) && (y < mCanvas.height)) {  
1241     mMousPosX = x;  
1242     mMousPosY = mCanvas.height - 1 - y;  
1243     inside = true;  
1244   }  
1245   return inside;  
1246 }
```

1246 Notice that the mouse event handler transforms a raw pixel position into the Canvas
1247 Coordinate space by first checking whether the position is within the bounds of the
1248 canvas and then flipping the y position such that the displacement is measured with
1249 respect to the lower-left corner.

1250 4. Define the mouse button click handler to record the button event:

```
1251 function onMouseDown(event) {  
1252   if (onMouseMove(event)) {  
1253     mIsButtonPressed[event.button] = true;  
1254   }  
1255 }
```

1256 5. Define the mouse button release handler to facilitate the detection
1257 of a mouse button click event. Recall from the keyboard input
1258 discussion in Chapter 4 that in order to detect the button up event,
1259 you should test for a button state that was previously released and
1260 currently clicked. The `mouseUp()` handler records the released
1261 state of a mouse button.

```

function onMouseUp(event) { 1262
    onMouseMove(event); 1263
    mIsButtonPressed[event.button] = false; 1264
} 1265

```

6. Modify the `init()` function to receive the `canvasID` parameter
and initialize mouse event handlers:

```

function init(canvasID) { 1268
    let i; 1269
    // keyboard support 1270
    ... identical to previous code ... 1271
    // Mouse support 1272
    for (i = 0; i < 3; i++) { 1273
        mButtonPreviousState[i] = false; 1274
        mIsButtonPressed[i] = false; 1275
        mIsButtonClicked[i] = false; 1276
    } 1277
    window.addEventListener('mousedown', onMouseDown); 1278
    window.addEventListener('mouseup', onMouseUp); 1279
    window.addEventListener('mousemove', onMouseMove); 1280
    mCanvas = document.getElementById(canvasID); 1281
} 1282

```

7. Modify the `update()` function to process mouse button state
changes in a similar fashion to the keyboard. Take note of the
mouse-click condition that a button that was previously not
clicked is now clicked.

```

function update() { 1287
    let i; 1288
    // update keyboard input state 1289
    ... identical to previous code ... 1290
    // update mouse input state 1291
    for (i = 0; i < 3; i++) { 1292
        mIsButtonClicked[i] = (!mButtonPreviousState[i]) && 1293

```

```

1294           mIsButtonPressed[i];
1295           mButtonPreviousState[i] = mIsButtonPressed[i];
1296       }
1297   }

```

- 1298 8. Define the functions to retrieve mouse position and mouse button
1299 states:

```

1300 function isButtonPressed(button) { return mIsButtonPressed[button]; }
1301 function isButtonClicked(button) { return mIsButtonClicked[button]; }
1302 function getMousePosX() { return mMouePosX; }
1303 function getMousePosY() { return mMouePosY; }

```

- 1304 9. Lastly, remember to export the newly defined functionality:

```

1305 export {
1306     keys, eMouseButton,
1307     init, cleanUp, update,
1308     // keyboard
1309     isKeyClicked, isKeyPressed,
1310     // mouse
1311     isButtonClicked, isButtonPressed, getMousePosX, getMousePosY
1312 }

```

1313 Modify the Camera to Support Viewport to WC Space Transform

1314 The Camera class encapsulates the WC window and viewport and thus should be
1315 responsible for transforming mouse positions. Recall that to maintain readability,
1316 the Camera class source code files are separated according to functionality. The basic
1317 functions of the class are defined in `camera_main.js`. The `camera_manipulate.js` file
1318 imports from `camera_main.js` and defines additional manipulation functions. Lastly, the
1319 `camera.js` file imports from `camera_manipulate.js` to include all the defined functions
1320 and exports the Camera class for external access.

1321 This chaining of imports from subsequent source code files to define additional
1322 functions will continue for the Camera class, with `camera_input.js` defining input
1323 functionality:

1. Create a new file in the `src/engine/cameras` folder and name it `camera_input.js`. This file will expand the `Camera` class by defining the mouse input support functions. Import the following files: 1324
1325
1326
1327
 - `camera_manipulation.js` for all the defined functions for the `Camera` class 1328
1329
 - `eViewport` constants for accessing the viewport array 1330
 - `input` module to access the mouse-related functions 1331

```
import Camera from "./camera_manipulation.js"; 1332
import { eViewport } from "./camera_main.js"; 1333
import * as input from "../input.js"; 1334
... implementation to follow ... 1335
export default Camera; 1336
```

2. Define functions to transform mouse positions from Canvas Coordinate space to the DC space, as illustrated in Figure 7-10: 1337
1338

```
Camera.prototype._mouseDCX = function () { 1339
    return input.getMousePosX() - this.mViewport[eViewport.eOrgX]; 1340
}
Camera.prototype._mouseDCY = function() { 1342
    return input.getMousePosY() - this.mViewport[eViewport.eOrgY]; 1343
} 1344
```

3. Define a function to determine whether a given mouse position is within the viewport bounds of the camera: 1345
1346

```
Camera.prototype.isMouseInViewport = function () { 1347
    let dcX = this._mouseDCX(); 1348
    let dcY = this._mouseDCY(); 1349
    return ((dcX >= 0) && (dcX < this.mViewport[eViewport.eWidth]) && 1350
        (dcY >= 0) && (dcY < this.mViewport[eViewport.eHeight])); 1351
} 1352
```

- 1353 4. Define the functions to transform the mouse position into the WC
 1354 space, as illustrated in Figure 7 11:

```

1355 Camera.prototype.mouseWCX = function () {
1356   let minWCX = this.getWCCenter()[0] - this.getWCWidth() / 2;
1357   return minWCX + (this._mouseDCX() *
1358     (this.getWCWidth() / this.mViewport[eViewport.eWidth]));
1359 }
1360 Camera.prototype.mouseWCY = function () {
1361   let minWCY = this.getWCCenter()[1] - this.getWCHeight() / 2;
1362   return minWCY + (this._mouseDCY() *
1363     (this.getWCHeight() / this.mViewport[eViewport.eHeight]));
1364 }
```

1365 Lastly, update the Camera class access file to properly export the newly defined
 1366 input functionality. This is accomplished by editing the camera.js file and replacing the
 1367 import from camera_manipulate.js with camera_input.js:

```

1368 import Camera from "./camera_input.js";
1369 export default Camera;
```

1370 Testing the Mouse Input in MyGame

1371 The main functionality to be tested includes the abilities to detect which view
 1372 should receive the mouse input, react to mouse button state changes, and transform
 1373 mouse-click pixel positions to the WC space. As in previous examples, the my_game.
 1374 js implementation is largely similar to previous projects. In this case, only the
 1375 update() function contains noteworthy changes that work with the new mouse input
 1376 functionality.

```

1377 update() {
1378   ... identical to previous code ...
1379
1380   msg = "";
1381   // testing the mouse input
1382   if (engine.input.isButtonPressed(engine.input.eMouseButton.eLeft)) {
1383     msg += "[L Down];
```

```

if (this.mCamera.isMouseInViewport()) { 1383
    this.mPortal.getXform().setXPos(this.mCamera.mouseWCX()); 1384
    this.mPortal.getXform().setYPos(this.mCamera.mouseWCY()); 1385
}
} 1386
} 1387

if (engine.input.isButtonPressed(engine.input.eMouseButton.eMiddle)){ 1388
    if (this.mHeroCam.isMouseInViewport()) { 1389
        this.mHero.getXform().setXPos(this.mHeroCam.mouseWCX()); 1390
        this.mHero.getXform().setYPos(this.mHeroCam.mouseWCY()); 1391
    }
} 1392
} 1393

if (engine.input.isButtonClicked(engine.input.eMouseButton.eRight)) { 1394
    this.mPortal.setVisibility(false); 1395
}
} 1396

if (engine.input.isButtonClicked(engine.input.eMouseButton.eMiddle)){ 1397
    this.mPortal.setVisibility(true);
}
} 1399

msg += " X=" + engine.input.getMousePosX() + 1400
      " Y=" + engine.input.getMousePosY(); 1401
this.mMsg.setText(msg); 1402
} 1403

```

The camera.isMouseInViewport() condition is checked when the viewport context is important, as in the case of a left mouse button click in the main camera view or a middle mouse button click in the mHeroCam view. This is in contrast to a right or middle mouse button click for setting the visibility of the Portal object. These two mouse clicks will cause execution no matter where the mouse position is.

You can now run the project and verify the correctness of the transformation to WC space. Click and drag with left mouse button in the main view, or middle mouse button in the mHeroCam view, to observe the accurate movement of the corresponding object as they follow the changing mouse position. The left or middle mouse button drag

1415 actions in the wrong views have no effect on the corresponding objects. For example, a
1416 left mouse button drag in the `mHeroCam` or `mBrainCam` view has no effect on the `Portal`
1417 object. However, notice that the right or middle mouse button click controls the visibility
1418 of the `Portal` object, independent of the location of the mouse pointer. Be aware that the
1419 browser maps the right mouse button click to a default pop-up menu. For this reason,
1420 you should avoid working with right mouse button clicks in your games.

1421 Summary

1422 This chapter was about controlling and interacting with the `Camera` object. You have
1423 learned about the most common camera manipulation operations including clamping,
1424 panning, and zooming. These operations are implemented in the game engine with
1425 utility functions that map the high-level specifications to actual WC window bound
1426 parameters. The sudden, often annoying, and potentially confusing movements from
1427 camera manipulations are mitigated with the introduction of interpolation. Through the
1428 implementation of the camera shake effect, you have discovered that some movements
1429 can be modeled by simple mathematical formulations. You have also experienced the
1430 importance of effective `Camera` object abstraction in supporting multiple camera views.
1431 The last section guided you through the implementation of transforming a mouse
1432 position from the Canvas Coordinate space to the WC space.

1433 In Chapter 5, you found out how to represent and draw an object with a visually
1434 appealing image and control the animation of this object. In Chapter 6, you read
1435 about how to define an abstraction to encapsulate the behaviors of an object and the
1436 fundamental support required to detect collisions between objects. This chapter was
1437 about the “directing” of these objects: what should be visible, where the focus should be,
1438 how much of the world to show, how to ensure smooth transition between foci, and how
1439 to receive input from the mouse. With these capabilities, you now have a well-rounded
1440 game engine framework that can represent and draw objects, model and manage the
1441 behaviors of the objects, and control how, where, and what objects are shown.

1442 The following chapters will continue to examine object appearance and behavior at
1443 more advanced levels, including creating lighting and illumination effects in a 2D world
1444 and simulating and integrating behaviors based on simple classical mechanics.

Game Design Considerations

1445

You've learned the basics of object interaction, and it's a good time to start thinking about creating your first simple game mechanic and experimenting with the logical conditions and rules that constitute well-formed gameplay experiences. Many designers approach game creation from the top down (meaning they start with an idea for an implementation of a specific genre like a real-time strategy, tower defense, or role-playing game), which we might expect in an industry like video games where the creators typically spend quite a bit of time as content consumers before transitioning into content makers. Game studios often reinforce this top-down design approach, assigning new staff to work under seasoned leads to learn best practices for whatever genre that particular studio works in. This has proven effective for training designers who can competently iterate on known genres, but it's not always the best path to develop well-rounded creators who can design entirely new systems and mechanics from the ground up.

1446

1447

1448

1449

1450

1451

1452

1453

1454

1455

1456

1457

1458

1459

1460

1461

1462

1463

1464

1465

1466

1467

1468

1469

1470

1471

1472

1473

The aforementioned might lead us to ask, "What makes gameplay well formed?" At a fundamental level, a game is an interactive experience where rules must be learned and applied to achieve a specified outcome; all games must meet this minimum criterion, including card, board, physical, video, and other game types. Taking it a step further, a good game is an interactive experience with rules people enjoy learning and applying to achieve an outcome they feel invested in. There's quite a bit to unpack in this brief definition, of course, but as a general rule, players will enjoy a game more when the rules are discoverable, consistent, and make logical sense and when the outcome feels like a satisfactory reward for mastering those rules. This definition applies to both individual game mechanics and entire game experiences. To use a metaphor, it can be helpful to think of game designs as being built with letters (interactions) that form words (mechanics) that form sentences (levels) that ultimately form readable content (genres). Most new designers attempt to write novels before they know the alphabet, and everyone has played games where the mechanics and levels felt at best like sentences written with poor grammar and at worst like unsatisfying, random jumbles of unintelligible letters.

1474

1475

1476

1477

1478

1479

Over the next several chapters, you'll learn about more advanced features in 2D game engines, including simulations of illumination and physical behaviors. You'll also be introduced to a set of design techniques enabling you to deliver a complete and well-formed game level, integrating these techniques and utilizing more of the nine elements of game design discussed in Chapter 4 in an intentional way and working from the ground up to deliver a unified experience. In the earliest stages of design exploration,

1480 it's often helpful to focus only on creating and refining the basic game mechanics and
1481 interaction model; at this stage, try to avoid thinking about setting, meta-game, systems
1482 design, and the like (these will be folded into the design as it progresses).

1483 The first design technique we'll explore is a simple exercise that allows you to start
1484 learning the game design alphabet: an "escape the room" scenario with one simple
1485 mechanic, where you must accomplish a task in order to unlock a door and claim a
1486 reward. This exercise will help you develop insight into creating well-formed and logical
1487 rules that are discoverable and consistent, which is much easier to accomplish when the
1488 tasks are separated into basic interactions. You've already explored the beginnings of
1489 potential rule-based scenarios in earlier projects: recall the Keyboard Support project from
1490 Chapter 4, which suggested you might have players move a smaller square completely into
1491 the boundary of a larger square in order to trigger some kind of behavior. How might that
1492 single interaction (or "letter of the game alphabet") combine to form a game mechanic (or
1493 "word") that makes sense? Figure 7-13 sets the stage for the locked room puzzle.

this figure will be printed in b/w

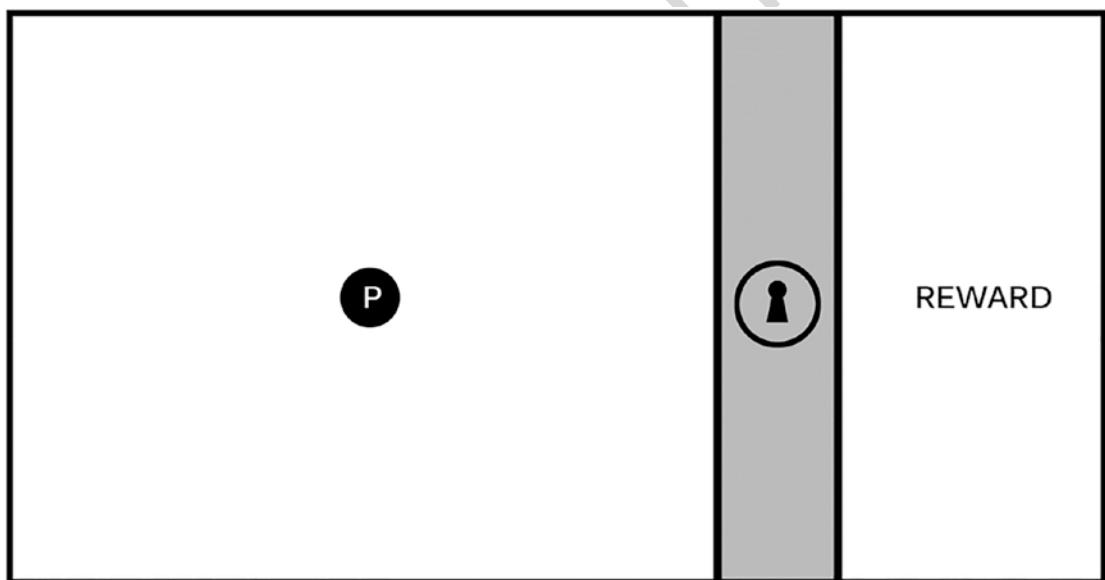
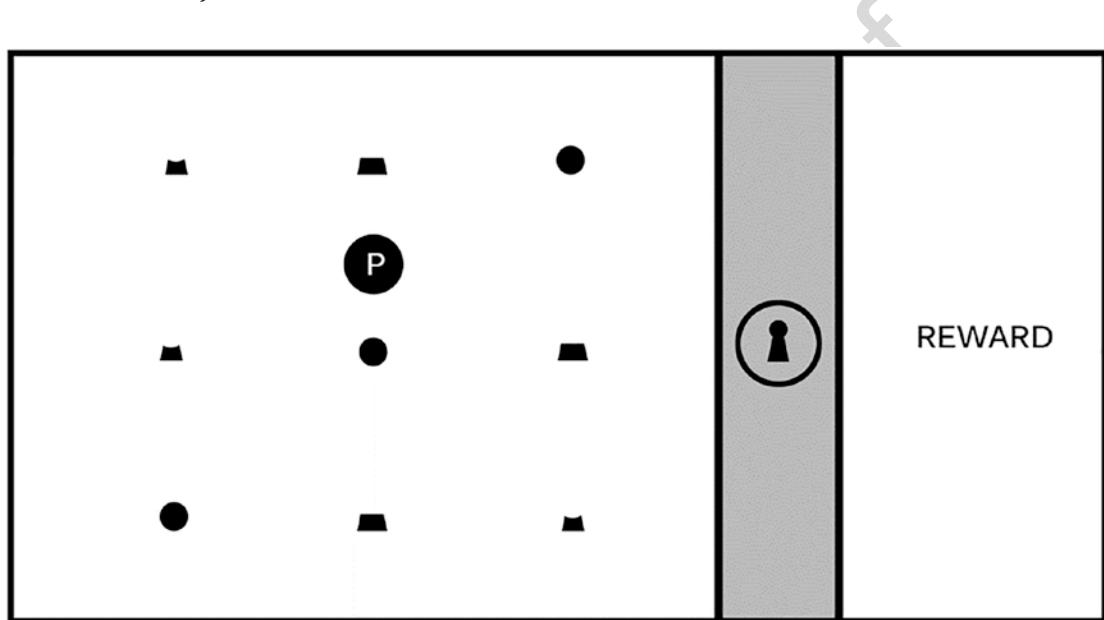


Figure 7-13. The image represents a single game screen divided into three areas. A playable area on the left with a hero character (the circle marked with a P), an impassable barrier marked with a lock icon, and a reward area on the right

1494 The screen represented in Figure 7-13 is a useful starting place when exploring new
1495 mechanics. The goal for this exercise is to create one logical challenge that a player must
1496 complete to unlock the barrier and reach the reward. The specific nature of the task can

be based on a wide range of elemental mechanics: it might involve jumping or shooting,
 puzzle solving, narrative situations, or the like. The key is to keep this first iteration
 simple (this first challenge should have a limited number of components contributing to
 the solution) and discoverable (players must be able to experiment and learn the rules
 of engagement so they can intentionally solve the challenge). You'll add complexity and
 interest to the mechanic in later iterations, and you'll see how elemental mechanics can
 be evolved to support many kinds of game types.

Figure 7-14 sets the stage for a logical relationship mechanic where players must
 interact with objects in the environment to learn the rules.



this figure will be printed in b/w

Figure 7-14. The game screen is populated with an assortment of individual objects

It's not immediately apparent just by looking at Figure 7-14 what the player needs
 to do to unlock the barrier, so they must experiment in order to learn the rules by which
 the game world operates; it's this experimentation that forms the core element of a game
 mechanic driving players forward through the level, and the mechanic will be more or
 less satisfying based on the discoverability and logical consistency of its rules. In this
 example, imagine that as the player moves around the game screen, they notice that
 when the hero character interacts with an object, it always "activates" with a highlight, as
 shown in Figure 7-15, and sometimes causes a section of the lock icon and one-third of
 the ring around the lock icon to glow. Some shapes, however, will not cause the lock and
 ring to glow when activated, as shown in Figure 7-16.

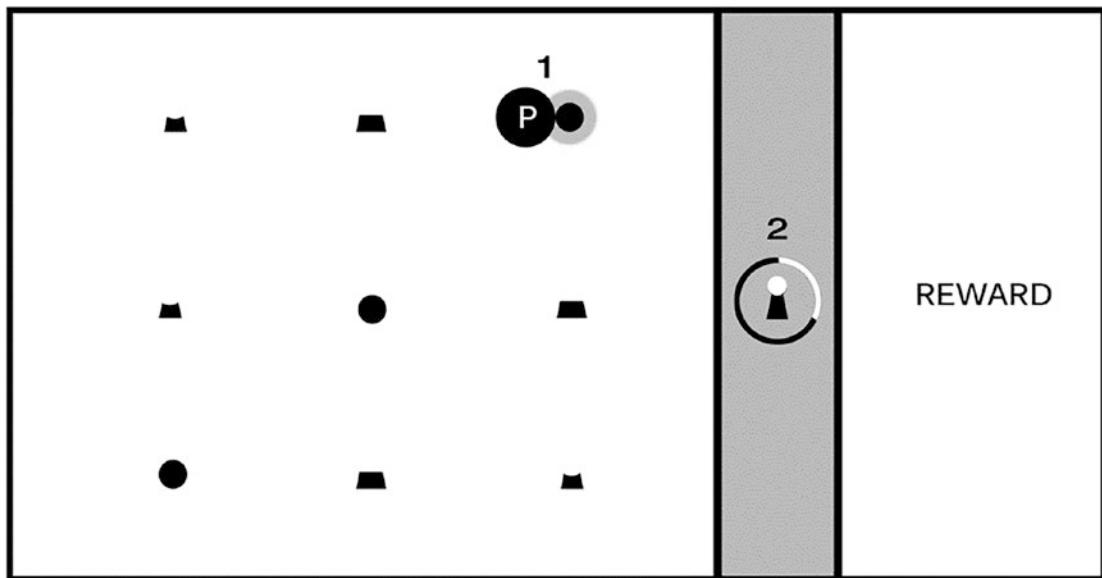


Figure 7-15. As the player moves the hero character around the game screen, the shapes “activate” with a highlight (#1); activating certain shapes causes a section of the lock and one-third of the surrounding ring to glow (#2)

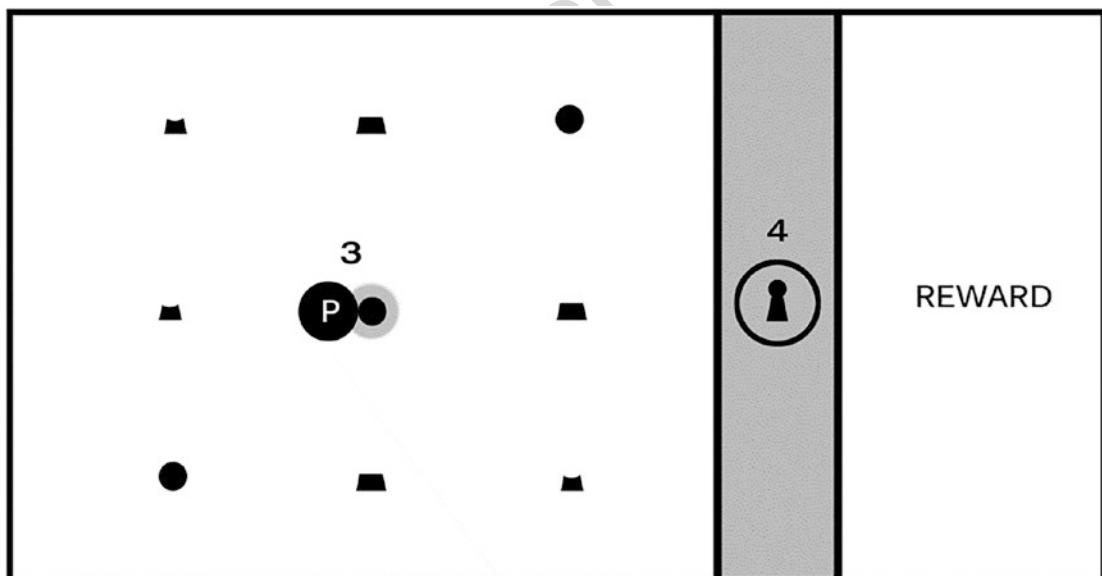


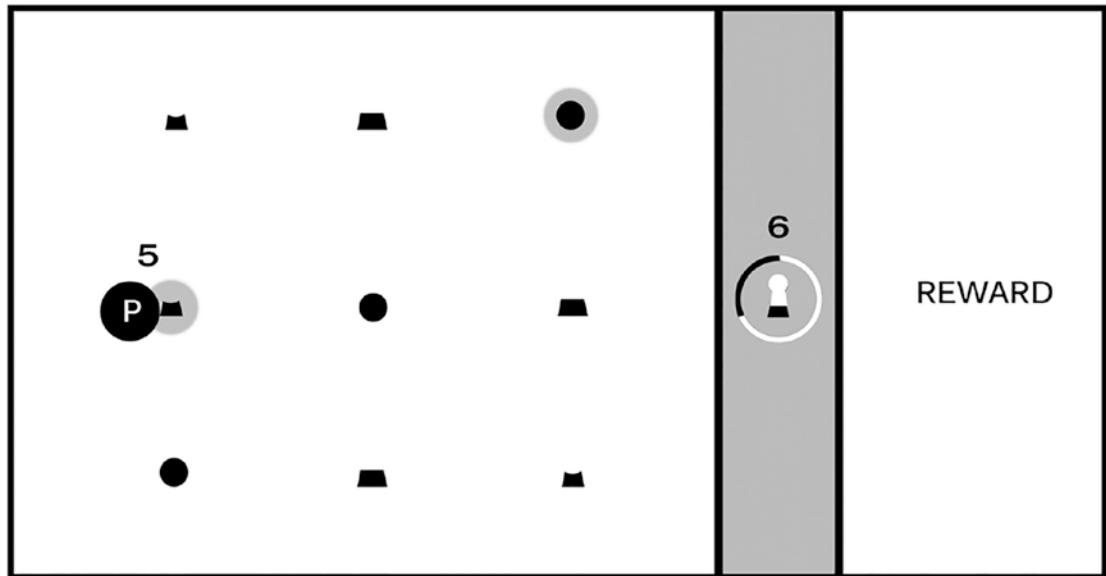
Figure 7-16. Activating some shapes (#3) will not cause the lock and ring to glow (#4)

Astute players will learn the rules for this puzzle fairly quickly. Can you guess what they might be just from looking at Figures 7-15 and 7-16? If you’re feeling stuck, Figure 7-17 should provide enough information to solve the puzzle.

1516

1517

1518



this figure will be printed in b/w

Figure 7-17. After the first object was activated (the circle in the upper right) and caused the top section of the lock and first third of the ring to glow, as shown in Figure 7-15, the second object in the correct sequence (#5) caused the middle section of the lock and second third of the ring to glow (#6)

You (and players) should now have all required clues to learn the rules of this mechanic and solve the puzzle. There are three shapes the player can interact with and only one instance of each shape per row; the shapes are representations of the top, middle, and bottom of the lock icon, and as shown in Figure 7-15, activating the circle shape caused the corresponding section of the lock to glow. Figure 7-16, however, did not cause the corresponding section of the lock to glow, and the difference is the “hook” for this mechanic: sections of the lock must be activated in the correct relative position: top in the top row, middle in the middle row, bottom on the bottom (you might also choose to require that players activate them in the correct sequence starting with the top section, although that requirement is not discoverable just from looking at Figures 7-15 to 7-17).

1519

1520

1521

1522

1523

1524

1525

1526

1527

1528

1529

CHAPTER 7 MANIPULATING THE CAMERA

1530 Congratulations, you've now created a well-formed and logically consistent (if
1531 simple) puzzle, with all of the elements needed to build a larger and more ambitious
1532 level! This unlocking sequence is a game mechanic without narrative context: the
1533 game screen is intentionally devoid of game setting, visual style, or genre alignment
1534 at this stage of design because we don't want to burden our exploration yet with any
1535 preconceived expectations. It can benefit you as a designer to spend time exploring
1536 game mechanics in their purest form before adding higher-level game elements like
1537 narrative and genre, and you'll likely be surprised at the unexpected directions, these
1538 simple mechanics will take you as you build them out.

1539 Simple mechanics like the one in this example can be described as "complete a
1540 multistage task in the correct sequence to achieve a goal" and are featured in many kinds
1541 of games; any game that requires players to collect parts of an object and combine them
1542 in an inventory to complete a challenge, for example, utilizes this mechanic. Individual
1543 mechanics can also be combined with other mechanics and game features to form
1544 compound elements that add complexity and flavor to your game experience.

1545 The camera exercises in this chapter provide good examples for how you might add
1546 interest to a single mechanic; the simple Camera Manipulations project, for example,
1547 demonstrates a method for advancing game action. Imagine in the previous example
1548 that after a player receives a reward for unlocking the barrier, they move the hero object
1549 to the right side of the screen and advance to a new "room" or area. Now imagine how
1550 gameplay would change if the camera advanced the screen at a fixed rate when the
1551 level started; the addition of autoscrolling changes this mechanic considerably because
1552 the player must solve the puzzle and unlock the barrier before the advancing barrier
1553 pushes the player off the screen. The first instance creates a leisurely puzzle-solving
1554 game experience, while the latter increases the tension considerably by giving the player
1555 a limited amount of time to complete each screen. In an autoscrolling implementation,
1556 how might you lay out the game screen to ensure the player had sufficient time to learn
1557 the rules and solve the puzzle?

1558 The Multiple Cameras project can be especially useful as a mini-map that provides
1559 information about places in the game world not currently displayed on the game screen;
1560 in the case of the previous exercise, imagine that the locked barrier appeared somewhere
1561 else in the game world other than the player's current screen and that a secondary
1562 camera acting as a mini-map displayed a zoomed out view of the entire game world
1563 map. As the game designer, you might want to let the player know when they complete
1564 a task that allows them to advance and provide information about where they need to

go next, so in this case, you might flash a beacon on the mini-map calling attention to the barrier that just unlocked and showing the player where to go. In the context of our “game design is like a written language” metaphor, adding additional elements like camera behavior to enhance or extend a simple mechanic is one way to begin forming “adjectives” that add interest to the basic nouns and verbs we’ve been creating from the letters in the game design alphabet.

A game designer’s primary challenge is typically to create scenarios that require clever experimentation while maintaining logical consistency; it’s perfectly fine to frustrate players by creating devious scenarios requiring creative problem solving (we call this “good” frustration), but it’s generally considered poor design to frustrate players by creating scenarios that are logically inconsistent and make players feel that they succeeded in a challenge only by random luck (“bad” frustration). Think back to the games you’ve played that have resulted in bad frustration: where did they go wrong, and what might the designers have done to improve the experience?

The locked room scenario is a useful design tool because it forces you to construct basic mechanics, but you might be surprised at the variety of scenarios that can result from this exercise. Try a few different approaches to the locked room puzzle and see where the design process takes you, but keep it simple. For now, stay focused on one-step events to unlock the room that require players to learn only one rule. You’ll revisit this exercise in the next chapter and begin creating more ambitious mechanics that add additional challenges.

Author Queries

Chapter No.: 7 0005244100

Queries	Details Required	Author's Response
AU1	Please check sentence starting “In effect, approximating...” for completeness.	
AU2	Please check if edit to sentence starting “Such formulation can...” is okay.	
AU3	Please check if edit to sentence starting “A shaking effect...” is okay.	

Uncorrected Proof

CHAPTER 8

1

Implementing Illumination and Shadow

2

3

After completing this chapter, you will be able to

- Understand the parameters of simple illumination models
- Define infrastructure supports for working with multiple light sources
- Understand the basics of diffuse reflection and normal mapping
- Understand the basics of specular reflection and the Phong illumination model
- Implement GLSL shaders to simulate diffuse and specular reflection and the Phong illumination model
- Create and manipulate point, directional, and spotlights
- Simulate shadows with the WebGL stencil buffer

4

5

6

7

8

9

10

11

12

13

Introduction

14

Up until now in the game engine, you have implemented mostly functional modules in order to provide the fundamentals required for many types of 2D games. That is, you have developed engine components and utility classes that are designed to support the actual gameplay directly. This is a great approach because it allows you to systematically expand the capabilities of your engine to allow more types of games and gameplay. For instance, with the topics covered thus far, you can implement a variety of different games including puzzle games, top-down space shooters, and even simple platform games.

15

16

17

18

19

20

21

An illumination model, or a lighting model, is a mathematical formulation that describes the color and brightness of a scene based on approximating light energy reflecting off the surfaces in the scene. In this chapter, you will implement an illumination model that indirectly affects the types of gameplay your game engine can support and the visual fidelity that can be achieved. This is because illumination support from a game engine can be more than a simple aesthetic effect. When applied creatively, illumination can enhance gameplay or provide a dramatic setting for your game. For example, you could have a scene with a torch light that illuminates an otherwise dark pathway for the hero, with the torch flickering to communicate a sense of unease or danger to the player. Additionally, while the lighting model is based on light behaviors in the physical world, in your game implementation, the lighting model allows surreal or physically impossible settings, such as an oversaturated light source that displays bright or iridescent colors or even a negative light source that absorbs visible energy around it.

When implementing illumination models commonly present in game engines, you will need to venture into concepts in 3D space to properly simulate light. As such, the third dimension, or depth, must be specified for the light sources to cast light energy upon the game objects, or the Renderable objects, which are flat 2D geometries. Once you consider concepts in 3D, the task of implementing a lighting model becomes much more straightforward, and you can apply knowledge from computer graphics to properly illuminate a scene.

A simplified variation of the Phong illumination model that caters specifically to the 2D aspect of your game engine will be derived and implemented. However, the principles of the illumination model remain the same. If you desire more information or a further in-depth analysis of the Phong illumination model, please refer to the recommended reference books from Chapter 1.

Overview of Illumination and GLSL Implementation

In general, an illumination model is one or a set of mathematical equations describing how humans observe the interaction of light with object materials in the environment. As you can imagine, an accurate illumination model that is based on the physical world can be highly complex and computationally expensive. The Phong illumination model captures many of the interesting aspects of light/material interactions with a relatively

simple equation that can be implemented efficiently. The projects in this chapter guide you in understanding the fundamental elements of the Phong illumination model in the following order:

- **Ambient light:** Reviews the effects of lights in the absence of explicit light sources 53
- **Light source:** Examines the effect of illumination from a single light source 54
- **Multiple light sources:** Develop game engine infrastructure to support multiple light sources 55
- **Diffuse reflection and normal maps:** Simulate light reflection from matte or diffuse surfaces 60
- **Specular light and material:** Models light reflecting off shinny surfaces and reaching the camera 61
- **Light source types:** Introduce illumination based on different types of light sources 62
- **Shadow:** Approximates the results from light being blocked 63

Together, the projects in this chapter build a powerful tool for adding visual intricacy into your games. In order to properly render and display the results of illumination, the associated computation must be performed for each affected pixel. Recall that the GLSL fragment shader is responsible for computing the color of each pixel. In this way, each fundamental element of the Phong illumination model can be implemented as additional functionality to existing or new GLSL fragment shaders. In all projects of this chapter, you will begin by working with the GLSL fragment shader.

Ambient Light

Ambient light, often referred to as background light, allows you to see objects in the environment when there are no explicit light sources. For example, in the dark of night, you can see objects in a room even though all lights are switched off. In the real world, light coming from the window, from underneath the door, or from the background illuminates the room for you. A realistic simulation of the background light illumination, often referred to as indirect illumination, is algorithmically complex and can be

83 computationally expensive. Instead, in computer graphics and most 2D games, ambient
84 lighting is approximated by adding a constant color, or the ambient light, to every object
85 within the current scene or world. It is important to note that while ambient lighting can
86 provide the desired results, it is only a rough approximation and does not mimic real-
87 world indirect lighting.

88 The Global Ambient Project

89 This project demonstrates how to implement ambient lighting within your scenes
90 by defining a global ambient color and a global ambient intensity for drawing each
91 Renderable object. You can see an example of this project running in Figure 8-1. The
92 source code of this project is located in the chapter8/8.1.global_ambient folder.

this figure will be printed in b/w

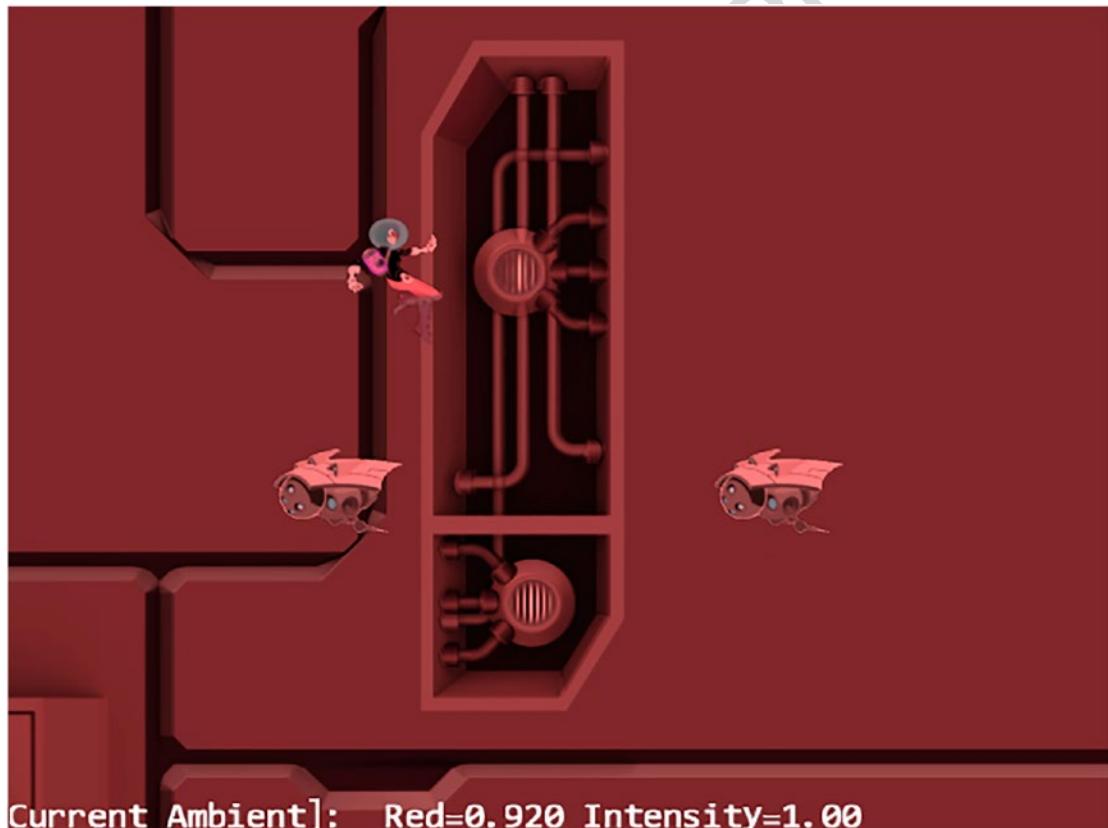


Figure 8-1. Running the Global Ambient project

The controls of the project are as follows:	93
• Left mouse button: Increases the global red ambient	94
• Middle mouse button: Decreases the global red ambient	95
• Left-/right-arrow keys: Decrease/increase the global ambient intensity	96
	97

The goals of the project are as follows:	98
--	----

- To experience the effects of ambient lighting 99
- To understand how to implement a simple global ambient illumination across a scene 100
- To refamiliarize yourself with the SimpleShader/Renderable pair structure to interface to GLSL shaders and the game engine 102

You can find the following external resources in the assets folder. The fonts folder contains the default system fonts and two texture images: `minion_sprite.png`, which defines the sprite elements for the hero and the minions, and `bg.png`, which defines the background.

Modifying the GLSL Shaders

A good place to start when implementing new shaders or shading functionality for the game engine is the GLSL shader. The GLSL code creation or modification allows you to implement the actual functional details, which, in turn, serves as the requirements for expanding the engine. For example, in this project, you will begin by adding ambient lighting functionality to all existing GLSL shaders. The support for this newly added functionality then becomes the requirements that guide the modifications to the rest of the engine. You will observe this implementation pattern for all of the examples in this chapter. Thus, to begin, integrate the global ambient into your `simple_fs.glsl`.

1. Modify the fragment shader `simple_fs.glsl` by defining two new uniform variables `uGlobalAmbientColor` and `uGlobalAmbientIntensity` and multiplying these variables with the `uPixelColor` when computing the final color for each pixel: 117

```

121 precision mediump float;
122 // Color of pixel
123 uniform vec4 uPixelColor;
124 uniform vec4 uGlobalAmbientColor; // this is shared globally
125 uniform float uGlobalAmbientIntensity; // this is shared globally
126 void main(void) {
127     // for every pixel called sets to the user specified color
128     gl_FragColor = uPixelColor * uGlobalAmbientIntensity *
129         uGlobalAmbientColor;
130 }
```

- 131 2. Similarly, modify the texture fragment shader `texture_fs.gsls`
 132 by adding the `uniform` variables `uGlobalAmbientColor` and
 133 `uGlobalAmbientIntensity`. Multiply these two variables with the
 134 sampled texture color to create the background lighting effect.

```

135 uniform sampler2D uSampler;
136 // Color of pixel
137 uniform vec4 uPixelColor;
138 uniform vec4 uGlobalAmbientColor; // this is shared globally
139 uniform float uGlobalAmbientIntensity; // this is shared globally
140 varying vec2 vTexCoord;
141 void main(void) {
142     // texel color look up based on interpolated UV value in vTexCoord
143     vec4 c = texture2D(uSampler, vec2(vTexCoord.s, vTexCoord.t));
144     c = c * uGlobalAmbientIntensity * uGlobalAmbientColor;
145     ... identical to previous code ...
146 }
```

147 Defining as Global Shared Resources

148 Ambient lighting affects the entire scene, and thus, the associated variables must be
 149 global and shared. In this case, the two variables, a color (ambient color) and a floating
 150 point (intensity of the color), should be globally accessible to the rest of the engine

and to the clients. The `defaultResources` module is perfectly suited for this purpose. 151
 Edit the `src/engine/resources/default_resources.js` file and define the color and 152
 intensity variables, their corresponding getters and setters, and remember to export the 153
 functionality. 154

```

import * as font from "./font.js";
import * as map from "../core/resource_map.js";

// Global Ambient color
let mGlobalAmbientColor = [0.3, 0.3, 0.3, 1];
let mGlobalAmbientIntensity = 1;
function getGlobalAmbientIntensity() { return mGlobalAmbientIntensity; }
function setGlobalAmbientIntensity(v) { mGlobalAmbientIntensity = v; }
function getGlobalAmbientColor() { return mGlobalAmbientColor; }
function setGlobalAmbientColor(v) {
    mGlobalAmbientColor = vec4.fromValues(v[0], v[1], v[2], v[3]);
}

... identical to previous code ...

export {
    init, cleanUp,
    // default system font name: this is guaranteed to be loaded
    getDefaultFontName,
    // Global ambient: intensity and color
    getGlobalAmbientColor, setGlobalAmbientColor,
    getGlobalAmbientIntensity, setGlobalAmbientIntensity
}

```

Modifying SimpleShader

With global ambient color and intensity now implemented in the GLSL shaders, you 174
 need to modify the rest of the game engine to support the newly defined functionality. 175
 Recall that `simple_fs.gls1` is referenced by the `SimpleShader` class and that `texture_` 176
`fs.gls1` is referenced by the `TextureShader` class. Since `TextureShader` is a subclass 177
of `SimpleShader`, the newly defined GLSL functionality in `texture_fs.gls1` will be 178
supported with appropriate `SimpleShader` modifications. 179

- 180 1. Modify the `simple_shader.js` file in the `src/engine/shaders`
181 folder to import from the `defaultResources` module for accessing
182 the global ambient light effects variables:

183 `import * as defaultResources from "../resources/default_resources.js";`

- 184 2. Define two new instance variables in the constructor for storing
185 the references or locations of the ambient color and intensity
186 variables in the GLSL shader:

187 `this.mGlobalAmbientColorRef = null;`

188 `this.mGlobalAmbientIntensityRef = null;`

- 189 3. In step E of the `SimpleShader` constructor, call the WebGL
190 `getUniformLocation()` function to query and store the locations
191 of the uniform variables for ambient color and intensity in the
192 GLSL shader:

193 `// Step E: Gets references to the uniform variables`

194 `this.mPixelColorRef = gl.getUniformLocation(`

195 `this.mCompiledShader, "uPixelColor");`

196 `this.mModelMatrixRef = gl.getUniformLocation(`

197 `this.mCompiledShader, "uModelXformMatrix");`

198 `this.mCameraMatrixRef = gl.getUniformLocation(`

199 `this.mCompiledShader, "uCameraXformMatrix");`

200 `this.mGlobalAmbientColorRef = gl.getUniformLocation(`

201 `this.mCompiledShader, "uGlobalAmbientColor");`

202 `this.mGlobalAmbientIntensityRef = gl.getUniformLocation(`

203 `this.mCompiledShader, "uGlobalAmbientIntensity");`

- 204 4. In the `activate()` function, retrieve the global ambient color and
205 intensity values from the `defaultResources` module and pass to
206 the corresponding uniform variables in the GLSL shader. Notice
207 the data type-specific WebGL function names for setting uniform
208 variables. As you can probably guess, `uniform4fv` corresponds
209 to `vec4`, which is the color storage, and to `uniform1f`, which
210 corresponds to a float and is the intensity.

```

activate(pixelColor, trsMatrix, cameraMatrix) {           211
    let gl = glSys.get();                                212
    ... identical to previous code ...                  213
    // load uniforms                               214
    gl.uniformMatrix4fv(this.mCameraMatrixRef, false, cameraMatrix); 215
    gl.uniform4fv(this.mGlobalAmbientColorRef,          216
        defaultResources.getGlobalAmbientColor());      217
    gl.uniform1f(this.mGlobalAmbientIntensityRef,     218
        defaultResources.getGlobalAmbientIntensity()); 219
}

```

Testing the Ambient Illumination

You can now define the MyGame class to verify the correctness of the newly defined ambient lighting effect. In anticipation of upcoming complexities in testing, the MyGame class source code will be separated into multiple files similar to your experience working with the Camera class in Chapter 7. All files implementing MyGame will have a name that begins with my_game and ends with an indication of the associated functionality defined in the file. For example, in later examples, my_game_light.js indicates that the file implements light source-related logic. For this project, similar to the Camera class naming scheme, the basic functionality of MyGame class will be implemented in my_game_main.js, and the access will be via the file my_game.js.

1. Create the MyGame class access file in src/my_game. For now, the MyGame functionality should be imported from the basic class implementation file, my_game_main.js. With full access to the MyGame class, it is convenient to define the webpage onload() function in this file.

```

import engine from "../engine/index.js";                236
import MyGame from "./my_game_main.js";            237

window.onload = function () {                          238
    engine.init("GLCanvas");                         239
    let myGame = new MyGame();                      240
    myGame.start();                                241
}

```

CHAPTER 8 IMPLEMENTING ILLUMINATION AND SHADOW

```
243     2. Create my_game_main.js; import from the engine access file,  
244         index.js, and from Hero and Minion; and remember to export  
245         the MyGame functionality. Now, as in all previous cases, define  
246         MyGame as a subclass of engine.Scene with the constructor that  
247         initializes instance variables to null.  
  
248 import engine from "../engine/index.js";  
  
249 // user stuff  
250 import Hero from "./objects/hero.js";  
251 import Minion from "./objects/minion.js";  
  
252 class MyGame extends engine.Scene {  
253     constructor() {  
254         super();  
255         this.kMinionSprite = "assets/minion_sprite.png";  
256         this.kBg = "assets/bg.png";  
  
257         // The camera to view the scene  
258         this.mCamera = null;  
259         this.mBg = null;  
  
260         this.mMsg = null;  
  
261         // the hero and the support objects  
262         this.mHero = null;  
263         this.mLMinion = null;  
264         this.mRMinion = null;  
265     }  
266     ... implementation to follow ...  
267 }  
268 export default MyGame;
```

3. Load and unload the background and the minions:

269

```
load() {  
    engine.texture.load(this.kMinionSprite);  
    engine.texture.load(this.kBg);  
}  
  
unload() {  
    engine.texture.unload(this.kMinionSprite);  
    engine.texture.unload(this.kBg);  
}
```

270

271

272

273

4. Initialize the camera and scene objects with corresponding values
to ensure proper scene view at startup. Note the simple elements
in the scene, the camera, the large background, a Hero, the left
and right Minion objects, and the status message.

274

275

276

277

```
init() {  
    // Step A: set up the cameras  
    this.mCamera = new engine.Camera(  
        vec2.fromValues(50, 37.5), // position of the camera  
        100, // width of camera  
        [0, 0, 640, 480] // viewport (orgX, orgY, width, height)  
    );  
    this.mCamera.setBackgroundColor([0.8, 0.8, 0.8, 1]);  
    // sets the background to gray  
  
    let bgR = new engine.SpriteRenderable(this.kBg);  
    bgR.setElementPixelPositions(0, 1900, 0, 1000);  
    bgR.getXform().setSize(190, 100);  
    bgR.getXform().setPosition(50, 35);  
    this.mBg = new engine.GameObject(bgR);  
  
    this.mHero = new Hero(this.kMinionSprite);  
  
    this.mLMinion = new Minion(this.kMinionSprite, 30, 30);  
    this.mRMinion = new Minion(this.kMinionSprite, 70, 30);
```

278

279

280

281

282

283

284

285

286

287

288

289

290

291

292

293

294

295

296

297

298

CHAPTER 8 IMPLEMENTING ILLUMINATION AND SHADOW

```
299     this.mMsg = new engine.FontRenderable("Status Message");
300     this.mMsg.setColor([1, 1, 1, 1]);
301     this.mMsg.getXform().setPosition(1, 2);
302     this.mMsg.setTextHeight(3);
303 }

304     5. Define the draw() function. As always, draw the status message
305         last such that it will not be covered by any other object.

306     draw() {
307         // Clear the canvas
308         engine.clearCanvas([0.9, 0.9, 0.9, 1.0]); // clear to light gray
309
310         // Set up the camera and draw
311         this.mCamera.setViewAndCameraMatrix();
312         this.mBg.draw(this.mCamera);
313         this.mHero.draw(this.mCamera);
314         this.mLMinion.draw(this.mCamera);
315         this.mRMinion.draw(this.mCamera);
316
317         this.mMsg.draw(this.mCamera);    // draw last
318     }

319     6. Lastly, implement the update() function to update all objects and
320         receive controls over global ambient color and intensity:
321
322     update() {
323         let deltaAmbient = 0.01;
324         let msg = "Current Ambient]: ";
325
326         this.mCamera.update(); // ensure proper interpolated movement
327         this.mLMinion.update(); // ensure sprite animation
328         this.mRMinion.update();
329         this.mHero.update(); // allow keyboard control to move
330         this.mCamera.panWith(this.mHero.getXform(), 0.8);
331
332         let v = engine.defaultResources.getGlobalAmbientColor();
333         if (engine.input.isButtonPressed(engine.input.eMouseButton.eLeft))
334             v[0] += deltaAmbient;
```

```

if (engine.input.isButtonPressed(engine.input.eMouseButton.eMiddle))      330
    v[0] -= deltaAmbient;                                                 331

if (engine.input.isKeyPressed(engine.input.keys.Left))                      332
    engine.defaultResources.setGlobalAmbientIntensity(                  333
        engine.defaultResources.getGlobalAmbientIntensity() -            334
            deltaAmbient);                                              335

if (engine.input.isKeyPressed(engine.input.keys.Right))                     336
    engine.defaultResources.setGlobalAmbientIntensity(                  337
        engine.defaultResources.getGlobalAmbientIntensity() +            338
            deltaAmbient);                                              339

msg += " Red=" + v[0].toPrecision(3) + " Intensity=" +                   340
    engine.defaultResources.getGlobalAmbientIntensity().toPrecision(3); 341
this.mMsg.setText(msg);                                                 342

}

```

Observations

You can now run the project and observe the results. Notice that the initial scene is dark. This is because the RGB values for the global ambient color were all initialized to 0.3. Since the ambient color is multiplied by the color sampled from the textures, the results are similar to applying a dark tint across the entire scene. The same effect can be accomplished if the RGB values were set to 1.0 and the intensity was set 0.3 because the two sets of values are simply multiplied.

Before moving onto the next project, try fiddling with the ambient red channel and the ambient intensity to observe their effects on the scene. By pressing the right-arrow key, you can increase the intensity of the entire scene and make all objects more visible. Continue with this increment and observe that when the intensity reaches values beyond 15.0, all colors in the scene converge toward white or begin to oversaturate. Without proper context, oversaturation can be a distraction. However, it is also true that strategically creating oversaturation on selective objects can be used to indicate significant events, for example, triggering a trap. The next section describes how to create and direct a light source to illuminate selected objects.

360

Light Source

361 Examine your surroundings and you can observe many types of light sources, for
 362 example, your table lamp, light rays from the sun, or an isolated light bulb. The isolated
 363 light bulb can be described as a point that emits light uniformly in all directions or a
 364 point light. The point light is where you will begin to analyze light sources.

365 Fundamentally, a point light illuminates an area or radius around a specified point.
 366 In 3D space, this region of illumination is simply a sphere, referred to as a volume of
 367 illumination. The volume of illumination of a point light is defined by the position of the
 368 light, or the center of the sphere, and the distance that the light illuminates, or the radius
 369 of the sphere. To observe the effects of a light source, objects must be present and within
 370 the volume of illumination.

371 As mentioned in the introduction of this chapter, the 2D engine will need to venture
 372 into the third dimension to properly simulate the propagation of light energy. Now,
 373 consider your 2D engine; thus far, you have implemented a system in which everything
 374 is in 2D. An alternative way is to interpret that the engine defines and renders everything
 375 on a single plane where $z = 0$ and objects are layered by drawing order. On this system,
 376 you are going to add light sources that reside in 3D.

377 To observe the effects of a light source, its illumination volume must overlap an
 378 object on the XY plane where your objects are defined. Figure 8-2 shows the volume of
 379 illumination from a simple point light located at $z = 10$ intersecting a plane at $z = 0$. This
 380 intersection results in an illuminated circle on the plane. The next project implements
 381 Figure 8-2 where you will examine light sources with an object-oriented approach while
 382 adhering to the expectations of how a light illuminates a scene. This can be achieved
 383 through the definition of a Light object to represent a light source.

this figure will be printed in b/w

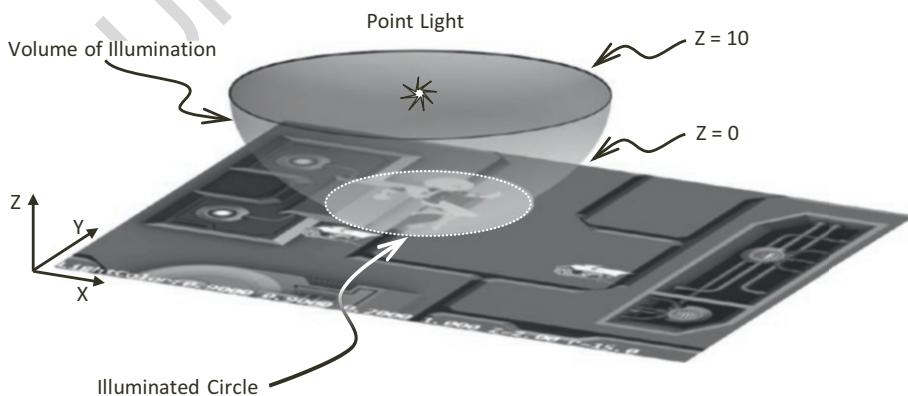


Figure 8-2. Point light and the corresponding volume of illumination in 3D

GLSL Implementation and Integration into the Game Engine

384

Recall that the engine interfaces to the GLSL shaders with the corresponding subclasses of the SimpleShader/Renderable pairs. SimpleShader and its subclasses interface to the GLSL shaders and Renderable, and its subclasses provide programmers with the convenience of manipulating many copies of geometries with the same shader type. For example, texture_vs.gls1 and texture_fs.gls1 are interfaced to the game engine via the TextureShader object, and the TextureRenderable objects allow game programmers to create and manipulate multiple instances of geometries shaded by the texture_vs/fs shaders. Figure 8-3 depicts that the next project extends this architecture to implement point light illumination. The Light class encapsulates the attributes of a point light including position, radius, and color. This information is forwarded to the GLSL fragment shader, light_fs, via the LightShader/LightRenderable pair for computing the appropriate pixel colors. The GLSL vertex shader, texture_vs, is reused because light source illumination involves the same information to be processed at each vertex.

this figure will be printed in b/w

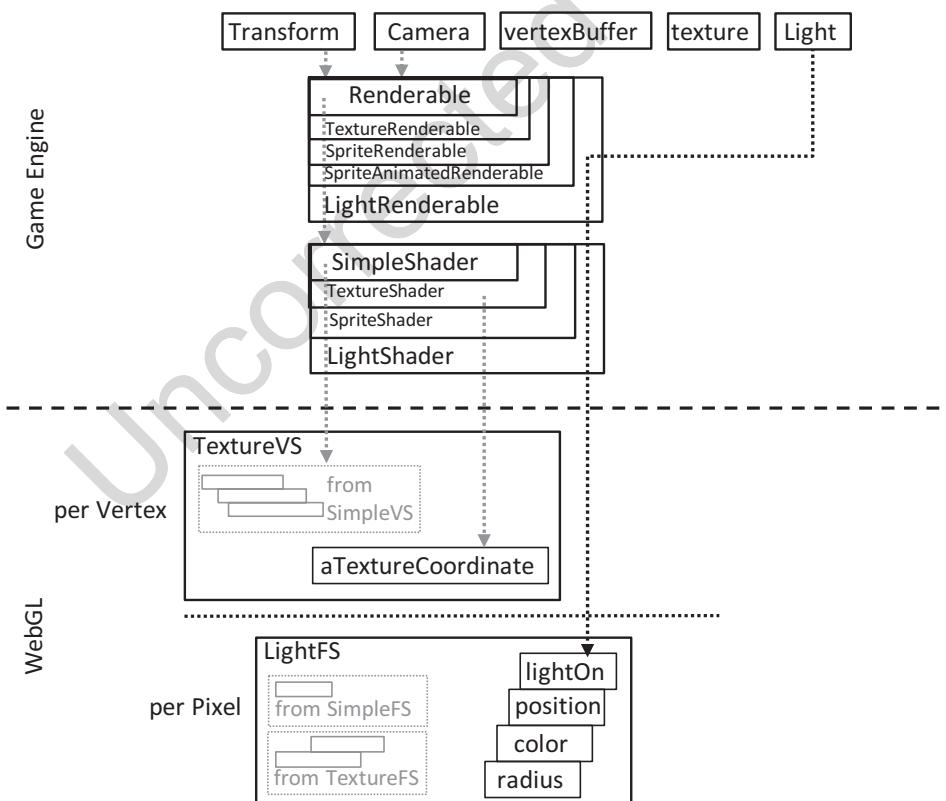


Figure 8-3. LightShader/LightRenderable pair and the corresponding GLSL LightShader

Finally, it is important to remember that the GLSL fragment shader is invoked once for every pixel covered by the corresponding geometry. This means that the GLSL fragment shaders you are about to create will be invoked many times per frame, probably in the range of hundreds of thousands or even millions. Considering the fact that the game loop initiates redrawing at a real-time rate, or around 60 frame redraws per second, the GLSL fragment shaders will be invoked many millions of times per second! The efficiency of the implementation is important for a smooth experience.

The Simple Light Shader Project

This project demonstrates how to implement and illuminate with a simple point light. You can see an example of this project running in Figure 8-4. The source code of this project is located in the chapter8/8.2.simple_light_shader folder.

this figure will be printed in b/w



Figure 8-4. Running the Simple Light Shader project

The controls of the project are as follows:	409
• WASD keys: Move the hero character on the screen	410
• WASD keys + left mouse button: Move the hero character and the light source around the screen	411
• Left-/right-arrow key: Decreases/increases the light intensity	413
• Z/X key: Increases/decreases the light Z position	414
• C/V key: Increases/decreases the light radius	415
The goals of the project are as follows:	416
• To understand how to simulate the illumination effects from a point light	417
• To observe point light illumination	419
• To implement a GLSL shader that supports point light illumination	420

Creating the GLSL Light Fragment Shader

As with the previous section, the implementation will begin with the GLSL shader. It is not necessary to define a new GLSL vertex shader as the per vertex information and computation involved are identical to that of `texture_vs`. A new GLSL fragment shader must be defined to compute the illuminated circle.

1. In the `src/gls1_shaders` folder, create a new file and name it `light_fs.gls1`.
426
427
2. Refer to `texture_fs.gls1` and copy all uniform and varying variables. This is an important step because the `light_fs` fragment shader will interface to the game engine via the `LightShader` class.
428
429
The `LightShader` class, in turn, will be implemented as a subclass of `TextureShader`, where the existence of these variables is assumed.
430
431
432

```
precision mediump float;  
433  
  
// The object that fetches data from texture.  
434  
// Must be set outside the shader.  
435  
uniform sampler2D uSampler;  
436
```

CHAPTER 8 IMPLEMENTING ILLUMINATION AND SHADOW

```
437 // Color of pixel  
438 uniform vec4 uPixelColor;  
439 uniform vec4 uGlobalAmbientColor; // this is shared globally  
440 uniform float uGlobalAmbientIntensity;  
441 // "varying" keyword signifies that the texture coordinate will be  
442 // interpolated and thus varies.  
443 varying vec2 vTexCoord;  
444 ... implementation to follow ...
```

- 445 3. Now, define the variables to support a point light: on/off switch,
446 color, position, and radius. It is important to note that the position
447 and radius are in units of pixels.

```
448 // Light information  
449 uniform bool uLightOn;  
450 uniform vec4 uLightColor;  
451 uniform vec3 uLightPosition; // in pixel space!  
452 uniform float uLightRadius; // in pixel space!
```

- 453 4. Implement the light illumination in the `main()` function as follows:
 - 454 a. Step A, sample the texture color and apply the ambient color
455 and intensity.
 - 456 b. Step B, perform the light source illumination. This is
457 accomplished by determining if the computation is
458 required—testing if the light is switched on and if the pixel is
459 nontransparent. If both are favorable, the distance between
460 the light position and the current pixel is compared with the
461 light radius to determine if the pixel is inside the volume of
462 illumination. Note that `gl_FragCoord.xyz` is the GLSL-defined
463 variable for current pixel position and that this computation
464 assumes pixel-space units. When all conditions are favorable,
465 the color of the light is accumulated to the final results.

- c. The last step is to apply the tint and to set the final color via
gl_FragColor.

```

void main(void)  {
    // Step A: sample the texture and apply ambient
    vec4 textureMapColor = texture2D(uSampler,
                                    vec2(vTexCoord.s, vTexCoord.t));
    vec4 lgtResults = uGlobalAmbientIntensity * uGlobalAmbientColor;
    // Step B: decide if the light should illuminate
    if (uLightOn && (textureMapColor.a > 0.0)) {
        float dist = length(uLightPosition.xyz - gl_FragCoord.xyz);
        if (dist <= uLightRadius)
            lgtResults += uLightColor;
    }
    lgtResults *= textureMapColor;
    // Step C: tint texture leave transparent area defined by texture
    vec3 r = vec3(lgtResults) * (1.0-uPixelColor.a) +
             vec3(uPixelColor) * uPixelColor.a;
    vec4 result = vec4(r, textureMapColor.a);
    gl_FragColor = result;
}

```

Defining a Light Class

With the GLSL light_fs shader defined, you can now define a class to encapsulate a point light source for the game engine:

1. Create a new lights folder in the src/engine folder. In the lights folder, add a new file and name it lights.js.
2. Edit lights.js to create the Light class, and define the constructor to initialize the light color, position, radius, and on/off status. Remember to export the class.

CHAPTER 8 IMPLEMENTING ILLUMINATION AND SHADOW

```
494 class Light {  
495     constructor() {  
496         this.mColor = vec4.fromValues(0.1, 0.1, 0.1, 1); // light color  
497         this.mPosition = vec3.fromValues(0, 0, 5); // WC light position  
498         this.mRadius = 10; // effective radius in WC  
499         this.mIsOn = true;  
500     }  
501     ... implementation to follow ...  
502 }  
503 export default Light;
```

504 3. Define the getters and setters for the instance variables:

```
505 // simple setters and getters  
506 setColor(c) { this.mColor = vec4.clone(c); }  
507 getColor() { return this.mColor; }  
508 set2DPosition(p) {  
509     this.mPosition = vec3.fromValues(p[0], p[1], this.mPosition[2]); }  
510 setXPos(x) { this.mPosition[0] = x; }  
511 setYPos(y) { this.mPosition[1] = y; }  
512 setZPos(z) { this.mPosition[2] = z; }  
513 getPosition() { return this.mPosition; }  
514 setRadius(r) { this.mRadius = r; }  
515 getRadius() { return this.mRadius; }  
516 setLightTo(isOn) { this.mIsOn = isOn; }  
517 isLightOn() { return this.mIsOn; }
```

518 Lastly, remember to update the engine access file, `index.js`, to forward the newly
519 defined functionality to the client.

Defining the LightShader Class

The LightShader class subclasses from the SpriteShader to encapsulate the communication of the values that are specific to the uniform variables defined for a point light source in the `light_fs` fragment shader. In this way, the LightShader class can serve as a convenient interface for the GLSL fragment shader.

1. In the `src/engine/shaders` folder, create a new file and name it `light_shader.js`. 526
2. Define the LightShader class to be a subclass of SpriteShader. In the constructor, define the necessary variables to support sending the information associated with a point light to the `light_fs` fragment shader. The point light information in the engine is stored in `mLight`, while the reference to the Camera is important to convert all information from WC to pixel space. The last four lines of the constructor query to obtain the reference locations to the uniform variables in `light_fs`. Don't forget to export the class. 534

```
import SpriteShader from "./sprite_shader.js";
import * as glSys from "../core/gl.js";

class LightShader extends SpriteShader {
    constructor(vertexShaderPath, fragmentShaderPath) {
        // Call super class constructor
        super(vertexShaderPath, fragmentShaderPath);

        // glsl uniform position references
        this.mColorRef = null;
        this.mPosRef = null;
        this.mRadiusRef = null;
        this.mIsOnRef = null;

        this.mLight = null; // the light source in the Game Engine
        this.mCamera = null; // camera to draw, need for WC to DC xform
        //
        // create the references to these uniforms in the LightShader
        let shader = this.mCompiledShader;
        let gl = glSys.get();
```

CHAPTER 8 IMPLEMENTING ILLUMINATION AND SHADOW

```
552     this.mColorRef = gl.getUniformLocation(shader, "uLightColor");  
553     this.mPosRef = gl.getUniformLocation(shader, "uLightPosition");  
554     this.mRadiusRef = gl.getUniformLocation(shader, "uLightRadius");  
555     this.mIsOnRef = gl.getUniformLocation(shader, "uLightOn");  
556 }  
557 ... implementation to follow ...  
558 }  
559 export default LightShader;
```

- 560 3. Define a simple setter function to associate a light and camera
561 with the shader:

```
562 setCameraAndLight(c, l) {  
563     this.mCamera = c;  
564     this.mLight = l;  
565 }
```

- 566 4. Override the activate() function to append the new functionality
567 of loading the point light information in mLight when the light is
568 present. Notice that you still call the activate() function of the
569 super class to communicate the rest of the values to the uniform
570 variables of the light_fs fragment shader.

```
571 activate(pixelColor, trsMatrix, cameraMatrix) {  
572     // first call the super class's activate  
573     super.activate(pixelColor, trsMatrix, cameraMatrix);  
574  
575     if (this.mLight !== null) {  
576         this._loadToShader();  
577     } else {  
578         glSys.get().uniform1i(this.mIsOnRef, false); // switch off light!  
579     }  
580 }
```

5. Implement the `_loadToShader()` function to communicate the values of the point light to the uniform variables in the shader. Recall that this communication is performed via the references created in the constructor and the set uniform functions. It is important to note that the camera provides the new coordinate space transformation functionality of `wcPosToPixel()` and `wcSizeToPixel()`. These two functions ensure corresponding values in the `light_fs` are in pixel space such that relevant computations such as distances between positions can be performed. The implementation of these functions will be examined shortly.

```

_loadToShader(aCamera) {
    let gl = glSys.get();
    gl.uniform1i(this.mIsOnRef, this.mLight.isLightOn());
    if (this.mLight.isLightOn()) {
        let p = this.mCamera.wcPosToPixel(this.mLight.getPosition());
        let r = this.mCamera.wcSizeToPixel(this.mLight.getRadius());
        let c = this.mLight.getColor();
        gl.uniform4fv(this.mColorRef, c);
        gl.uniform3fv(this.mPosRef, vec3.fromValues(p[0], p[1], p[2]));
        gl.uniform1f(this.mRadiusRef, r);
    }
}

```

Defining the LightRenderable Class

With `LightShader` defined to interface to the GLSL `light_fs` shader, you can now focus on defining a new `Renderable` class for the game programmer. It is important that a light can shine on and illuminate all `Renderable` types, including those with texture and animated sprites. For this reason, the new class must encapsulate all existing `Renderable` functionality and be a subclass of `SpriteAnimateRenderable`. You can think of this new class as a `SpriteAnimateRenderable` that can be illuminated by a `Light` object.

- Create a new file in the `src/engine/renderables` folder and name it `light_renderable.js`.

CHAPTER 8 IMPLEMENTING ILLUMINATION AND SHADOW

612 2. Define the LightRenderable class to extend
613 SpriteAnimateRenderable, set the shader to reference the
614 new LightShader, and initialize a Light reference in the
615 constructor. This is the light that shines and illuminates the
616 SpriteAnimateRenderable. Don't forget to export the class.

```
617 import SpriteAnimateRenderable from "./sprite_animate_renderable.js";  
618 import * as defaultShaders from "../core/shader_resources.js";  
  
619 class LightRenderable extends SpriteAnimateRenderable {  
  
620     constructor(myTexture) {  
621         super(myTexture);  
622         super._setShader(defaultShaders.getLightShader());  
  
623         // here is the light source  
624         this.mLight = null;  
625     }  
  
626     ... implementation to follow ...  
  
627 }  
628 export default LightRenderable;
```

629 3. Define a draw function to pass the camera and illuminating light
630 source to the LightShader before invoking the superclass draw()
631 function to complete the drawing:

```
632     draw(camera) {  
633         this.mShader.setCameraAndLight(camera, this.mLight);  
634         super.draw(camera);  
635     }
```

636 4. Lastly, simply add the support to get and set the light:

```
637     getLight() { return this.mLight; }  
638     addLight(l) { this.mLight = l; }
```

639 Before moving on, remember to update the engine access file, index.js, to forward
640 the newly defined functionality to the client.

Defining a Default LightShader Instance

641

As discussed, when you first defined TextureShader (Chapter 5), only a single instance is required for each shader type, and all the shaders are always hidden from the game programmer by a corresponding Renderable type. Each instance of the shader type is created during engine initialization by the shaderResources module in the src/engine/core folder.

642

643

644

645

646

You can now modify the engine to support the initializing, loading, and unloading of a LightShader object to be shared engine-wide:

647

648

1. Edit shader_resources.js in the src/engine/core folder to import LightShader; define the path to the GLSL source code, a corresponding variable and access function for the shader:

649

650

651

... identical to previous code ...

652

```
import LightShader from "../shaders/light_shader.js";
```

653

```
// Light Shader
```

654

```
let kLightFS = "src/glsl_shaders/light_fs.glsl"; // FragmentShader
```

655

```
let mLightShader = null;
```

656

```
function getLightShader() { return mLightShader; }
```

657

2. Create a new instance of light shader in the createShaders() function:

658

659

```
function createShaders() {
```

660

```
    mConstColorShader = new SimpleShader(kSimpleVS, kSimpleFS);
```

661

```
    mTextureShader = new TextureShader(kTextureVS, kTextureFS);
```

662

```
    mSpriteShader = new SpriteShader(kTextureVS, kTextureFS);
```

663

```
    mLineShader = new LineShader(kSimpleVS, kLineFS);
```

664

```
mLightShader = new LightShader(kTextureVS, kLightFS);
```

665

```
}
```

666

3. Load the light shader GLSL source code in the init() function:

667

```
function init() {
```

668

```
    let loadPromise = new Promise(
```

669

```
        async function(resolve) {
```

670

```
            await Promise.all([
```

671

```

672         text.load(kSimpleFS),
673         text.load(kSimpleVS),
674         text.load(kTextureFS),
675         text.load(kTextureVS),
676         text.load(kLineFS),
677         text.load(kLightFS)
678     ]);
679     resolve();
680 }).then(
681     function resolve() { createShaders(); }
682 );
683 map.pushPromise(loadPromise);
684 }

```

- 685 4. Remember to release GLSL resources and unload the source code
- 686 during cleanup:

```

687 function cleanUp() {
688     mConstColorShader.cleanUp();
689     mTextureShader.cleanUp();
690     mSpriteShader.cleanUp();
691     mLineShader.cleanUp();
692     mLightShader.cleanUp();
693     text.unload(kSimpleVS);
694     text.unload(kSimpleFS);
695     text.unload(kTextureVS);
696     text.unload(kTextureFS);
697     text.unload(kLineFS);
698     text.unload(kLightFS);
699 }

```

- 700 5. Lastly, export the access function to allow sharing of the created
- 701 instance in the engine:

```

702 export {init, cleanUp,
703         getConstColorShader, getTextureShader,
704         getSpriteShader, getLineShader, getLightShader}

```

Modifying the Camera

The Camera utility functions, such as `wcPosToPixel()`, are invoked multiple times while rendering the `LightShader` object. These functions compute the transformation between WC and pixel space. This transformation requires the computation of intermediate values, for example, lower-left corner of WC window, that do not change during each rendering invocation. To avoid repeated computation of these values, a per-render invocation cache should be defined for the Camera object.

Defining a Per-Render Cache for the Camera

Define a per-render cache to store intermediate values that are required to support shading operations:

1. Edit `camera_main.js` and define a `PerRenderCache` class; in the constructor, define variables to hold the ratio between the WC space and the pixel space as well as the origin of the Camera. These are intermediate values required for computing the transformation from WC to pixel space, and these values do not change once rendering begins.

```
class PerRenderCache {
    // Information to be updated once per render for efficiency concerns
    constructor() {
        this.mWCToPixelRatio = 1; // WC to pixel transformation
        this.mCameraOrgX = 1; // Lower-left corner of camera in WC
        this.mCameraOrgY = 1;
    }
}
```

2. Modify the `Camera` class to instantiate a new `PerRenderCache` object. It is important to note that this variable represents local caching of information and should be hidden from the rest of the engine.

```
constructor(wcCenter, wcWidth, viewportArray, bound) {
    ... identical to previous code ...
}
```

```

735 // per-rendering cached information
736 // needed for computing transforms for shaders
737 // updated each time in SetupViewProjection()
738 this.mRenderCache = new PerRenderCache();
739     // SHOULD NOT be used except
740     // xform operations during the rendering
741     // Client game should not access this!
742 }

743     3. Initialize the per-render cache in the setViewAndCameraMatrix()
744         function by adding a step B3 to calculate and set the cache based
745         on the Camera viewport width, world width, and world height:

746 setViewAndCameraMatrix() {
747     ... identical to previous code ...
748
749     // Step B2: first operation is to translate camera center to origin
750     mat4.translate(this.mCameraMatrix, this.mCameraMatrix,
751                     vec3.fromValues(-center[0], -center[1], 0));
752
753     // Step B3: compute and cache per-rendering information
754     this.mRenderCache.mWCtoPixelRatio =
755         this.mViewport[eViewport.eWidth] / this.getWCWidth();
756     this.mRenderCache.mCameraOrgY = center[1] - (this.getWCHeight() / 2);
757     this.mRenderCache.mCameraOrgX = center[0] - (this.getWCWidth() / 2);
758 }

```

757 Notice that the PerRenderCache class is completely local to the `camera_main.js` file.
 758 It is important to hide and carefully handle complex local caching functionality.

759 Adding Camera Transform Functions

760 Now that the per-render cache is defined and properly initialized, you can extend the
 761 functionality of the camera to support transformations from WC to pixel space. For code
 762 readability and maintainability, this functionality will be implemented in a separate file.
 763 Another important note is that since you are converting from WC to pixel space and pixel
 764 space has no z axis, you need to calculate a *fake* z value for the pixel space coordinate.

1. Edit the Camera access file, `camera.js`, to import from the file, `camera_xform.js`, which will contain the latest functionality additions, the WC to pixel space transform support:

```
import Camera from "./camera_xform.js"; 768
export default Camera; 769
```

2. In the `src/engine/cameras` folder, create a new file and name it `camera_xform.js`. Import from `camera_input.js` such that you can continue to add new functionality to the Camera class, and do not forget to export.

```
import Camera from "./camera_input.js"; 774
import { eViewport } from "./camera_main.js"; 775
... implementation to follow ... 776
export default Camera; 777
```

3. Create a function to approximate a fake pixel space z value by scaling the input parameter according to the `mWCToPixelRatio` variable:

```
Camera.prototype.fakeZInPixelSpace = function (z) { 781
  return z * this.mRenderCache.mWCToPixelRatio;
} 783
```

4. Define a function to convert from WC to pixel space by subtracting the camera origin followed by scaling with the `mWCToPixelRatio`. The 0.5 offset at the end of the x and y conversion ensures that you are working with the center of the pixel rather than a corner.

```
Camera.prototype.wcPosToPixel = function (p) { // p is a vec3, fake Z 788
  // Convert the position to pixel space
  let x = this.mViewport[eViewport.eOrgX] +
    ((p[0] - this.mRenderCache.mCameraOrgX) *
      this.mRenderCache.mWCToPixelRatio) + 0.5;
  let y = this.mViewport[eViewport.eOrgY] +
    ((p[1] - this.mRenderCache.mCameraOrgY) *
```

```
795         this.mRenderCache.mWCToPixelRatio) + 0.5;  
796     let z = this.fakeZInPixelSpace(p[2]);  
797     return vec3.fromValues(x, y, z);  
798 }
```

- 799 5. Lastly, define a function for converting a length from WC to pixel
800 space by scaling with the mWCToPixelRatio variable:

```
801 Camera.prototype.wcSizeToPixel = function (s) { //  
802     return (s * this.mRenderCache.mWCToPixelRatio) + 0.5;  
803 }
```

804 Testing the Light

805 The MyGame level must be modified to utilize and test the newly defined light
806 functionality.

807 Modifying the Hero and Minion

808 Modify the Hero and Minion classes to accommodate the new LightRenderable object:

- 809 1. Edit the hero.js file in the src/my_game/objects folder;
810 in the constructor, replace the SpriteRenderable with a
811 LightRenderable instantiation:

```
812 constructor(spriteTexture) {  
813     super(null);  
814     this.kDelta = 0.3;  
815     this.mRenderComponent = new engine.LightRenderable(spriteTexture);  
816     ... identical to previous code ...  
817 }
```

- 818 2. Edit the minion.js file in the src/my_game/objects folder;
819 in the constructor, replace the SpriteRenderable with a
820 LightRenderable instantiation:

```
821 constructor(spriteTexture, atX, atY) {  
822     super(null);  
823     this.kDelta = 0.2;
```

```

this.mRenderComponent = new engine.LightRenderable(spriteTexture);      824
... identical to previous code ...                                     825
}

```

Modifying the MyGame Object

827

With the implementation of the light completed and the game objects properly updated, you can now modify the MyGame level to display and test the light source. Because of the simplistic and repetitive nature of the code changes in the `my_game_main.js` file of adding variables for the new objects, initializing the objects, drawing the objects, and updating the objects, the details will not be shown here.

828

829

830

831

832

Observations

833

With the project now complete, you can run it and examine the results. There are a few observations to take note of. First is the fact that the illuminated results from the light source look like a circle. As depicted in Figure 8-2, this is the illuminated circle of the point light on the $z = 0$ plane where your objects are located. Press the Z or X key to increase or decrease the light's z position to observe that the illuminated circle decreases and increases in size as a result of intersection area changes. The sphere/plane intersection result can be verified when you continue to increase/decrease the z position. The illuminated circle will eventually begin to decrease in size and ultimately disappear completely when the sphere is moved more than its radius away from the $z=0$ plane.

834

835

836

837

838

839

840

841

842

843

You can also press the C or V key to increase or decrease the point light radius to increase or decrease the volume of illumination, and observe the corresponding changes in the illuminated circle radius.

844

845

846

Now, press the WASD keys along with the left mouse button to move the Hero and observe that the point light always follow the Hero and properly illuminates the background. Notice that the light source illuminates the left minion, the hero, and the background but not the other three objects in the scene. This is because the right minion and the red and green blocks are not `LightRenderable` objects and thus cannot be illuminated by the defined light source.

847

848

849

850

851

852

853

Multiple Light Sources and Distance Attenuation

854 In the previous project, a single point light source was defined with the capability of
 855 illuminating a spherical volume. This type of light source is useful in many games, but
 856 it is restrictive to be limited to only a single light source. The engine should support the
 857 illumination from multiple light sources to fulfill the design needs of different games.
 858 This shortcoming is remedied in the next project with general support for multiple
 859 light sources. The implementation principle for multiple lights remains the same as
 860 the previous project, with the modification of replacing the single light source with an
 861 array of lights. As illustrated in Figure 8-5, a new Light object will be defined, while the
 862 LightRenderable object will be modified to support an array of the Light objects. The
 863 LightShader object will define an array of ShaderLightAtIndex objects that are capable
 864 of communicating light source information to the uLights array in the GLSL light_fs
 865 fragment shader for illumination computations.

this figure will be printed in b/w

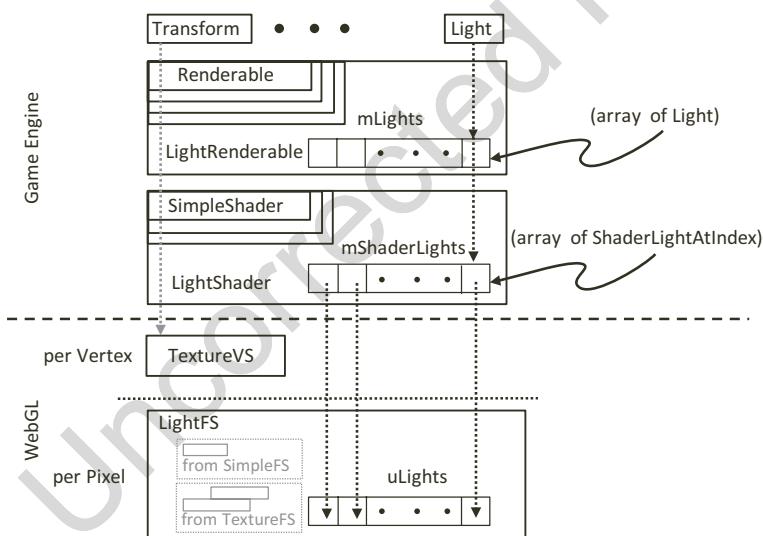


Figure 8-5. Support for multiple light sources

866 The point light illumination results from the previous project can be improved.
 867 You have observed that at its boundary, the illuminated circle disappears abruptly
 868 with a sharp brightness transition. This sudden disappearance of illumination results
 869 does not reflect real life where effects from a given light source decrease gradually over
 870 distance instead of switching off abruptly. A more visually pleasing light illumination

result should show an illuminated circle where the illumination results at the boundary 871
disappear gradually. This gradual decrease of light illumination effect over distance 872
is referred to as distance attenuation. It is a common practice to approximate distant 873
attenuation with quadratic functions because they produce effects that resemble the 874
real world. In general, distance attenuation can be approximated in many ways, and it is 875
often refined to suit the needs of the game. 876

In the following, you will implement a near and far cutoff distance, that is, two 877
distances from the light source at which the distance attenuation effect will begin and 878
end. These two values give you control over a light source to show a fully illuminated 879
center area with illumination drop-off occurring only at a specified distance. Lastly, a 880
light intensity will be defined to allow the dimming of light without changing its color. 881
With these additional parameters, it becomes possible to define dramatically different 882
effects. For example, you can have a soft, barely noticeable light that covers a wide area 883
or an oversaturated glowing light that is concentrated over a small area in the scene. 884

The Multiple Lights Project

This project demonstrates how to implement multiple point lights within a single scene. 885
It also demonstrates how to increase the sophistication of your point light model so 886
that they are more flexible to serve a wider variety of purposes. You can see an example 887
of this project running in Figure 8-6. The source code of this project is located in the 888
chapter8/8.3.multiple_lights folder. 889

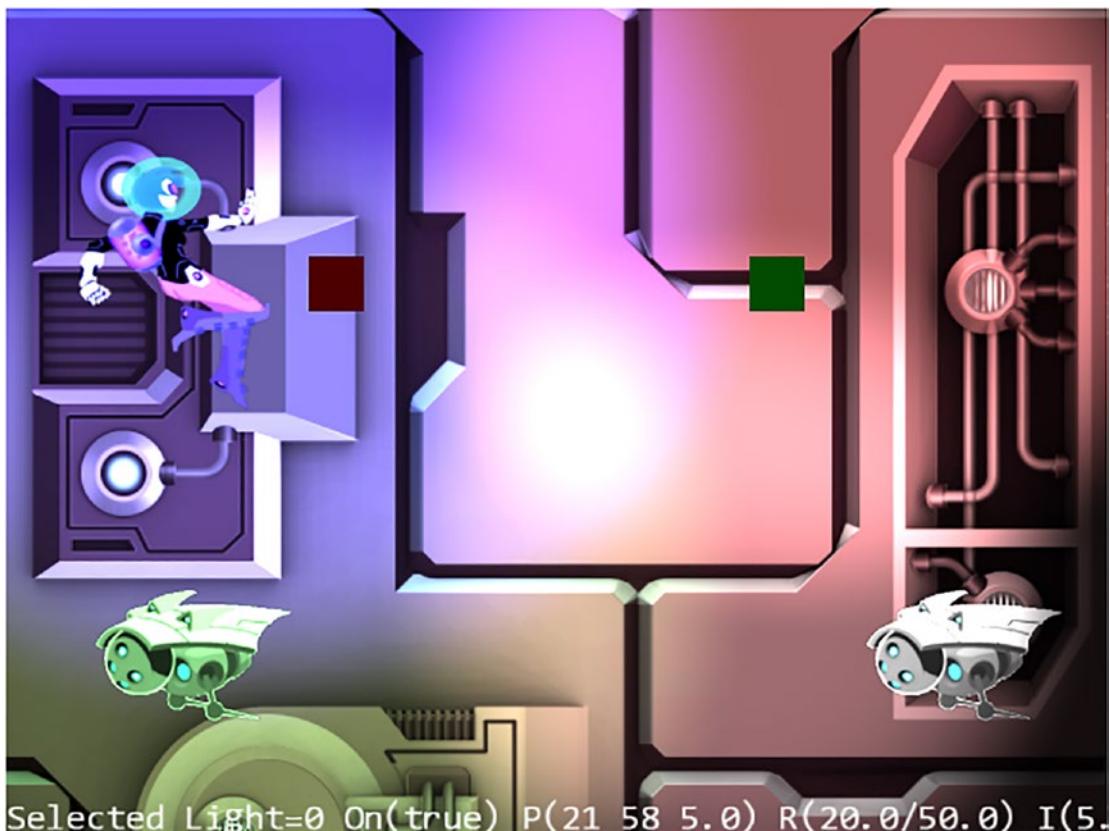


Figure 8-6. Running the Multiple Lights project

The controls of the project are as follows:

- **WASD keys:** Move the hero character on the screen
- **Number keys 0, 1, 2, and 3:** Select the corresponding light source
- **Arrow keys:** Move the currently selected light
- **Z/X keys:** Increase/decrease the light z position
- **C/V and B/N keys:** Increase/decrease the near and far cutoff distances of the selected light
- **K/L keys:** Increase/decrease the intensity of the selected light
- **H key:** Toggles the selected light on/off

- The goals of the project are as follows: 900
- To build the infrastructure for supporting multiple light sources in 901
the engine and GLSL shaders 902
 - To understand and examine the distance attenuation effects of light 903
 - To experience controlling and manipulating multiple light sources in 904
a scene 905

Modifying the GLSL Light Fragment Shader 906

The light_fs fragment shader needs to be modified to support the distance attenuation, 907
cutoffs, and multiple light sources: 908

1. In the light_fs.gls1 file, remove the light variables that were 909
added for a single light and add a struct for light information that 910
holds the position, color, near-distance, far-distance, intensity, 911
and on/off variables. With the struct defined, add a uniform 912
array of lights to the fragment shader. Notice that a #define has 913
been added to hold the number of light sources to be used. 914

Note GLSL requires array sizes and the number of loop iterations to be constants. 915
The kGLSLuLightArraySize is the constant for light array size and the 916
corresponding loop iteration control. Feel free to change this value to define as 917
many lights as the hardware can support. For example, you can try increasing the 918
number of lights to 50 and then test and measure the performance. 919

```
// Light information 920
#define kGLSLuLightArraySize 4 921
    // GLSL Fragment shader requires loop control 922
    // variable to be a constant number. This number 4 923
    // says, this fragment shader will _ALWAYS_ process 924
    // all 4 light sources. 925
    // *****WARNING***** 926
    // This number must correspond to the constant with 927
    // the same name defined in LightShader.js file. 928
```

CHAPTER 8 IMPLEMENTING ILLUMINATION AND SHADOW

```
929 // *****WARNING*****  
930 // To change this number MAKE SURE: to update the  
931 //      kGLSLuLightArraySize  
932 // defined in LightShader.js file.  
  
933 struct Light {  
934     vec3 Position;    // in pixel space!  
935     vec4 Color;  
936     float Near;      // distance in pixel space  
937     float Far;        // distance in pixel space  
938     float Intensity;  
939     bool IsOn;  
940 };  
941 uniform Light uLights[kGLSLuLightArraySize];  
942         // Maximum array of lights this shader supports  
  
943     2. Define LightEffect() function to compute the illumination  
944         results from a light source. This function uses the distance  
945         between the light and the current pixel to determine whether the  
946         pixel lies within the near radius, in between near and far radii, or  
947         farther than the far radius. If the pixel position lies within the near  
948         radius, there is no attenuation, so the strength is set to 1. If the  
949         position is in between the near and far radii, then the strength is  
950         modulated by a quadratic function. A distance of greater than the  
951         far radius will result in no illumination from the corresponding  
952         light source, or a strength of 0.  
  
953 vec4 LightEffect(Light lgt) {  
954     vec4 result = vec4(0);  
955     float strength = 0.0;  
956     float dist = length(lgt.Position.xyz - gl_FragCoord.xyz);  
957     if (dist <= lgt.Far) {  
958         if (dist <= lgt.Near)  
959             strength = 1.0; // no attenuation  
960         else {  
961             // simple quadratic drop off  
962             float n = dist - lgt.Near;
```

```

        float d = lgt.Far - lgt.Near;                                963
        strength = smoothstep(0.0, 1.0, 1.0-(n*n)/(d*d));          964
            // blended attenuation                                965
        }
    }
    result = strength * lgt.Intensity * lgt.Color;                968
    return result;
}

```

3. Modify the main function to iterate through all the defined light sources and call the `LightEffect()` function to calculate and accumulate the contribution from the corresponding light in the array:

```

void main(void) {
    // simple tint based on uPixelColor setting
    vec4 textureMapColor = texture2D(uSampler,
                                    vec2(vTexCoord.s, vTexCoord.t));
    vec4 lgtResults = uGlobalAmbientIntensity * uGlobalAmbientColor;
    // now decide if we should illuminate by the light
    if (textureMapColor.a > 0.0) {
        for (int i=0; i<kGLSLuLightArraySize; i++) {
            if (uLights[i].IsOn) {
                lgtResults += LightEffect(uLights[i]);
            }
        }
    }
    lgtResults *= textureMapColor;
    ... identical to previous code ...
}

```

Modifying the Light Class

The game engine Light object must be modified to reflect the newly added properties in the `light_fs` fragment shader: near and far attenuations and intensity.

- 994 1. Modify the Lights.js constructor to define variables for the new
 995 properties:

```
996 constructor() {  

  997     this.mColor = vec4.fromValues(0.1, 0.1, 0.1, 1); // light color  

  998     this.mPosition = vec3.fromValues(0, 0, 5); // light position in WC  

  999     this.mNear = 5; // effective radius in WC  

 1000    this.mFar = 10; // within near is full on, outside far is off  

 1001    this.mIntensity = 1;  

 1002    this.mIsOn = true;  

 1003 }
```

- 1004 2. Define the corresponding get and set accessors for the variables.
 1005 Note that the radius variable has been generalized and replaced
 1006 by the near and far cutoff distances.

```
1007 setNear(n) { this.mNear = n; }  

1008 getNear() { return this.mNear; }  

1009 setFar(f) { this.mFar = f; }  

1010 getFar() { return this.mFar; }  

1011 setIntensity(i) { this.mIntensity = i; }  

1012 getIntensity() { return this.mIntensity; }  

1013 setLightTo(on) { this.mIsOn = on; }
```

1014 Defining the LightSet Class

1015 You will define a LightSet class to facilitate the working with a collection of Light
 1016 objects. In the src/engine/lights folder, create a new file and name it light_set.js.
 1017 Define the basic interface for working with a set of Light objects.

```
1018 class LightSet {  

  1019     constructor() { this.mSet = []; }  

  1020     numLights() { return this.mSet.length; }  

  1021     getLightAt(index) { return this.mSet[index]; }
```

```

    addToSet(light) { this.mSet.push(light); }
}
export default LightSet;

```

Lastly, don't forget to export the class and remember to update the engine access file, index.js, to forward the newly defined functionality to the client.

Defining the ShaderLightAt Class

Define the ShaderLightAt class to send information from a Light object to an element in the uLights array in the light_fs GLSL fragment shader:

1. In the src/engine/shaders folder, create a new file and name it shader_light_at.js; define the ShaderLightAt class and the constructor to receive a shader and an index to the uLight array.
Don't forget to export the class.

```

import * as glSys from "../core/gl.js";

class ShaderLightAt {
    constructor(shader, index) {
        this._setShaderReferences(shader, index);
    }
    ... implementation to follow ...
}
export default ShaderLightAt;

```

2. Implement the _setShaderReferences() function to set the light property references to a specific index in the uLights array in the light_fs fragment shader:

```

_setShaderReferences(aLightShader, index) {
    let gl = glSys.get();
    this.mColorRef = gl.getUniformLocation(
        aLightShader, "uLights[" + index + "].Color");
    this.mPosRef = gl.getUniformLocation(
        aLightShader, "uLights[" + index + "].Position");
    this.mNearRef = gl.getUniformLocation(
        aLightShader, "uLights[" + index + "].Near");

```

CHAPTER 8 IMPLEMENTING ILLUMINATION AND SHADOW

```
1053     this.mFarRef = gl.getUniformLocation(  
1054             aLightShader, "uLights[" + index + "].Far");  
1055     this.mIntensityRef = gl.getUniformLocation(  
1056             aLightShader, "uLights[" + index + "].Intensity");  
1057     this.mIsOnRef = gl.getUniformLocation(  
1058             aLightShader, "uLights[" + index + "].IsOn");  
1059 }
```

- 1060 3. Implement the `loadToShader()` function to push the properties of
1061 a light to the `light_fs` fragment shader. Notice that this function
1062 is similar to the `_loadToShader()` function defined in the `light_`
1063 `shader.js` file from previous project. The important difference is
1064 that in this case, light information is loaded to a specific array index.

```
1065 loadToShader(aCamera, aLight) {  
1066     let gl = glSys.get();  
1067     gl.uniform1i(this.mIsOnRef, aLight.isLightOn());  
1068     if (aLight.isLightOn()) {  
1069         let p = aCamera.wcPosToPixel(aLight.getPosition());  
1070         let n = aCamera.wcSizeToPixel(aLight.getNear());  
1071         let f = aCamera.wcSizeToPixel(aLight.getFar());  
1072         let c = aLight.getColor();  
1073         gl.uniform4fv(this.mColorRef, c);  
1074         gl.uniform3fv(this.mPosRef, vec3.fromValues(p[0], p[1], p[2]));  
1075         gl.uniform1f(this.mNearRef, n);  
1076         gl.uniform1f(this.mFarRef, f);  
1077         gl.uniform1f(this.mIntensityRef, aLight.getIntensity());  
1078     }  
1079 }
```

- 1080 4. Define a simple function to update the on/off status of the light in
1081 the array of the `light_fs` fragment shader:

```
1082 switchOffLight() {  
1083     let gl = glSys.get();  
1084     gl.uniform1i(this.mIsOnRef, false);  
1085 }
```

Note that the ShaderLightAt class is defined for loading a light to a specific array element in the GLSL fragment shader. This is an internal engine operation. There is no reason for the game programmer to access this class, and thus, the engine access file, index.js, should not be modified to forward the definition of this class.

Modifying the LightShader Class

You must now modify the LightShader object to properly handle the communication between the Light object and the array of lights in the light_fs fragment shader:

1. Begin by editing the light_shader.js file, importing ShaderLightAt, and *removing* the _loadToShader() function. The actual loading of light information to the light_fs fragment shader is now handled by the newly defined ShaderLightAt objects.

```

import ShaderLightAt from "./shader_light_at.js";

```

2. Modify the constructor to define mLights, which is an array of ShaderLightAt objects to correspond to the uLights array defined in the light_fs fragment shader. It is important to note that the mLights and uLights arrays must be the exact same size.

```

constructor(vertexShaderPath, fragmentShaderPath) {
    // Call super class constructor
    super(vertexShaderPath, fragmentShaderPath);

    this.mLights = null; // lights from the Renderable
    this.mCamera = null; // camera to draw, need for WC to DC xform

    //*****WARNING*****
    // MUST correspond to GLSL uLight[] array size (for LightFS.gls)
    //*****WARNING*****
    this.kGLSLuLightArraySize = 4; // must be the same as LightFS.gls
    this.mShaderLights = [];
    let i, ls;
    for (i = 0; i < this.kGLSLuLightArraySize; i++) {
        ls = new ShaderLightAt(this.mCompiledShader, i);
        this.mShaderLights.push(ls);
    }
}

```

- 1118 3. Modify the `activate()` function to iterate and load the contents
1119 of each `ShaderLightAt` object to the `light_fs` shader by calling
1120 the corresponding `loadToShader()` function. Recall that the
1121 GLSL fragment shader requires the for-loop control variable to
1122 be a constant. This implies that all elements of the `uLights` array
1123 will be processed on each `light_fs` invocation. For this reason,
1124 it is important to ensure all unused lights are switched off. This is
1125 ensured by the last while loop in the following code:

```
1126 activate(pixelColor, trsMatrix, cameraMatrix) {  
1127     // first call the super class's activate  
1128     super.activate(pixelColor, trsMatrix, cameraMatrix);  
1129  
1130     // now push the light information to the shader  
1131     let numLight = 0;  
1132     if (this.mLights !== null) {  
1133         while (numLight < this.mLights.length) {  
1134             this.mShaderLights[numLight].loadToShader(  
1135                             this.mCamera, this.mLights[numLight]);  
1136             numLight++;  
1137         }  
1138     // switch off the left over ones.  
1139     while (numLight < this.kGLSLuLightArraySize) {  
1140         this.mShaderLights[numLight].switchOffLight(); // off the un-use  
1141         numLight++;  
1142     }  
1143 }
```

1144 4. Rename the `setCameraAndLight()` function to
1145 `setCameraAndLights()`; in addition to setting the corresponding
1146 variables, check to ensure that the light array size is not greater
1147 than the defined array size in the `light_fs` fragment shader.
1148 Lastly, remember to update the corresponding function name in
1149 `sprite_shader.js`.

```

setCameraAndLights(c, l) { 1150
    this.mCamera = c; 1151
    this.mLights = l; 1152
    if (this.mLights.length > this.kGLSLuLightArraySize) 1153
        throw new Error ("Error: " ...); 1154
}

```

Modifying the LightRenderable Class

You can now modify the LightRenderable class to support multiple light sources:

1. In the LightRenderable constructor, replace the single light reference variable with an array:

```

constructor(myTexture) { 1160
    super(myTexture); 1161
    super._setShader(defaultShaders.getLightShader()); 1162
    // the light sources 1163
    this.mLights = []; 1164
}

```

2. Make sure to update the draw function to reflect the change to multiple light sources:

```

draw(camera) { 1168
    this.mShader.setCameraAndLights(camera, this.mLights); 1169
    super.draw(camera); 1170
}

```

3. Define the corresponding accessor functions for the light array:

```

getLightAt(index) { return this.mLights[index]; } 1173
addLight(l) { this.mLights.push(l); } 1174

```

Testing the Light Sources with MyGame

With proper integration for multiple lights support in the engine, you can now modify MyGame to test your implementation and examine the results. In addition to adding multiple lights to the scene, you will be adding the ability to control the properties of

CHAPTER 8 IMPLEMENTING ILLUMINATION AND SHADOW

1179 each light. In order to maintain readability, you will divide the light instantiation and
1180 controls into separate files. To avoid redundancy and repetitive code listings, the details
1181 to the straightforward implementations are not shown.

- 1182 1. Modify the `my_game_main.js` file in the `src/my_game` folder to
1183 reflect the changes to the constructor, initialize function, draw
1184 function, and update function. All these changes revolve around
1185 handling multiple lights through a light set.
- 1186 2. In the `src/my_game` folder, create the new file `my_game_lights.js` to import `MyGame` class from `my_game_main.js` and to add
1187 functionality to instantiate and initialize the lights.
- 1188 3. In the `src/my_game` folder, create the new file `my_game_light_control.js` to import from `my_game_lights.js` and to continue to
1189 add controls of the lights to `MyGame`.
- 1190 4. Modify `my_game.js` to import from `my_game_light_control.js`
1191 ensuring access to all of the newly defined functionality.

1194 Observations

1195 Run the project to examine the implementation. Try selecting the lights with the
1196 0, 1, 2, and 3 keys and toggling the selected light on/off. Notice that the game
1197 programmer has control over which light illuminates which of the objects: all lights
1198 illuminate the background, while the hero is illuminated only by lights 0 and 3, the
1199 left minion is illuminated only by lights 1 and 3, and the right minion is illuminated
1200 only by lights 2 and 3.

1201 Move the Hero object with the WASD keys to observe how the illumination changes
1202 as it is moved through the near and far radii of light source 0. With light source 0 selected
1203 (type 0), press the C key to increase the near radius of the light. Notice that as the
1204 near radius approaches the value of the far, the illuminated circle boundary edge also
1205 becomes sharper. Eventually, when near radius is greater than far radius, you can once
1206 again observe the sudden brightness change at the boundary. You are observing the
1207 violation of the implicit assumption of the underlying illumination model that the near is
1208 always less than the far radius. This exact situation can be created by decreasing the far
1209 radius with the N key.

You can move the light sources with the arrow keys to observe the additive property of lights. Experiment with changing the light source's z position and its near/far values to observe how similar illumination effects can be accomplished with different z/near/far settings. In particular, try adjusting light intensities with the K/L keys to observe the effects of oversaturation and barely noticeable lighting. You can continue to press the L key till the intensity becomes negative to create a source that removes color from the scene. The two constant color squares are in the scene to confirm that nonilluminated objects can still be rendered.

1210
1211
1212
1213
1214
1215
1216
1217

Diffuse Reflection and Normal Mapping

1218

You can now place or move many light sources and control the illumination or shading at targeted regions. However, if you run the previous project and move one of the light sources around, you may notice some peculiar effects. Figure 8-7 highlights these effects by comparing the illumination results from the previous project on the left to an illumination that you probably expect on the right. Now, refer to the image on the left. First, take note of the general uniform lighting within the near cutoff region where the expected brighter spot around the position of the point light source cannot be observed. Second, examine the vertical faces of the geometric block and take note of the bright illumination on the bottom face that is clearly behind, or *pointing away from*, the light source. Both of these peculiarities are absent from the right image in Figure 8-7.

1219
1220
1221
1222
1223
1224
1225
1226
1227
1228

Although visually odd, results from the left image of Figure 8-7 are to be expected in a 2D world. The vertical faces are only artist renditions, and your illumination calculation does not consider the geometric contours suggested by the image content. This restriction of illumination in a flat 2D world is remedied in this section with the introduction of diffuse reflection and normal mapping to approximate normal vectors of surfaces.

1229
1230
1231
1232
1233
1234

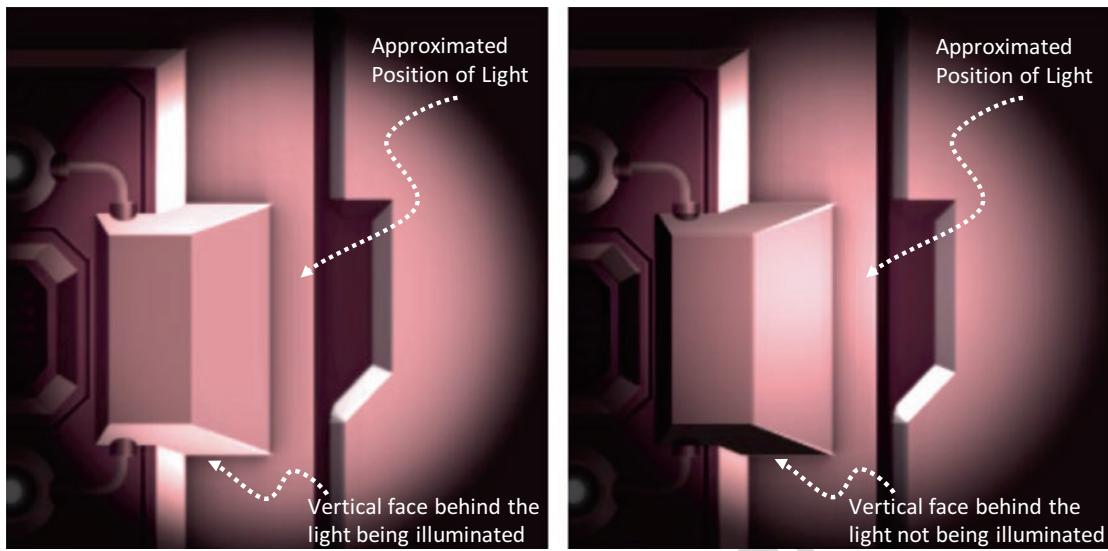


Figure 8-7. Left: from previous project. Right: expected illumination

1235 As illustrated by the left drawing in Figure 8-8, a surface normal vector, a surface
 1236 normal, or a normal vector is the vector that is perpendicular to a given surface element.
 1237 The right drawing of Figure 8-8 shows that in 3D space, the surface normal vectors of an
 1238 object describe the shape or contour of the object.

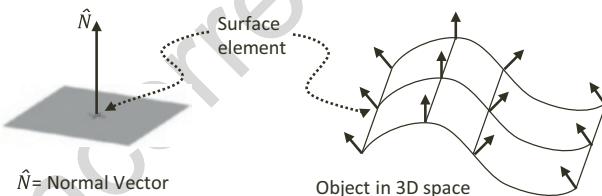
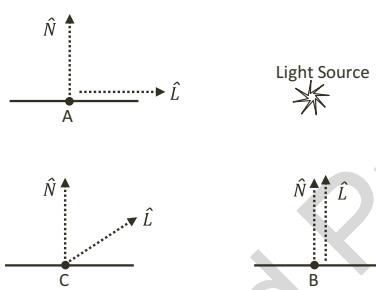


Figure 8-8. Surface normal vectors of an object

1239 A human's observation of light illumination is the result of visible energy from
 1240 light sources reflecting off object surfaces and reaching the eyes. A diffuse, matte, or
 1241 Lambertian surface reflects light energy uniformly in all directions. Examples of diffuse
 1242 surfaces include typical printer papers or matte-painted surfaces. Figure 8-9 shows
 1243 a light source illuminating three diffuse surface element positions, A, B, and C. First,
 1244 notice that the direction from the position being illuminated toward the light source is
 1245 defined as the light vector, \hat{L} , at the position. It is important to note that the direction of
 1246 the \hat{L} vector is always toward the light source and that this is a normalized vector with a
 1247 magnitude of 1.

Figure 8-9 also illustrates the diffuse illumination, or magnitude of diffuse reflection, with examples. Position A cannot receive any energy from the given light source because its normal vector, \hat{N} , is perpendicular to its light vector \hat{L} , or $\hat{N} \cdot \hat{L} = 0$. Position B can receive all the energy because its normal vector is pointing in the same direction as its light vector, or $\hat{N} \cdot \hat{L} = 1$. In general, as exemplified by position C, the proportion of light energy received and reflected by a diffuse surface is proportional to the cosine of the angle between its normal and the light vector, or $\hat{N} \cdot \hat{L}$. In an illumination model, the term with the $\hat{N} \cdot \hat{L}$ computation is referred to as the diffuse, or Lambertian, component.

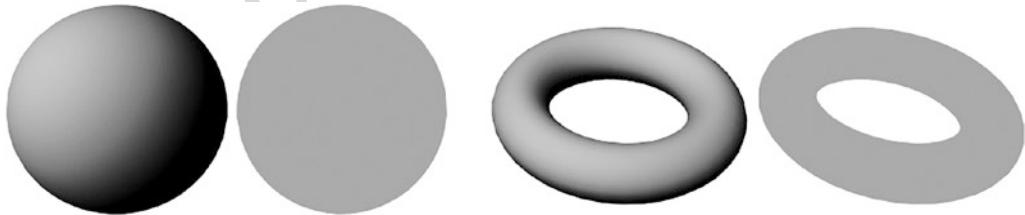


1248
1249
1250
1251
1252
1253
1254
1255

this figure will be printed in b/w

Figure 8-9. Normal and light vectors and diffuse illumination

The human vision system deduces 3D geometric shape contours based significantly on $\hat{N} \cdot \hat{L}$, or the diffuse component. For example, Figure 8-10 shows a sphere and torus (doughnut shape object) with (the left images) and without (the right images) the corresponding diffuse components. Clearly, in both cases, the 3D contour of the objects is captured by the left versions of the image with the diffuse component.



1256
1257
1258
1259
1260

this figure will be printed in b/w

Figure 8-10. Examples of 3D objects with and without diffuse component

In a 2D world, as in the case of your game engine, all objects are represented as 2D images, or textures. Since all objects are 2D textured images defined on the xy plane, the normal vectors for all the objects are the same: a vector in the z direction. This lack of distinct normal vectors for objects implies that it is not possible to compute a distinct

1261
1262
1263
1264

CHAPTER 8 IMPLEMENTING ILLUMINATION AND SHADOW

1265 diffuse component for objects. Fortunately, similar to how texture mapping addresses
1266 the limitation of each geometry having only a single color, normal mapping can resolve
1267 the issue of each geometry having only a single normal vector.

1268 Figure 8-11 shows the idea behind normal mapping where in addition to the
1269 color texture image, a corresponding normal texture image is required. The left image
1270 of Figure 8-11 is a typical color texture image, and the two right images are zoomed
1271 images of the highlighted square on the left image. Notice once again that two images
1272 are involved in normal mapping: the color texture image where the RGB channels of
1273 the texture record the color of objects (bottom of the right image of Figure 8-11) and
1274 a corresponding normal texture image where the RGB channels record the x, y, and z
1275 values of the normal vector for the corresponding object in the color texture (top of the
1276 right image).

this figure will be printed in b/w

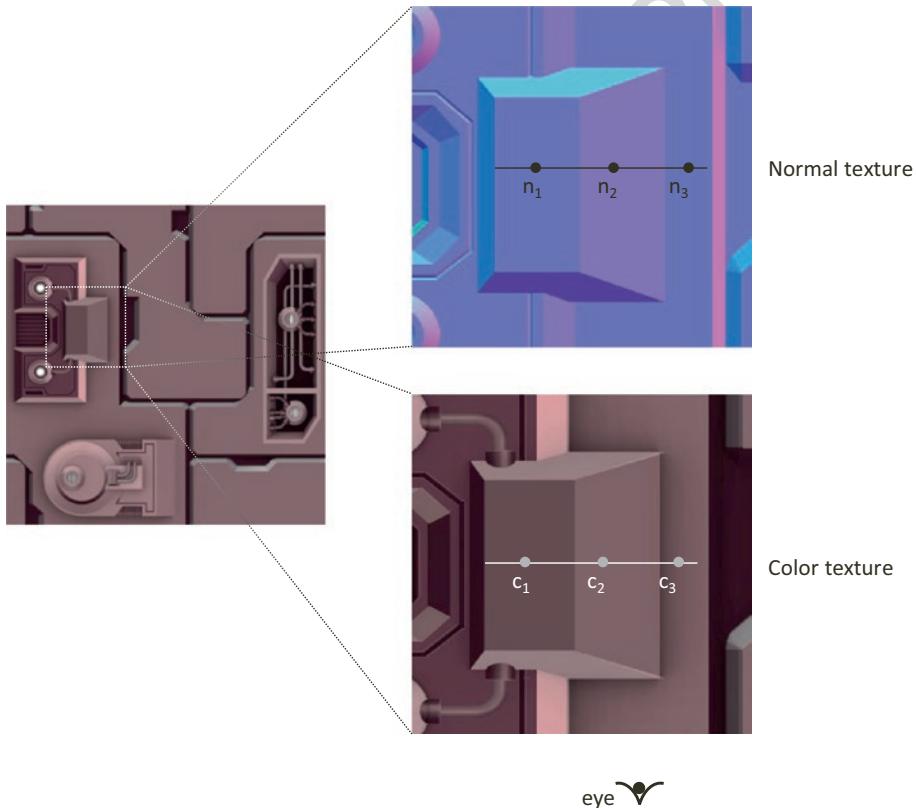
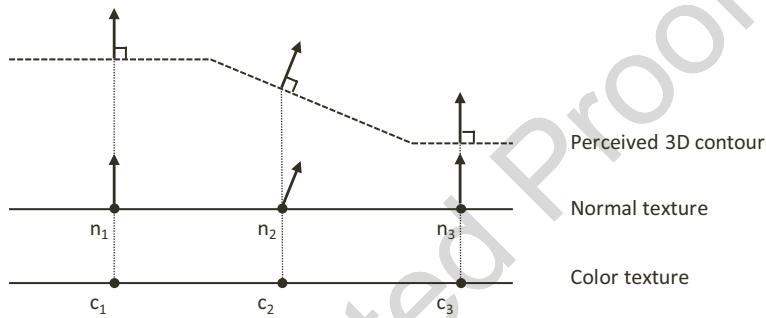


Figure 8-11. Normal mapping with two texture images: the normal and the color texture

Figure 8-12 captures the view of the three corresponding positions labeled on the right images of Figure 8-11, the positions n_1 , n_2 , and n_3 on the normal texture and the corresponding positions c_1 , c_2 , and c_3 on the color texture, to illustrate the details of normal mapping. The bottom layer of Figure 8-12 shows that the color texture records colors and the colors c_1 , c_2 , and c_3 are sampled at those three positions. The middle layer of Figure 8-12 shows that the RGB components of the normal texture record the normal vector xyz values of objects at the corresponding color texture positions. The top layer of Figure 8-12 shows that when illuminated by a light source, with the $\hat{N} \cdot \hat{L}$ term properly computed and displayed, the human vision system will perceive a sloped contour.

1277
1278
1279
1280
1281
1282
1283
1284
1285



this figure will be printed in b/w

Figure 8-12. Normal mapping with two texture images: the normal and the color texture

In summary, a normal texture map or a normal map is a texture map that stores normal vector information rather than the usual color information. Each texel of a normal map encodes the xyz values of a normal vector in the RGB channels. In lieu of displaying the normal map texels as you would with a color texture, the texels are used purely for calculating how the surface would interact with light. In this way, instead of a constant normal vector pointing in the z direction, when a square is normal mapped, the normal vector of each pixel being rendered will be defined by texels from the normal map and can be used for computing the diffuse component. For this reason, the rendered image will display contours that resemble the shapes encoded in the normal map.

1286
1287
1288
1289
1290
1291
1292
1293
1294
1295

In the previous project, you expanded the engine to support multiple light sources. In this section, you will define the `IllumShader` class to generalize a `LightShader` to support the computation of the diffuse component based on normal mapping.

1296
1297
1298

1299 The Normal Maps and Illumination Shaders Project

1300 This project demonstrates how to integrate normal mapping into your game engine and
 1301 use the results to compute the diffuse component of objects. You can see an example
 1302 of this project running in Figure 8-13. The source code of this project is located in the
 1303 chapter8/8.4.normal_maps_and_illumination_shaders folder.

this figure will be printed in b/w

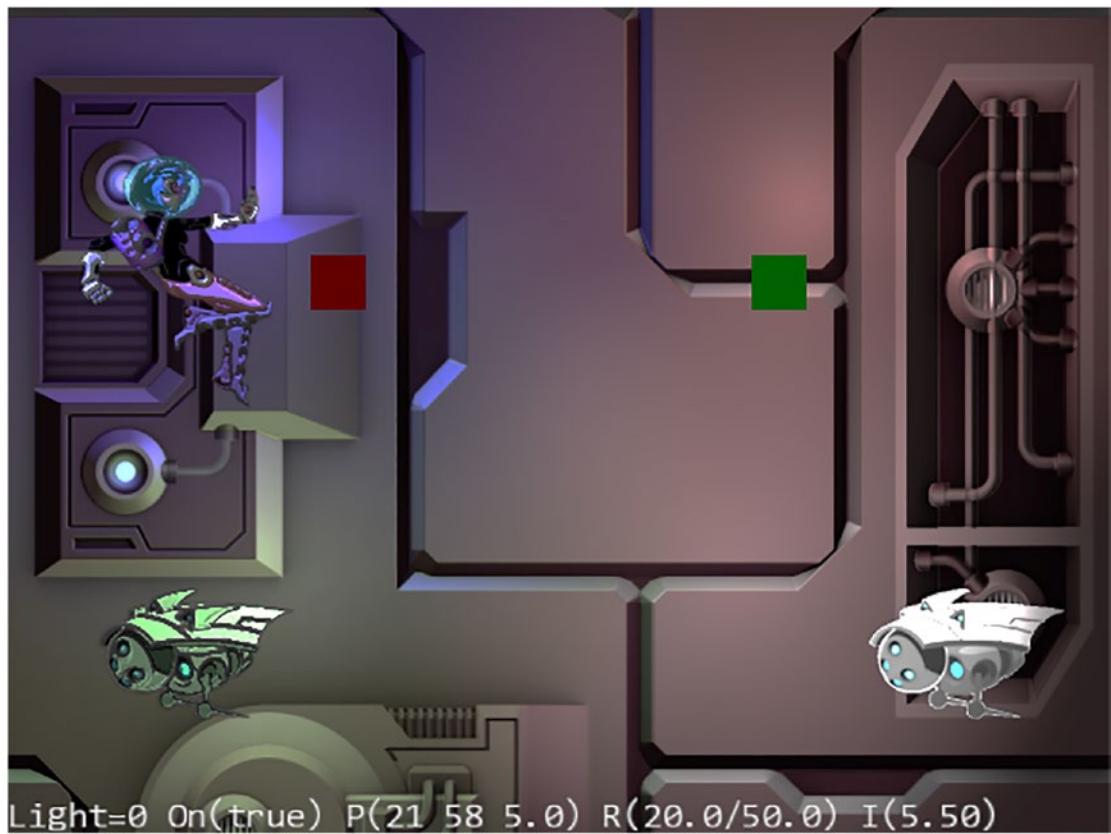


Figure 8-13. Running the Normal Maps and Illumination Shaders project

1304 The controls of the project are identical to the previous project:

- 1305 • **WASD keys:** Move the hero character on the screen
- 1306 • **Number keys 0, 1, 2, and 3:** Select the corresponding light source
- 1307 • **Arrow keys:** Move the currently selected light
- 1308 • **Z/X key:** Increases/decreases the light z position

- **C/V and B/N keys:** Increases/decreases the near and far cutoff distances of the selected light 1309
1310
- **K/L key:** Increases/decreases the intensity of the selected light 1311
- **H key:** Toggles the selected light on/off 1312

The goals of the project are as follows:

- To understand and work with normal maps 1314
- To implement normal maps as textures in the game engine 1315
- To implement GLSL shaders that support diffuse component illumination 1316
1317
- To examine the diffuse, $\hat{N} \cdot \hat{L}$, component in an illumination model 1318

You can find the following external resource files in the assets folder. The fonts folder contains the default system fonts, two texture images, and two corresponding normal maps for the texture images, `minion_sprite.png` and `bg.png`, and the corresponding normal maps, `minion_sprite_normal.png` and `bg_normal.png`. As in previous projects, the objects are sprite elements of `minion_sprite.png`, and the background is represented by `bg.png`.

Note The `minion_sprite_normal.png` normal map is generated 1325
algorithmically from <http://cpetry.github.io/NormalMap-Online/> based 1326
on the `minion_sprite.png` image. 1327

Creating the GLSL Illumination Fragment Shader

As with the previous projects, your normal map integration will begin with the implementation of the GLSL shader. Note that this new shader will be remarkably similar to your `light_fs.glsl` but with the inclusion of normal mapping and diffuse computation support. To ensure the support for simple lighting without normal mapping, you will create a new GLSL fragment shader.

CHAPTER 8 IMPLEMENTING ILLUMINATION AND SHADOW

1334 1. Begin by copying from `light_fs.glsl` and pasting to a new file,
 1335 `illum_fs.glsl`, in the `src/glsl_shaders` folder.

1336 2. Edit the `illum_fs.glsl` file and add a `sampler2D` object,
 1337 `uNormalSampler`, to sample the normal map:

```
1338 uniform sampler2D uSampler;  

1339 uniform sampler2D uNormalSampler;  

1340 ... identical to the variables declare in light_fs.glsl ...
```

1341 3. Modify the `LightEffect()` function to receive a normal vector
 1342 parameter, `N`. This normal vector `N` is assumed to be normalized
 1343 with a magnitude of 1 and will be used in the diffuse component
 1344 $\hat{N} \cdot \hat{L}$ computation. Enter the code to compute the \hat{L} vector,
 1345 remember to normalize the vector, and use the result of $\hat{N} \cdot \hat{L}$ to
 1346 scale the light strength accordingly.

```
1347 vec4 LightEffect(Light lgt, vec3 N) {  

  1348     vec4 result = vec4(0);  

  1349     float strength = 0.0;  

  1350     vec3 L = lgt.Position.xyz - gl_FragCoord.xyz;  

  1351     float dist = length(L);  

  1352     if (dist <= lgt.Far) {  

  1353         if (dist <= lgt.Near) {  

  1354             ... identical to previous code ...  

  1355         }  

  1356         L = L / dist; // To normalize L  

  1357         // Not calling normalize() function to avoid re-computing  

  1358         // "dist". This is computationally more efficient.  

  1359         float NdotL = max(0.0, dot(N, L));  

  1360         strength *= NdotL;  

  1361     }  

  1362     result = strength * lgt.Intensity * lgt.Color;  

  1363     return result;  

  1364 }
```

4. Edit the `main()` function to sample from both the color texture with `uSampler` and the normal texture with `uNormalSampler`. Remember that the normal map provides you with a vector that represents the normal vector of the surface element at that given position. Because the xyz normal vector values are stored in the 0 to 1 RGB color format, the sampled normal map results must be scaled and offset to the -1 to 1 range. In addition, recall that texture uv coordinates can be defined with the v direction increasing upward or downward. In this case, depending on the v direction of the normal map, you may also have to flip the y direction of the sampled normal map values. The normalized normal vector, `N`, is then passed on to the `LightEffect()` function for the illumination calculations.

```

void main(void)  {
    // simple tint based on uPixelColor setting
    vec4 textureMapColor = texture2D(uSampler, vTexCoord);
    vec4 normal = texture2D(uNormalSampler, vTexCoord); // same UV
    vec4 normalMap = (2.0 * normal) - 1.0;

    //
    // normalMap.y = -normalMap.y; // flip Y
    // depending on the normal map you work with,
    // this may or may not be flipped
    //

    vec3 N = normalize(normalMap.xyz);

    vec4 lgtResult = uGlobalAmbientColor * uGlobalAmbientIntensity;
    // now decide if we should illuminate by the light
    if (textureMapColor.a > 0.0) {
        for (int i=0; i<kGLSLuLightArraySize; i++) {
            if (uLights[i].IsOn) {
                lgtResult += LightEffect(uLights[i], N);
            }
        }
    }
    ... identical to previous code ...
}

```

1399 **Note** Normal maps can be created in a variety of different layouts where x
1400 or y might need to be flipped in order to properly represent the desired surface
1401 geometries. It depends entirely upon the tool or artist that created the map.

1402 **Defining the IllumShader Class**

1403 With the `Illum_fs` fragment shader supporting normal maps, you can create the
1404 JavaScript `IllumShader` class to interface with it:

1405 1. In the `src/engine/shaders` folder, create `illum_shader.js`,
1406 and define `IllumShader` to be a subclass of `LightShader` to take
1407 advantage of the functionality related to light sources. In the
1408 constructor, define a variable, `mNormalSamplerRef`, to maintain
1409 the reference to the normal sampler in the `illum_fs` fragment
1410 shader. Don't forget to export the class.

```
1411 import LightShader from "./light_shader.js";
1412 import * as glSys from "../core/gl.js";

1413 class IllumShader extends LightShader {
1414     constructor(vertexShaderPath, fragmentShaderPath) {
1415         super(vertexShaderPath, fragmentShaderPath);
1416         let gl = glSys.get();
1417         // reference to the normal map sampler
1418         this.mNormalSamplerRef = gl.getUniformLocation(
1419             this.mCompiledShader, "uNormalSampler");
1420     }
1421     ... implementation to follow ...
1422 }
1423 export default IllumShader;
```

1424 2. Override and extend the `activate()` function to binding the
1425 normal texture sampler reference to WebGL texture unit 1. You
1426 may recall from Chapter 5 that `TextureShader` binds the color
1427 texture sampler to texture unit 0. By binding normal mapping to

texture unit 1, the WebGL texture system can work concurrently with two active textures: units 0 and 1. As will be discussed in the next subsection, it is important to configure WebGL, via the texture module, to activate the appropriate texture units for the corresponding purpose: color vs. normal texture mapping.	1428 1429 1430 1431 1432
activate(pixelColor, trsMatrix, cameraMatrix) { // first call the super class's activate super.activate(pixelColor, trsMatrix, cameraMatrix); let gl = glSys.get(); gl.uniform1i(this.mNormalSamplerRef, 1); // binds to texture unit 1 // do not need to set up texture coordinate buffer // as we are going to use the ones from the sprite texture // in the fragment shader	1433 1434 1435 1436 1437 1438 1439 1440
}	1441

Note WebGL supports simultaneous activation of multiple texture units during rendering. Depending on the GPU, a minimum of eight texture units can be active simultaneously during a single rendering pass. In this book, you will activate only two of the texture units during rendering: one for color texture and the other for normal texture.

Modifying the Texture Module

So far, you have been binding the color texture map to WebGL texture unit 0. With the addition of the normal texture, the binding to the unit of WebGL texture system must now be parameterized. Fortunately, this is a straightforward change.

Modify the texture module by opening `texture.js` in the `src/engine/resources` folder. Edit the `activate()` function to accept a second parameter, the WebGL texture unit to bind to. Notice that this is an optional parameter with the default value set to texture unit 0. This is such that no changes are required for any of the existing calls to the `activate()` function.

```

1456 function activate(textureName, textureUnit = glSys.get().TEXTURE0) {
1457     let gl = glSys.get();
1458     let texInfo = get(textureName);
1459
1460     // Binds texture reference to current webGL texture functionality
1461     gl.activeTexture(textureUnit); // activate the WebGL texture unit
1462     gl.bindTexture(gl.TEXTURE_2D, texInfo.mGLTexID);
1463
1464     ... identical to previous code ...
1465 }
```

Creating the IllumRenderable Class

1465 You can now define the illumination Renderable class to leverage the newly created
 1466 illumination shader:

1467 1. Begin by creating `illum_renderable.js` in the `src/engine/renderables` folder, defining the `IllumRenderable` class to
 1468 subclass from `LightRenderable`, and initializing a `mNormalMap`
 1469 instance variable to record the normal map ID. The
 1470 `IllumRenderable` object works with two texture maps: `myTexture`
 1471 for color texture map and `myNormalMap` for normal mapping. Note
 1472 that these two texture maps share the same texture coordinates
 1473 defined in `mTexCoordBuffer` in the `SpriteShader`. This sharing
 1474 of texture coordinate implicitly assumes that the geometry of the
 1475 object is depicted in the color texture map and the normal texture
 1476 map is derived to capture the contours of the object, which is
 1477 *almost* always the case. Lastly, don't forget to export the class.

```

1479 import * as texture from "../resources/texture.js";
1480 import * as glSys from "../core/gl.js";
1481 import LightRenderable from "./light_renderable.js";
1482 import * as defaultShaders from "../core/shader_resources.js";
1483
1484 class IllumRenderable extends LightRenderable {
1485     constructor(myTexture, myNormalMap) {
1486         super(myTexture);
1487         super._setShader(defaultShaders.getIllumShader());
```

```

// here is the normal map resource id           1487
this.mNormalMap = myNormalMap;                 1488

// Normal map texture coordinate is same as sprite sheet   1489
// This means, the normal map MUST be based on the sprite sheet 1490
}
... implementation to follow ...                1491
}

export default IllumRenderable;               1493

```

Note Once again, it is important to reiterate that the normal texture map is an image that must be created explicitly by an artist or algorithmically by an appropriate program. Using a regular color texture map image as a normal texture map will not work in general.

2. Next, override the draw() function to activate the normal map before calling the draw() method of the super class. Notice the second argument of the texture.activate() function call where the WebGL texture unit 1 is explicitly specified. In concert with IllumShader linking uNormalSampler to WebGL texture unit 1 and illum_fs sampling the uNormalSampler as a normal map, your engine now supports proper normal mapping.

```

draw(camera) {                                1506
    texture.activate(this.mNormalMap, glSys.get().TEXTURE1); 1507
    // Here the normal map texture coordinate is copied from those of 1508
    // the corresponding sprite sheet                         1509
    super.draw(camera);                                1510
}

```

Lastly, remember to update the engine access file, index.js, to forward the newly defined functionality to the client.

1514 Defining a Default IllumShader Instance

Similar to all other shaders in the engine, a default instance of the `IllumShader` must be defined to be shared. The code involved in defining the default `IllumShader` instance is identical to that of `LightShader` presented earlier in this chapter, with the straightforward exception of substituting the corresponding variable names and data type. Please refer to the “Defining a Default LightShader Instance” subsection and the `shader_resources.js` source code file in the `src/engine/core` folder for details.

1521 Testing the Normal Map

1522 Testing the newly integrated normal map functionality must include the verification
1523 that the non-normal mapped simple color texture is working correctly. To accomplish
1524 this, the background, hero, and left minion will be created as the newly defined
1525 `IllumRenderable` object, while the right minion will remain a `LightRenderable` object.

1526 Modifying the Hero and the Minion

1527 The Hero and Minion objects should be instantiated as the newly defined
1528 **IllumRenderable** object:

1. Edit `hero.js` in `src/my_game/objects` to modify the constructor of the `Hero` class to instantiate the game object with an `IllumRenderable`:

```
1532 constructor(spriteTexture, normalMap) {  
1533     super(null);  
1534     this.kDelta = 0.3;  
1535     this.mRenderComponent = new engine.IllumRenderable(  
1536                                         spriteTexture, normalMap);  
1537     this.mRenderComponent.setColor([1, 1, 1, 0]);  
1538     ... identical to previous code ...  
1539 }
```

2. In the same folder, edit `minion.js` to modify the constructor of `Minion` class to conditionally instantiate the game object with either a `LightRenderable` or an `IllumRenderable` when the normal texture map is present:

```

constructor(spriteTexture, normalMap, atX, atY) { 1544
    super(null); 1545
    this.kDelta = 0.2; 1546

    if (normalMap === null) { 1547
        this.mRenderComponent = new engine.LightRenderable( 1548
            spriteTexture); 1549
    } else { 1550
        this.mRenderComponent = new engine.IllumRenderable( 1551
            spriteTexture, normalMap); 1552
    }
} 1553

... identical to previous code ... 1554
} 1555

```

Modifying MyGame

You can now modify MyGame to test and display your implementation of the illumination shader. Modify the `my_game_main.js` file in the `src/my_game` folder to load and unload the new normal maps and to create the Hero and Minion objects with the normal map files. As previously, the involved changes are straightforward and relatively minimal; as such, the details are not shown here.

Observations

With the project now completed, you can run it and check your results to observe the effects of diffuse illumination. Notice that the Hero, left Minion, and the background objects are illuminated with a diffuse computation and appear to provide more depth from the lights. There is much more variation of colors and shades across these objects.

You can verify that the peculiar effects observed on the left image of Figure 8-7 is resolved. For a clearer observation, switch off all other lights (type the light number followed the H key) except leaving light 2 switched on. Now, move the light position (with the arrow keys) to illuminate the geometric block behind the Hero character; you can move it away with the WASD keys. Verify that you are viewing similar results to those in the right image of Figure 8-7. You should be able to clearly observe the brightest spot that corresponds to the point light position. Additionally, take note that the bottom face on the block is only illuminated when the light position is *in front* of the face or when the diffuse term, $\hat{N} \cdot \hat{L}$, is positive.

1576 In general, as you move the light source, observe faces with vertical orientations, for
 1577 example, the side faces of the geometric block or gaps. As the light position moves across
 1578 such a boundary, the sign of the $\hat{N} \cdot \hat{L}$ term would flip, and the corresponding surface
 1579 illumination would undergo drastic changes (from dark to lit, or vice versa). For a more
 1580 dramatic result, lower the z height of the light (with the X key) to a value lower than 5.
 1581 With the normal map and diffuse computation, you have turned a static background
 1582 image into a background that is defined by complex 3D geometric shapes. Try moving
 1583 the other light sources and observe the illumination changes on all the objects as the
 1584 light sources move across them.

1585 Lastly, the slightly pixelated and rough appearances of the Hero and left Minion attest
 1586 to the fact that the normal maps for these objects are generated algorithmically from the
 1587 corresponding color images and that the maps are not created by an artist.

1588 Specular Reflection and Materials

1589 The diffuse lighting you have implemented is suitable for simulating the illumination
 1590 of matte surfaces such as typical printer papers, many painted interior walls, or even
 1591 a traditional blackboard. The Phong illumination model extends this simple diffuse
 1592 lighting by introducing a specular term to simulate the reflection of the light source
 1593 across a shiny surface. Figure 8-14 shows an example of three spheres, a simple matte
 1594 sphere, a sphere with moderate specular highlights, and a highly polished sphere. The
 1595 highlights on the two spheres to the right are results of the Phong specular term.

this figure will be printed in b/w

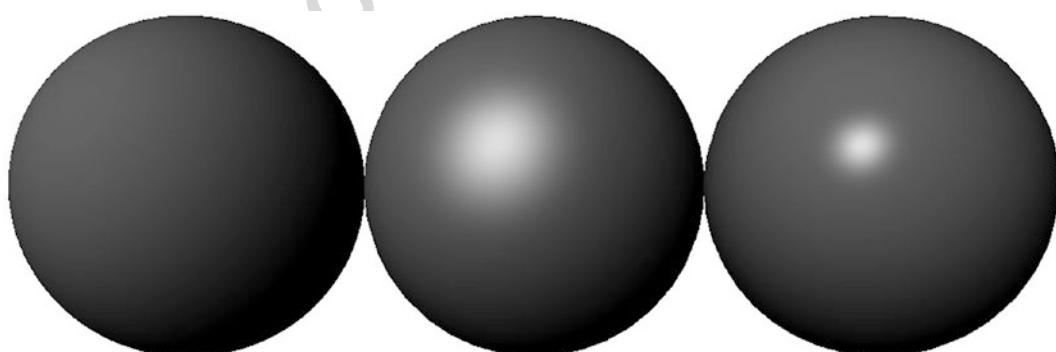
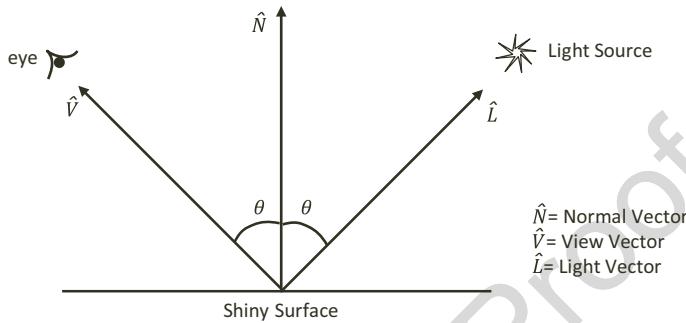


Figure 8-14. Specularity and shininess of objects

Figure 8-15 illustrates that given a shiny or reflective surface like a polished floor or a polished plastic, the reflection of the light source will be visible when the eye, or the camera, is in the reflection direction of the light source. This reflection of the light source across a shiny surface is referred to as *specular reflection*, *specular highlight*, or *specularity*.

1596
1597
1598
1599
1600

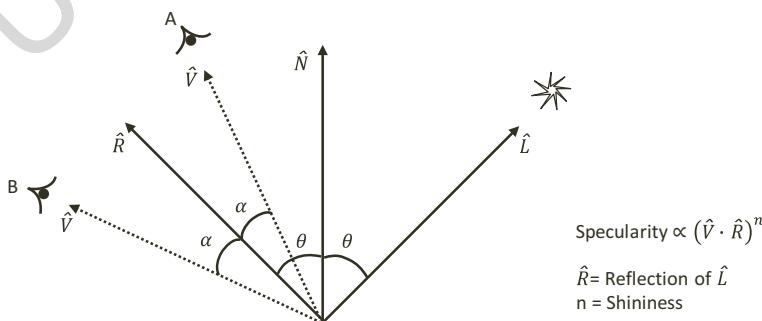


this figure will be printed in b/w

Figure 8-15. Specularity: the reflection of the light source

From real-life experience, you know that specular highlights are visible even when the eye's viewing direction is not perfectly aligned with the reflection direction of the light source. As illustrated in Figure 8-16, where the \hat{R} vector is the reflection direction of the light vector \hat{L} , the specular highlight on an object is visible even when the viewing direction \hat{V} is not perfectly aligned with the \hat{R} vector. Real-life experience also informs you that the further away \hat{V} is from \hat{R} , or the larger the angle- α is, the less likely you will observe the light reflection. In fact, you know that when α is zero, you would observe the maximum light reflection, and when α is 90° or when \hat{V} and \hat{R} are perpendicular, you would observe zero light reflection.

1601
1602
1603 **BU4**
1604
1605
1606
1607
1608
1609



this figure will be printed in b/w

Figure 8-16. The Phong specularity model

The Phong illumination model simulates the characteristics of specularity with a $(\hat{V} \cdot \hat{R})^n$ term. When \hat{V} and \hat{R} are aligned, or when $\alpha=0^\circ$, the specularity term evaluates to 1, and the term drops off to 0 according to the cosine function when the separation between \hat{V} and \hat{R} increases to 90° or when $\alpha=90^\circ$. The power n , referred to as shininess, describes how rapidly the specular highlight rolls off as α increases. The larger the n value, the faster the cosine function decreases as α increases, the faster the specular highlight drops off, and the glossier the surface would appear. For example, in Figure 8-14, the left, middle, and right spheres have corresponding n values of 0, 5, and 30.

While the $(\hat{V} \cdot \hat{R})^n$ term models specular highlight effectively, the cost involved in computing the \hat{R} vector for every shaded pixel can be significant. As illustrated in Figure 8-17, \hat{H} , the halfway vector, is defined as the average of the \hat{L} and \hat{V} vectors. It is observed that β , the angle between the \hat{N} and \hat{H} , can also be used to characterize specular reflection. Though slightly different, $(\hat{N} \cdot \hat{H})^n$ produces similar results as $(\hat{V} \cdot \hat{R})^n$ with less per-pixel computation cost. The halfway vector will be used to approximate specularity in your implementation.

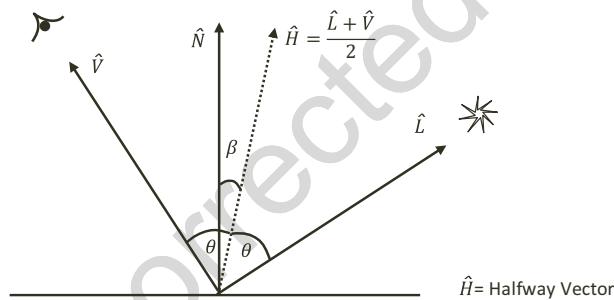


Figure 8-17. The halfway vector

As illustrated in Figure 8-18, the variation of the Phong illumination model that you will implement consists of simulating the interaction of three participating elements in the scene through three distinct terms. The three participating elements are the global ambient lighting, the light source, and the material property of the object to be illuminated. The previous examples have explained the first two: the global ambient lighting and the light source. In this way, to support the Phong illumination model, the material property of an object can be represented by K_a , K_d , K_s , and n . These stand for three colors, representing the ambient, diffuse, and specular reflectivity, and a floating-point number representing the shininess of an object. With the global ambient

light intensity, I_a , and color, C_a , and the light source intensity, I_L , and color, C_L , the three terms of the Phong illumination model are as follows:

- **The ambient term:** $I_a C_a K_a$
- **The diffuse term:** $I_L C_L K_d (\hat{N} \cdot \hat{L})$
- **The specular term:** $I_L C_L K_s (\hat{N} \cdot \hat{H})^n$

Note that the first two terms, the ambient and diffuse terms, have been covered in the previous examples. The `illum_fs` GLSL fragment shader from the previous example implements these two terms with a light distance attenuation and without the K_a and K_d material properties. This project guides you to build the support for per-object material property and complete the Phong illumination model implementation in the `illum_fs` GLSL shader with the engine support in the `IllumShader/IllumRenderable` object pair.

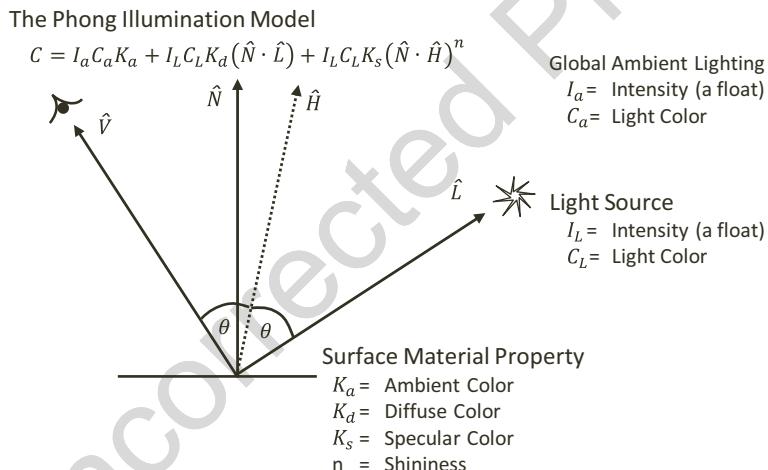


Figure 8-18. The Phong illumination model

Integration of Material in the Game Engine and GLSL Shaders

To implement the Phong illumination model, a `Material` class that encapsulates the surface material property in Figure 8-18 must be defined and referenced by each `IllumRenderable` object that is to be shaded by the corresponding `illum_fs` fragment shader. Figure 8-19 illustrates that in your implementation, a new `ShaderMaterial` object will be defined and referenced in the `IllumShader` to load the content of the `Material` object to the `illum_fs` GLSL fragment shader.

1634

1635

1636

1637

1638

1639

1640

1641

1642

1643

1644

this figure will be printed in b/w

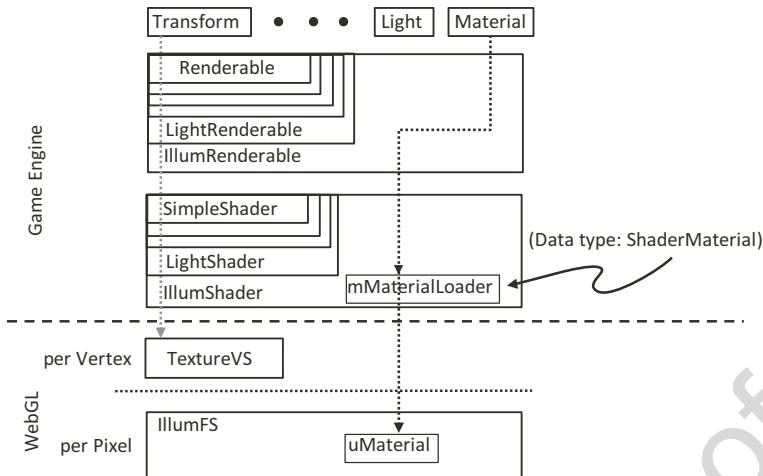


Figure 8-19. Support for material

1653 The Material and Specularity Project

1654 This project demonstrates the implementation of a version of Phong illumination model
 1655 utilizing the normal map and the position of a camera. It also implements a system
 1656 that stores and forwards per-Renderable object material properties to the GLSL shader
 1657 for the Phong lighting computation. You can see an example of the project running in
 1658 Figure 8-20. The source code of this project is located in the chapter8/8.5.material_
 1659 and_specularity folder.

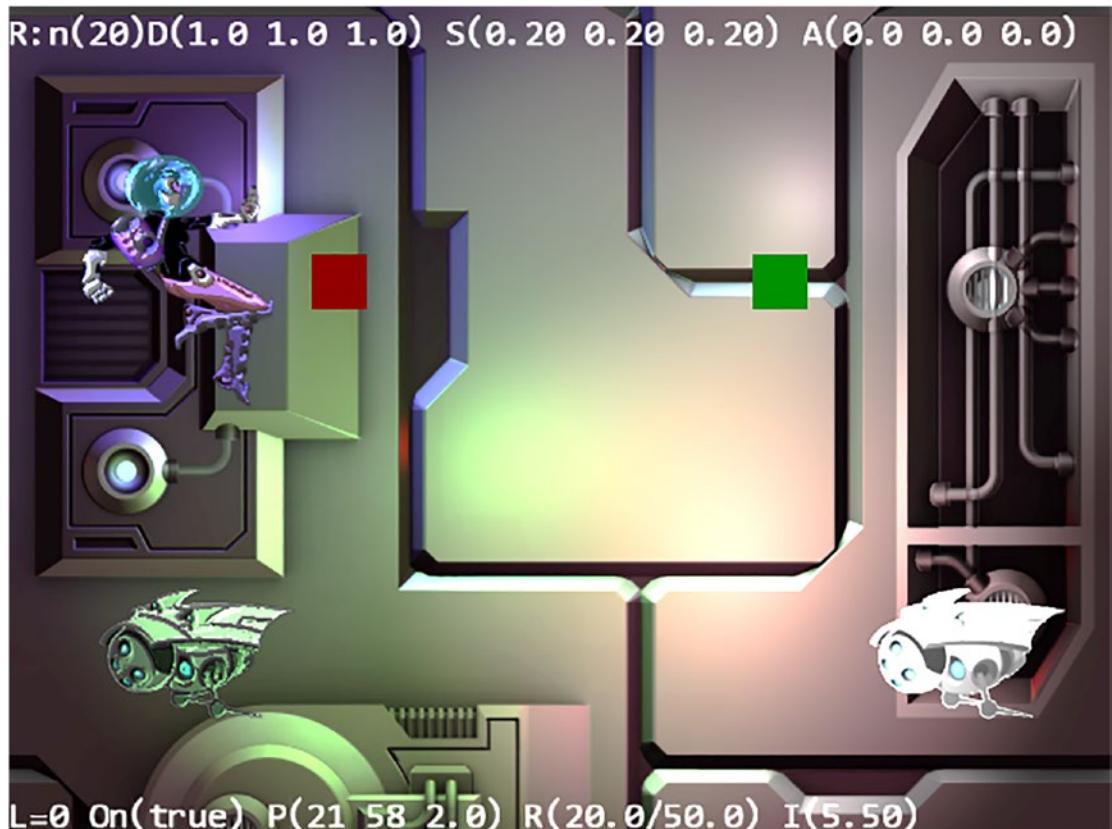


Figure 8-20. Running the Material and Specularity project

The main controls of the project are identical to the previous project:

- **WASD keys:** Move the hero character on the screen

Lighting controls:

- **Number keys 0, 1, 2, and 3:** Select the corresponding light source
- **Arrow keys:** Move the currently selected light
- **Z/X key:** Increases/decreases the light z position
- **C/V and B/N keys:** Increase/decrease the near and far cutoff distances of the selected light
- **K/L key:** Increases/decreases the intensity of the selected light
- **H key:** Toggles the selected light on/off

CHAPTER 8 IMPLEMENTING ILLUMINATION AND SHADOW

1670 The material property controls are new to this project:

- 1671 • **Number keys 5 and 6:** Select the left minion and the hero
- 1672 • **Number keys 7, 8, and 9:** Select the K_a , K_d , and K_s material properties
1673 of the selected character (left minion or the hero)
- 1674 • **E/R, T/Y, and U/I keys:** Increase/decrease the red, green, and blue
1675 channels of the selected material property
- 1676 • **O/P keys:** Increase/decrease the shininess of the selected material
1677 property

1678 The goals of the project are as follows:

- 1679 • To understand specular reflection and the Phong specular term
- 1680 • To implement specular highlight illumination in GLSL fragment
1681 shaders
- 1682 • To understand and experience controlling the Material of an
1683 illuminated object
- 1684 • To examine specular highlights in illuminated images

1685 Modifying the GLSL Illumination Fragment Shader

1686 As in the previous projects, you will begin with implementing the actual illumination
1687 model in the GLSL `illum_fs` fragment shader:

- 1688 1. Edit the `illum_fs.glsl` file and define a variable,
1689 `uCameraPosition`, for storing the camera position. This position
1690 is used to compute the \hat{V} vector, the viewing direction.
1691 Now, create a material struct and a corresponding variable,
1692 `uMaterial`, for storing the per-object material properties. Note the
1693 correspondence between the variable names K_a , K_d , K_s , and n and
1694 the terms in the Phong illumination model in Figure 8-18.

```
1695 // for supporting a simple Phong-like illumination model
1696 uniform vec3 uCameraPosition; // for computing the V-vector
1697 // material properties
1698 struct Material {
```

```

vec4 Ka;      // simple boosting of color          1699
vec4 Kd;      // Diffuse                         1700
vec4 Ks;      // Specular                        1701
float Shininess; // this is the "n"           1702
};

uniform Material uMaterial;                   1703

```

2. To support readability, mathematical terms in the illumination model will be defined into separate functions. You will begin by defining the `DistanceDropOff()` function to perform the exact same near/far cutoff computation as in the previous project. 1705
- 1706
- 1707
- 1708

```

// Computes the L-vector, returns strength          1709
float DistanceDropOff(Light lgt, float dist) {    1710
    float strength = 0.0;                          1711
    if (dist <= lgt.Far) {                         1712
        if (dist <= lgt.Near)                      1713
            strength = 1.0; // no attenuation         1714
        else {                                     1715
            // simple quadratic drop off           1716
            float n = dist - lgt.Near;           1717
            float d = lgt.Far - lgt.Near;         1718
            strength = smoothstep(0.0, 1.0, 1.0-(n*n)/(d*d)); 1719
                                            // blended attenuation 1720
        }
    }                                              1721
    return strength;                                1722
}                                              1723

```

3. Define the function to compute the diffuse term. Notice that the texture map color is applied to the diffuse term. 1725
- 1726

```

vec4 DiffuseResult(vec3 N, vec3 L, vec4 textureMapColor) { 1727
    return uMaterial.Kd * max(0.0, dot(N, L)) * textureMapColor; 1728
}

```

- 1730 4. Define the function to compute the specular term. The \hat{V}
 1731 vector, V , is computed by normalizing the results of subtracting
 1732 uCameraPosition from the current pixel position, gl_FragCoord.
 1733 It is important to observe that this operation is performed in the
 1734 pixel space, and the IllumShader/IllumRenderable object pair
 1735 must transform the WC camera position to pixel space before
 1736 sending over the information.

```
1737 vec4 SpecularResult(vec3 N, vec3 L) {  

1738     vec3 V = normalize(uCameraPosition - gl_FragCoord.xyz);  

1739     vec3 H = (L + V) * 0.5;  

1740     return uMaterial.Ks * pow(max(0.0, dot(N, H)), uMaterial.Shininess);  

1741 }
```

- 1742 5. Now you can implement the Phong illumination model to
 1743 accumulate the diffuse and specular terms. Notice that
 1744 lgt.Intensity, I_L , and lgt.Color, C_L , in Figure 8-18 are factored
 1745 out and multiplied to the sum of diffuse and specular results. The
 1746 scaling by the light strength based on near/far cutoff computation,
 1747 strength, is the only difference between this implementation and
 1748 the diffuse/specular terms listed in Figure 8-18.

```
1749 vec4 ShadedResult(Light lgt, vec3 N, vec4 textureMapColor) {  

1750     vec3 L = lgt.Position.xyz - gl_FragCoord.xyz;  

1751     float dist = length(L);  

1752     L = L / dist;  

1753     float strength = DistanceDropOff(lgt, dist);  

1754     vec4 diffuse = DiffuseResult(N, L, textureMapColor);  

1755     vec4 specular = SpecularResult(N, L);  

1756     vec4 result = strength * lgt.Intensity *  

1757                           lgt.Color * (diffuse + specular);  

1758     return result;  

1759 }
```

6. Complete the implementation in the `main()` function by
 accounting for the ambient term and looping over all defined light
 sources to accumulate for `ShadedResults()`. The bulk of the main
 function is similar to the one in the `illum_fs.gls` file from the
 previous project. The only important differences are highlighted
 in bold.

```

void main(void)  {
    ... identical to previous code ...
    vec3 N = normalize(normalMap.xyz);

vec4 shadedResult = uGlobalAmbientIntensity *
uGlobalAmbientColor * uMaterial.Ka;

// now decide if we should illuminate by the light
if (textureMapColor.a > 0.0) {
    for (int i=0; i<kGLSLuLightArraySize; i++) {
        if (uLights[i].IsOn) {
            shadedResult += ShadedResult(
                uLights[i], N, textureMapColor);
        }
    }
}
... identical to previous code ...
}
```

Defining the Material Class

As described, a simple `Material` class is required to encapsulate the per-Renderable
 material property for the Phong illumination model:

- Create `material.js` in the `src/engine` folder, define the `Material`
 class, and in the constructor, initialize the variables as defined in
 the surface material property in Figure 8-18. Notice that ambient,
 diffuse, and specular (`Ka`, `Kd`, and `Ks`) are colors, while shininess is
 a floating-point number.

CHAPTER 8 IMPLEMENTING ILLUMINATION AND SHADOW

```
1790 class Material {  
1791     constructor() {  
1792         this.mKa = vec4.fromValues(0.0, 0.0, 0.0, 0);  
1793         this.mKs = vec4.fromValues(0.2, 0.2, 0.2, 1);  
1794         this.mKd = vec4.fromValues(1.0, 1.0, 1.0, 1);  
1795         this.mShininess = 20;  
1796     }  
1797     ... implementation to follow ...  
1798 }  
1799 export default Material;
```

1800 2. Provide straightforward get and set accessors to the variables:

```
1801 setAmbient(a) { this.mKa = vec4.clone(a); }  
1802 getAmbient() { return this.mKa; }  
  
1803 setDiffuse(d) { this.mKd = vec4.clone(d); }  
1804 getDiffuse() { return this.mKd; }  
  
1805 setSpecular(s) { this.mKs = vec4.clone(s); }  
1806 getSpecular() { return this.mKs; }  
  
1807 setShininess(s) { this.mShininess = s; }  
1808 getShininess() { return this.mShininess; }
```

1809 Note that the Material class is designed to represent material properties of
1810 Renderable objects and must be accessible to the game programmer. As such, remember
1811 to update the engine access file, index.js, to forward the newly defined functionality to
1812 the client.

1813 Defining the ShaderMaterial Class

1814 Similar to defining the ShaderLightAt class to pass light source information at an
1815 array index to GLSL fragment shader, a new ShaderMaterial class should be defined
1816 to communicate the contents of Material to the GLSL illum_fs shader. Similar to the
1817 implementation of ShaderLightAt, the ShaderMaterial class will also be defined in the
1818 src/engine/shaders folder.

1. Create `shader_material.js` in the `src/engine/shaders` folder, define `ShaderMaterial` class, and in the constructor, initialize the variables as references to the ambient, diffuse, specular, and shininess in the `illum_fs` GLSL shader. 1819
1820
1821
1822

```

import * as glSys from "../core/gl.js"; 1823

class ShaderMaterial { 1824
  constructor(aIllumShader) { 1825
    let gl = glSys.get(); 1826
    this.mKaRef = gl.getUniformLocation( 1827
      aIllumShader, "uMaterial.Ka"); 1828
    this.mKdRef = gl.getUniformLocation( 1829
      aIllumShader, "uMaterial.Kd"); 1830
    this.mKsRef = gl.getUniformLocation( 1831
      aIllumShader, "uMaterial.Ks"); 1832
    this.mShineRef = gl.getUniformLocation( 1833
      aIllumShader, "uMaterial.Shininess"); 1834
  } 1835
  ... implementation to follow ... 1836
} 1837

export default ShaderMaterial; 1838

```

2. Define the `loadToShader()` function to push the content of a Material to the GLSL shader: 1839
1840

```

loadToShader(aMaterial) { 1841
  let gl = glSys.get(); 1842
  gl.uniform4fv(this.mKaRef, aMaterial.getAmbient()); 1843
  gl.uniform4fv(this.mKdRef, aMaterial.getDiffuse()); 1844
  gl.uniform4fv(this.mKsRef, aMaterial.getSpecular()); 1845
  gl.uniform1f(this.mShineRef, aMaterial.getShininess()); 1846
} 1847

```

Similar to the `ShaderLightAt` class, the `ShaderMaterial` class is defined for loading a material to the GLSL fragment shader. This is an internal engine operation. There is no reason for the game programmer to access this class, and thus, the engine access file, `index.js`, should not be modified to forward the definition of this class. 1848
1849
1850
1851

1852 **Modifying the IllumShader Class**

1853 Recall that the `IllumShader` class is the engine's interface to the corresponding GLSL
 1854 `illum_fs` fragment shader. Now the `IllumShader` class must be modified to support
 1855 the newly defined Phong illumination functionality in `illum_fs`. This support can be
 1856 accomplished by modifying the `IllumShader` to define a `ShaderMaterial` object to load
 1857 the contents of a `Material` object to the `illum_fs` fragment shader.

1858 1. Edit `illum_shader.js` in `src/engine/shaders` to import
 1859 `ShaderMaterial`, and modify the constructor to define
 1860 new variables, `mMaterial` and `mCameraPos`, to support
 1861 Phong illumination computation. Then define the variables
 1862 `mMaterialLoader` and `mCameraPosRef` for keeping references and
 1863 for loading the corresponding contents to the uniform variables in
 1864 the shader.

```
1865 import ShaderMaterial from "./shader_material.js";
1866 constructor(vertexShaderPath, fragmentShaderPath) {
1867   // Call super class constructor
1868   super(vertexShaderPath, fragmentShaderPath);
1869   // this is the material property of the Renderable
1870   this.mMaterial = null;
1871   this.mMaterialLoader = new ShaderMaterial(this.mCompiledShader);
1872   let gl = glSys.get();
1873   // Reference to the camera position
1874   this.mCameraPos = null; // points to a vec3
1875   this.mCameraPosRef = gl.getUniformLocation(
1876     this.mCompiledShader, "uCameraPosition");
1877   // reference to the normal map sampler
1878   this.mNormalSamplerRef = gl.getUniformLocation(
1879     this.mCompiledShader, "uNormalSampler");
1880 }
```

2. Modify the activate() function to load the material and camera position to the illum_fs fragment shader: 1881
1882

```
activate(pixelColor, trsMatrix, cameraMatrix) { 1883
    // first call the super class's activate 1884
    super.activate(pixelColor, trsMatrix, cameraMatrix); 1885
    let gl = glSys.get(); 1886
    gl.uniform1i(this.mNormalSamplerRef, 1); // binds to texture unit 1 1887
this.mMaterialLoader.loadToShader(this.mMaterial); 1888
gl.uniform3fv(this.mCameraPosRef, this.mCameraPos); 1889
}
```

1890

3. Define the setMaterialAndCameraPos() function to set the corresponding variables for Phong illumination computation: 1891
1892

```
setMaterialAndCameraPos(m, p) { 1893
    this.mMaterial = m; 1894
    this.mCameraPos = p; 1895
} 1896
```

Modifying the IllumRenderable Class

You can now modify the IllumRenderable class to include a material property and to properly support IllumShader. This is a straightforward change. 1898
1899

1. Edit illum_renderable.js in the src/engine/renderables folder, and modify the constructor to instantiate a new Material object: 1900
1901
1902

```
import Material from "../material.js"; 1903
constructor(myTexture, myNormalMap) { 1904
    ... identical to previous code ...
    // Material for this Renderable 1905
    this.mMaterial = new Material(); 1906
}
```

1907

1908 2. Update the `draw()` function to set the material and camera
 1909 position to the shader before the actual rendering. Notice that
 1910 in the call to `camera.getWCCenterInPixelSpace()`, the camera
 1911 position is properly transformed into pixel space.

```
1912 draw(camera) {  

1913   texture.activate(this.mNormalMap, glSys.get().TEXTURE1);  

1914   this.mShader.setMaterialAndCameraPos(  

1915     this.mMaterial, camera.getWCCenterInPixelSpace());  

1916   super.draw(camera);  

1917 }
```

1918 3. Define a simple accessor for the material object:

```
1919 getMaterial() { return this.mMaterial; }
```

1920 Modifying the Camera Class

1921 As you have seen in the `illum_fs` fragment shader implementation, the camera position
 1922 required for computing the \hat{V} vector must be in pixel space. The `Camera` object must be
 1923 modified to provide such information. Since the `Camera` object stores its position in WC
 1924 space, this position must be transformed to pixel space for each `IllumRenderable` object
 1925 rendered.

1926 There may be a large number of `IllumRenderable` objects in a scene, and the camera
 1927 position cannot be changed once rendering begins. These observations suggest that the
 1928 pixel space camera position should be computed once and cached for each drawing cycle.
 1929 The `PerRenderCache` class, defined in the *Light Source Project* specifically for caching
 1930 per-drawing cycle information, is ideal for caching the pixel space camera position.

1931 1. Edit the `camera_main.js` file and add a `vec3` to the
 1932 `PerRenderCache` to cache the camera's position in pixel space:

```
1933 class PerRenderCache {  

1934   // Information to be updated once per render for efficiency concerns  

1935   constructor() {  

1936     this.mWCToPixelRatio = 1; // WC to pixel transformation  

1937     this.mCameraOrgX = 1; // Lower-left corner of camera in WC  

1938     this.mCameraOrgY = 1;  

1939     this.mCameraPosInPixelSpace = vec3.fromValues(0, 0, 0);
```

```

    }
}

2. In the Camera constructor, define a z variable to simulate the
   distance between the Camera object and the rest of the Renderable
   objects. This third piece of information represents depth and is
   required for the illumination computation.                                1940
                                                               1941
                                                               1942
                                                               1943
                                                               1944
                                                               1945

This.kCameraZ = 10; // this is for illumination computation          1946

3. In step B4 of the setViewAndCameraMatrix() function, call the
   wcPosToPixel() function to transform the camera's position to 3D
   pixel space and cache the computed results:                           1947
                                                               1948
                                                               1949

// Step B4: compute and cache per-rendering information           1950
this.mRenderCache.mWCtoPixelRatio =
    this.mViewport[eViewport.eWidth] / this.getWCwidth();           1951
this.mRenderCache.mCameraOrgX = center[0] - (this.getWCwidth() / 2);  1952
this.mRenderCache.mCameraOrgY = center[1] - (this.getWCheight() / 2); 1953
let p = this.wcPosToPixel(this.getWCCenter());                      1954
this.mRenderCache.mCameraPosInPixelSpace[0] = p[0];                  1955
this.mRenderCache.mCameraPosInPixelSpace[1] = p[1];                  1956
this.mRenderCache.mCameraPosInPixelSpace[2] =
    this.fakeZInPixelSpace(this.kCameraZ);                            1957
                                                               1958
                                                               1959

4. Define the accessor for the camera position in pixel space:      1960

getWCCenterInPixelSpace() {
    return this.mRenderCache.mCameraPosInPixelSpace; }                1961
                                                               1962

```

Testing Specular Reflection

You can now test your implementation of the Phong illumination model and observe the effects of altering an object's material property and specularity. Since the background, Hero, and left Minion are already instances of the IllumRenderable object, these three objects will now exhibit specularity. To ensure prominence of specular reflection, the specular material property, Ks, of the background object is set to bright red in the init() function.

1970 A new function, `_selectCharacter()`, is defined to allow the user to work with the
1971 material property of either the Hero or the left Minion object. The file `my_game_material_`
1972 `control.js` implements the actual user interaction for controlling the selected material
1973 property.

1974 Observations

1975 You can run the project and interactively control the material property of the currently
1976 selected object (type keys 5 to select the left Minion and 6 for the Hero). By default, the
1977 material property of the Hero object is selected. You can try changing the diffuse RGB
1978 components by pressing the E/R, T/Y, or U/I keys. Notice that you can press multiple
1979 keys simultaneously to change multiple color channels at the same time.

1980 The normal map of the background image is carefully generated and thus is best for
1981 examining specularity effects. You can observe red highlights along vertical boundaries
1982 in the background image. If you are unsure, pay attention to the top right region of the
1983 background image, select light 3 (type the 3 key), and toggle the on/off switch (typing the
1984 H key). Notice that as the light toggles from off to on, the entire top right region becomes
1985 brighter with a red highlight along the vertical boundary. This red highlight is the
1986 reflection of light 3 toward the camera. Now, with light 3 switched on, move it toward the
1987 left and right (the left-/right-arrow keys). Observe how the highlight intensifies and then
1988 fades as results of the angle between the halfway vector, \hat{H} , and the face normal vector,
1989 \hat{N} , change.

1990 You can also adjust the material to observe specularity on the Hero. Now, select the
1991 Hero object (type the 6 key), decrease its diffuse material property (press R, Y, and I keys
1992 at the same time) to around 0.2, and increase the specularity property (type 9 to select
1993 Specular, and then press E, T, and U keys at the same time) to values beyond 1. With
1994 this setting, the diffused term is reduced and specular highlight emphasized, you can
1995 observe a dark Hero figure with bright highlight spots. If you are unsure, try toggling light
1996 0 (type the 0 key) on/off (type the H key). At this point, you can press and hold the P key
1997 to decrease the value of the shininess, n . As the n value decreases, you can observe the
1998 increase in the sizes of the highlighted spots coupled with decrease in the brightness
1999 of these spots. As depicted by the middle sphere of Figure 8-14, a smaller n value
2000 corresponds to a less polished surface which typically exhibits highlights that are larger
2001 in area but with less intensity.

Relatively small objects, such as the Hero, do not occupy many pixels; the associated highlight is likely to span even smaller number of pixels and can be challenging to observe. Specular highlights can convey subtle and important effects; however, its usage can also be challenging to master.

2002
2003
2004
2005

Light Source Types

2006
2007
2008
2009
2010
2011

At this point, your game engine supports the illumination by many instances of a single type of light, a point light. A point light behaves much like a lightbulb in the real world. It illuminates from a single position with near and far radii where objects can be fully, partially, or not lit at all by the light. There are two other light types that are popular in most game engines: the directional light and the spotlight.

2012
2013
2014
2015
2016
2017
2018
2019
2020

A directional light, in contrast to the point light, does not have a light position or a range. Rather, it illuminates everything in a specific direction. While these characteristics may not seem intuitive, they are perfect for general background lighting. This is the case in the real world. During the day, the general environment is illuminated by the sun where rays from the sun can conveniently be modeled as a directional light. The light rays from the sun, from the perspective of the earth, are practically parallel coming from a fixed direction, and these rays illuminate everything. A directional light is a simple light type that requires only a direction variable and has no distance drop-off. The directional lights are typically used as global lights that illuminate the entire scene.

2021
2022
2023
2024
2025
2026
2027
2028
2029

A spotlight models a desk lamp with a cone-shape lampshade. As illustrated in Figure 8-21, a spotlight is a point light encompassed by a cone pointing in a specific direction, the light direction, with angular attenuation parameters for the inner and outer cone angles. Similar to the near and far radii of the distance attenuation, objects inside the inner cone angle are fully lit, outside the outer cone angle are not lit, and in between the two angles are partially lit. Just as in the case of a point light, a spotlight is often used for creating illumination effects in specific regions of a game scene. The spotlight, with directional and angular attenuation parameters, offers finer controls for simulating effects that are local to specific areas in a game.

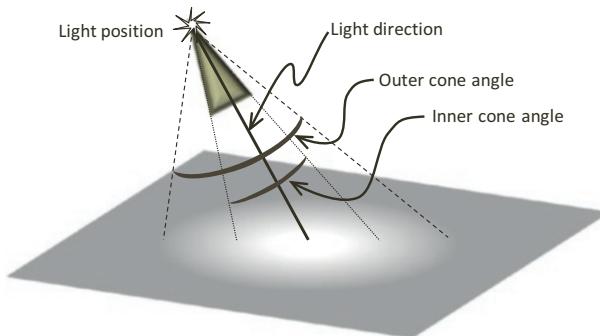


Figure 8-21. A spotlight and its parameters

2030 **Note** In illustrative diagrams, like Figure 8-21, for clarity purposes, light
2031 directions are usually represented by lines extending from the light position toward
2032 the environment. These lines are usually for illustrative purposes and do not carry
2033 mathematical meanings. These illustrative diagrams are contrasted with vector
2034 diagrams that explain illumination computations, like Figures 8-15 and 8-16. In
2035 vector diagrams, all vectors always point away from the position being illuminated
2036 and are assumed to be normalized with a magnitude of 1.

2037 **The Directional and Spotlights Project**

2038 This project demonstrates how to integrate directional lights and spotlights into your
2039 engine to support a wider range of illumination effects. You can see an example of
2040 the project running in Figure 8-22. The source code of this project is located in the
2041 chapter8/8.6.directional_and_spotlights folder.

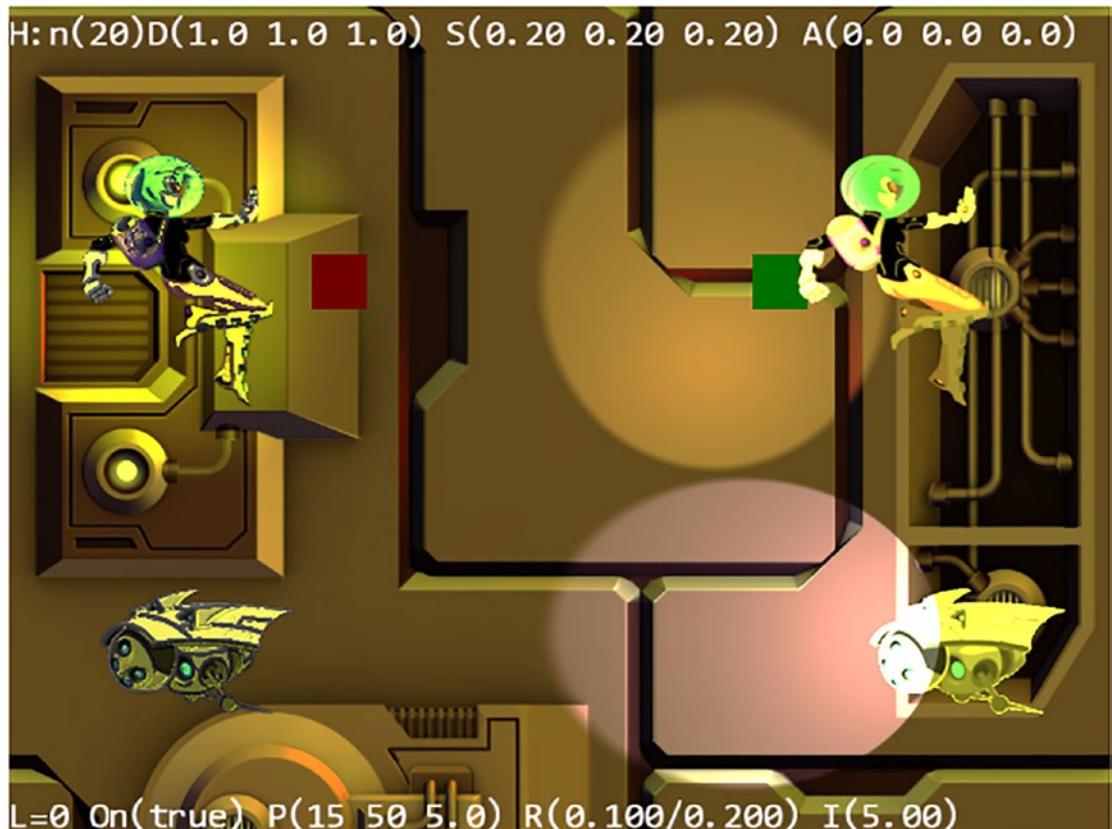


Figure 8-22. Running the Directional and Spotlights project

The controls of the project are as follows:

- **WASD keys:** Move the hero character on the screen

Lighting controls:

- **Number keys 0, 1, 2, and 3:** Select the corresponding light source.
- **Arrow keys:** Move the currently selected light; note that this has no effect on the directional light (light 1).
- **Arrow keys with spacebar pressed:** Change the direction of the currently selected light; note that this has no effect on the point light (light 0).
- **Z/X key:** Increases/decreases the light z position; note that this has no effect on the directional light (light 1).

2042

2043

2044

2045

2046

2047

2048

2049

2050

2051

2052

CHAPTER 8 IMPLEMENTING ILLUMINATION AND SHADOW

- **C/V and B/N keys:** Increase/decrease the inner and outer cone angles of the selected light; note that these only affect the two spotlights in the scene (lights 2 and 3).
- **K/L key:** Increases/decreases the intensity of the selected light.
- **H key:** Toggles the selected light on/off.

Material property controls:

- **Number keys 5 and 6:** Select the left minion and the hero
- **Number keys 7, 8, and 9:** Select the K_a , K_d , and K_s material properties of the selected character (left minion or the hero)
- **E/R, T/Y, and U/I keys:** Increase/decrease the red, green, and blue channels of the selected material property
- **O/P keys:** Increase/decrease the shininess of the selected material property

The goals of the project are as follows:

- To understand the two additional light types: directional lights and spotlights
- To examine the illumination results from all three different light types
- To experience controlling the parameters of all three light types
- To support the three different light types in the engine and GLSL shaders

Supporting New Light Types in GLSL Fragment Shaders

As with the previous projects, the integration of the new functionality will begin with the GLSL shader. You must modify the GLSL `IllumShader` and `LightShader` fragment shaders to support the two new light types.

Modifying the GLSL Illumination Fragment Shader

Recall that the `IllumShader` simulates the Phong illumination model based on a point light. This will be expanded to support the two new light types.

1. Begin by editing `illum_fs.glsl` and defining constants for the three light types. Notice that to support proper communications between the GLSL shader and the engine, these constants must have identical values as the corresponding enumerated data defined in the `light.js` file.

```
#define ePointLight      0
#define eDirectionalLight 1
#define eSpotLight        2
// ***** WARNING *****
// The above enumerated values must be identical to
// Light.eLightType values defined in Light.js
// ***** WARNING *****
```

2. Expand the `Light` struct to accommodate the new light types. While the directional light requires only a `Direction` variable, a spotlight requires a `Direction`, inner and outer angles, and a `DropOff` variable. As will be detailed next, instead of the actual angle values, the cosines of the inner and outer angles are stored in the struct to facilitate efficient implementation. The `DropOff` variable controls how rapidly light drops off between the inner and outer angles of the spotlight. The `LightType` variable identifies the type of light that is being represented in the struct.

```
struct Light {
    vec3 Position; // in pixel space!
vec3 Direction; // Light direction
    vec4 Color;
    float Near;
    float Far;
    float CosInner; // Cosine of inner cone angle for spotlight
    float CosOuter; // Cosine of outer cone angle for spotlight
    float Intensity;
    float DropOff; // for spotlight
    bool IsOn;
    int LightType; // One of ePoint, eDirectional, or eSpot
};
```

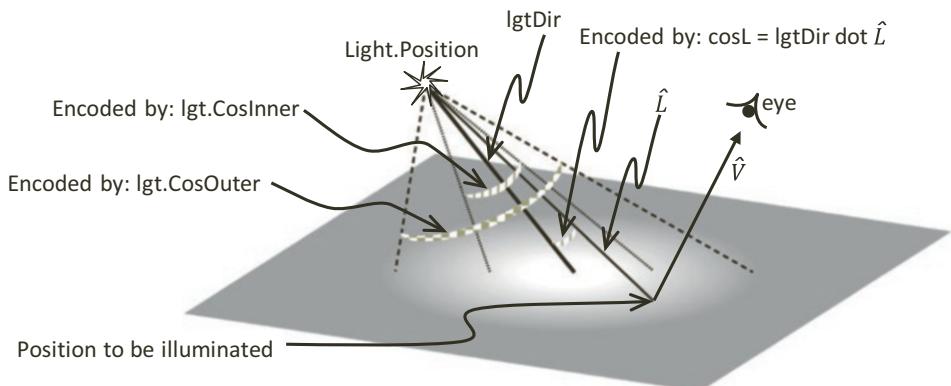
CHAPTER 8 IMPLEMENTING ILLUMINATION AND SHADOW

- 2114 3. Define an `AngularDropOff()` function to compute the angular
2115 attenuation for the spotlight:

```
2116 float AngularDropOff(Light lgt, vec3 lgtDir, vec3 L) {  
2117     float strength = 0.0;  
2118     float cosL = dot(lgtDir, L);  
2119     float num = cosL - lgt.CosOuter;  
2120     if (num > 0.0) {  
2121         if (cosL > lgt.CosInner)  
2122             strength = 1.0;  
2123         else {  
2124             float denom = lgt.CosInner - lgt.CosOuter;  
2125             strength = smoothstep(0.0, 1.0, pow(num/denom, lgt.DropOff));  
2126         }  
2127     }  
2128     return strength;  
2129 }
```

2130 The parameter `lgt` is a spotlight in the `Light` struct, `lgtDir` is the direction of
2131 the spotlight (or `Light.Direction` normalized), and `L` is the light vector of the current
2132 position to be illuminated. Note that since the dot product of normalized vectors is
2133 the cosine of the angle between the vectors, it is convenient to represent all angular
2134 displacements by their corresponding cosine values and to perform the computations
2135 based on cosines of the angular displacements. Figure 8-23 illustrates the parameters
2136 involved in angular attenuation computation.

2137 **Note** The `lgtDir` is the direction of the spotlight, while the light vector, `L`, is the
2138 vector from the position being illuminated to the position of the spotlight.



this figure will be printed in b/w

Figure 8-23. Computing the angular attenuation of a spotlight

Note The following code is based on cosine of angular displacements. It is important to remember that given two angles α and β , where both are between 0 and 180 degrees, if $\alpha > \beta$, then, $\cos \alpha < \cos \beta$.

- a. The $\cos L$ is the dot product of L with $lgtDir$; it records the angular displacement of the position currently being illuminated. 2142
2143
- b. The num variable stores the difference between $\cos L$ and \cosOuter . 2144
A negative num would mean that the position currently being 2145
illuminated is outside the outer cone where the position will not 2146
be lit and thus no further computation is required. 2147
- c. If the point to be illuminated is within the inner cone, $\cos L$ would 2148
be greater than $lgt.CosInner$, and full strength of the light, 1.0, 2149
will be returned. 2150
- d. If the point to be illuminated is in between the inner and outer 2151
cone angles, use the `smoothstep()` function to compute the 2152
effective strength from the light. 2153

```

2154     4. Modify the ShadedResults() function to handle each separate
2155         case of light source type before combining the results into a color:
2156
2157     vec4 ShadedResult(Light lgt, vec3 N, vec4 textureMapColor) {
2158         float aStrength = 1.0, dStrength = 1.0;
2159         vec3 lgtDir = -normalize(lgt.Direction.xyz);
2160         vec3 L; // light vector
2161         float dist; // distance to light
2162         if (lgt.LightType == eDirectionalLight) {
2163             L = lgtDir;
2164         } else {
2165             L = lgt.Position.xyz - gl_FragCoord.xyz;
2166             dist = length(L);
2167             L = L / dist;
2168         }
2169         if (lgt.LightType == eSpotLight) {
2170             // spotlight: do angle dropoff
2171             aStrength = AngularDropOff(lgt, lgtDir, L);
2172         }
2173         if (lgt.LightType != eDirectionalLight) {
2174             // both spot and point light has distance dropoff
2175             dStrength = DistanceDropOff(lgt, dist);
2176         }
2177         vec4 diffuse = DiffuseResult(N, L, textureMapColor);
2178         vec4 specular = SpecularResult(N, L);
2179         vec4 result = aStrength * dStrength *
2180                         lgt.Intensity * lgt.Color * (diffuse + specular);
2181         return result;
2182     }

```

2182 **Modifying the GLSL Light Fragment Shader**

2183 You can now modify the GLSL `light_fs` fragment shader to support the two new light
 2184 types. The modifications involved are remarkably similar to the changes made for `illum_`
 2185 `fs`, where constant values that correspond to light types are defined, the `Light` struct is

extended to support directional and spotlights, and the angular and distant attenuation functions are defined to properly compute the strengths from the light. Please refer to the `light_fs.glsl` source code file for details of the implementation.

Modifying the Light Class

You must extend the Light class to support the parameters of the two new light types: 2190

1. Edit `light.js` in the `src/engine/lights` folder to define and export an enumerated data type for the different light types. It is important that the enumerated values correspond to the constant values defined in the GLSL `illum_fs` and `light_fs` shaders. 2191
2192
2193
2194

```
// **** WARNING: The following enumerate values must be identical to      2195
// the values of          2196
//
//    ePointLight, eDirectionalLight, eSpotLight      2197
//
// defined in LightFS.glsl and IllumFS.glsl      2198
const eLightType = Object.freeze({
    ePointLight: 0,          2199
    eDirectionalLight: 1,    2200
    eSpotLight: 2           2201
});
export { eLightType }      2202
                                2203
                                2204
                                2205
                                2206
```

2. Modify the constructor to define and initialize the new variables
that correspond to the parameters of directional light and
spotlight.

```
constructor() {  
    this.mColor = vec4.fromValues(1, 1, 1, 1); // light color  
    this.mPosition = vec3.fromValues(0, 0, 5); // light position in WC  
this.mDirection = vec3.fromValues(0, 0, -1); // in WC  
    this.mNear = 5; // effective radius in WC  
    this.mFar = 10;  
this.mInner = 0.1; // in radian  
}
```

```

2217   this.mOuter = 0.3;
2218   this.mIntensity = 1;
2219   this.mDropOff = 1; // 
2220   this.mLightType = eLightType.ePointLight;
2221   this.mIsOn = true;
2222 }
```

- 2223 3. Define the get and set accessors for the new variables. The
 2224 exhaustive listing of these functions is not shown here. Please
 2225 refer to the light.js source code file for details.

2226 Modifying the ShaderLightAt Class

2227 Recall that the ShaderLightAt class is responsible for loading the values from a light
 2228 source to the GLSL fragment shader. This object must be refined to support the new light
 2229 source parameters that correspond to directional lights and spotlights.

- 2230 1. Edit shader_light_at.js to import the eLightType enumerated
 2231 type from light.js:

2232 `import { eLightType } from "../lights/light.js";`

- 2233 2. Modify the _setShaderReferences() function to set the
 2234 references to the newly added light properties:

```

2235 _setShaderReferences(aLightShader, index) {
2236   let gl = glSys.get();
2237   this.mColorRef = gl.getUniformLocation(
2238     aLightShader, "uLights[" + index + "].Color");
2239   this.mPosRef = gl.getUniformLocation(
2240     aLightShader, "uLights[" + index + "].Position");
2241   this.mDirRef = gl.getUniformLocation(
2242     aLightShader, "uLights[" + index + "].Direction");
2243   this.mNearRef = gl.getUniformLocation(
2244     aLightShader, "uLights[" + index + "].Near");
2245   this.mFarRef = gl.getUniformLocation(
2246     aLightShader, "uLights[" + index + "].Far");
2247   this.mInnerRef = gl.getUniformLocation(
2248     aLightShader, "uLights[" + index + "].CosInner");
```

```

this.mOuterRef = gl.getUniformLocation(
    aLightShader, "uLights[" + index + "].CosOuter"); 2249
this.mIntensityRef = gl.getUniformLocation( 2250
    aLightShader, "uLights[" + index + "].Intensity"); 2251
this.mDropOffRef = gl.getUniformLocation( 2252
    aLightShader, "uLights[" + index + "].DropOff"); 2253
this.mIsOnRef = gl.getUniformLocation( 2254
    aLightShader, "uLights[" + index + "].IsOn"); 2255
this.mLightTypeRef = gl.getUniformLocation( 2256
    aLightShader, "uLights[" + index + "].LightType"); 2257
} 2258
2259

```

3. Modify the `loadToShader()` function to load the newly added light variables for the directional light and spotlight. Notice that depending upon the light type, the values of some variables may not be transferred to the GLSL shader. For example, the parameters associated with angular attenuation, the inner and outer angles, and the drop-off will be transferred only for spotlights.

```

loadToShader(aCamera, aLight) { 2267
    let gl = glSys.get(); 2268
    gl.uniform1i(this.mIsOnRef, aLight.isLightOn()); 2269
    // Process a light only when it is switched on 2270
    if (aLight.isLightOn()) { 2271
        ... identical to previous code ... 2272
        gl.uniform1f(this.mFarRef, f); 2273
        gl.uniform1f(this.mInnerRef, 0.0); 2274
        gl.uniform1f(this.mOuterRef, 0.0); 2275
        gl.uniform1f(this.mIntensityRef, aLight.getIntensity()); 2276
        gl.uniform1f(this.mDropOffRef, 0); 2277
        gl.uniform1i(this.mLightTypeRef, aLight.getLightType()); 2278
        // Point light does not need the direction 2279
        if (aLight.getLightType() === eLightType.ePointLight) { 2280
            gl.uniform3fv(this.mDirRef, vec3.fromValues(0, 0, 0)); 2281
        }
    }
}

```

```

2282     } else {
2283         // either spot or directional lights: must compute direction
2284         let d = aCamera.wcDirToPixel(aLight.getDirection());
2285         gl.uniform3fv(this.mDirRef, vec3.fromValues(d[0],d[1],d[2]));
2286         if (aLight.getLightType() === eLightType.eSpotLight) {
2287             gl.uniform1f(this.mInnerRef,
2288                         Math.cos(0.5 * aLight.getInner()));
2289             gl.uniform1f(this.mOuterRef,
2290                         Math.cos(0.5 * aLight.getOuter()));
2291             gl.uniform1f(this.mDropOffRef, aLight.getDropOff());
2292         }
2293     }
2294 }
2295 }
```

2296 Note, for `mInnerRef` and `mOuterRef`, the cosines of half the angles are actually
 2297 computed and passed. Inner and outer angles are the total angular spreads of the
 2298 spotlight where the half of these angles describe the angular displacements from the
 2299 light direction. For this reason, cosines of the half angles will actually be used in the
 2300 computations. This optimization relieves the GLSL fragment shaders from recomputing
 2301 the cosine of these angles on every invocation.

2302 Modifying the Camera Transform Class

2303 Directional lights and spotlights require a light direction, and the GLSL `illum_fs` and
 2304 `light_fs` shaders expect this direction to be specified in pixel space. Edit the `camera_`
 2305 `xform.js` file of the `Camera` object to define the `wcDirToPixel()` function to transform a
 2306 direction from WC to pixel space.

```

2307 Camera.prototype.wcDirToPixel = function (d) { // d:vec3 direction in WC
2308   // Convert the position to pixel space
2309   let x = d[0] * this.mRenderCache.mWCToPixelRatio;
2310   let y = d[1] * this.mRenderCache.mWCToPixelRatio;
2311   let z = d[2];
2312   return vec3.fromValues(x, y, z);
2313 }
```

Testing the New Light Types

The main goals of the MyGame level are to test and provide functionality for manipulating the new light types. The modifications involved are straightforward; `my_game_lights.js` is modified to create all three light types, and `my_game_light_control.js` is modified to support the manipulation of the direction of the selected light when the arrow and space keys are pressed simultaneously. The implementation of these simple changes is not shown here. Please refer to the source code files for details.

Observations

You can run the project and interactively control the lights to examine the corresponding effects. There are four light sources defined, each illuminating all objects in the scene. Light source 0 is a point light, 1 is a directional light, and 2 and 3 are spotlights.

You can examine the effect from a directional light by typing the 1 key to select the light. Now hold the spacebar while taking turns pressing the left/right or up/down keys to swing the direction of the directional light. You will notice drastic illumination changes on the boundary edges of the 3D geometric shapes in the background image, together with occasional prominent red spots of specular reflections. Now, type the H key to switch off the directional light and observe the entire scene becomes darker. Without any kinds of attenuation, directional lights can be used as effective tools for brightening the entire scene.

Type the 2 or 3 keys to select one of the spotlights, once again, by holding the spacebar while taking turns pressing the left/right or up/down keys to swing the direction of the spotlight. With the spotlight, you will observe the illuminated region swinging and changing shapes between a circle (when the spotlight is pointing perpendicularly toward the background image) and different elongated ellipses. The arrow keys will move the illuminated region around. Try experimenting with the C/V and B/N keys to increase/decrease the inner and outer cone angles. Notice that if you set the inner cone angle to be larger than the outer one, the boundary of the illuminated region becomes sharp where lighting effects from the spotlight will drop off abruptly. You can consider switching off the direction light, light 1, for a clearer observation of the spotlight effects.

Try experimenting with the different light settings, including overlapping the light illumination regions and setting the light intensities, the K and L keys, to negative numbers. While impossible in the physical world, negative intensity lights are completely valid options in a game world.

Shadow Simulation

2347

2348 Shadow is the result of light being obstructed or occluded. As an everyday phenomenon,
2349 shadow is something you observe but probably do not give much thought to. However,
2350 shadow plays a vital role in the visual perception system of humans. For example, the
2351 shadows of objects convey important cues of relative sizes, depths, distances, orderings,
2352 and so on. In video games, proper simulation of shadows can increase the quality of
2353 appearance and fidelity. For example, you can use shadows to properly convey the
2354 distance between two game objects or the height that the hero is jumping.

2355 Shadows can be simulated by determining the visibility between the position to be
2356 illuminated and each of the light source positions in the environment. A position is in
2357 shadow with respect to a light source if something occludes it from the light source or
2358 the position is not visible from the light source. Computationally, this is an expensive
2359 operation because general visibility determination is an $O(n)$ operation, where n is
2360 the number of objects in the scene, and this operation must be performed for every
2361 pixel being illuminated. Algorithmically, this is a challenging problem because the
2362 solutions for visibility must be available within the fragment shader during illumination
2363 computation, again, for every pixel being illuminated.

2364 Because of the computation and algorithmic challenges, instead of simulating
2365 shadow according to the physical world, many videogames approximate or create
2366 shadow-like effects for only selected objects based on dedicated hardware resources. In
2367 this section, you will learn about approximating shadows by selecting dedicated shadow
2368 casters and receivers based on the WebGL stencil buffer.

2369 Figure 8-24 shows an example where a game wants to cast the shadow of the Hero
2370 object on the minion and yet not on the background. In this case, the background object
2371 will not participate in the shadow computation and thus will not receive the shadow.

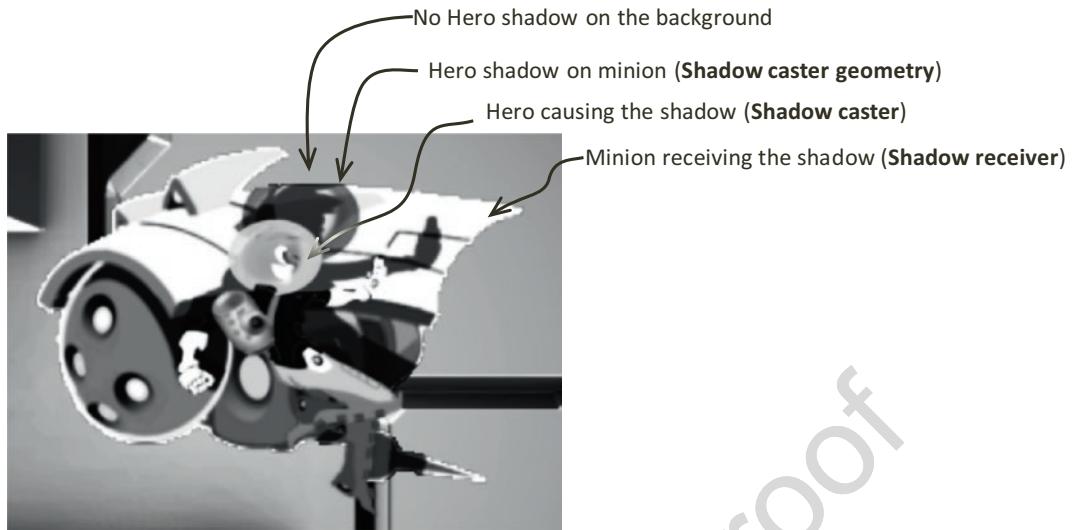


Figure 8-24. Hero casting shadow on the minion but not on the background

To properly simulate and render the shadow in Figure 8-24, as illustrated in Figure 8-25, there are three important elements.

- **Shadow caster:** This is the object that causes the shadow. In the Figure 8-24 example, the Hero object is the shadow caster.
- **Shadow receiver:** This is the object that the shadow appears on. In the Figure 8-24 example, the Minion object is the shadow receiver.
- **Shadow caster geometry:** This is the actual shadow, in other words, the darkness on the shadow receiver because of the occlusion of light. In the Figure 8-24 example, the dark imprint of the hero appearing on the minion behind the actual hero object is the shadow caster geometry.

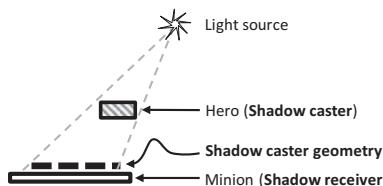


Figure 8-25. The three participating elements of shadow simulation: the caster, the caster geometry, and the receiver

Given the three participating elements, the shadow simulation algorithm is rather straightforward: compute the shadow caster geometry, render the shadow receiver as usual, render the shadow caster geometry as a dark shadow caster object over the receiver, and, finally, render the shadow caster as usual. For example, to render the shadow in Figure 8-24, the dark hero shadow caster geometry is first computed based on the positions of the light source, the Hero object (shadow caster), and the Minion object (shadow receiver). After that, the Minion object (shadow receiver) is first rendered as usual, followed by rendering the shadow caster geometry as the Hero object with a dark constant color, and lastly the Hero object (shadow caster) is rendered as usual.

Take note that shadow is actually a visual effect where colors on objects appear darker because light energy is obstructed. The important point to note is that when a human observes shadows, there are no new objects or geometries involved. This is in stark contrast to the described algorithm, where shadows are simulated by the shadow caster geometry, a dark color object. This dark color object does not actually exist in the scene. It is algorithmically created to approximate the visual perception of light being occluded. This creation and rendering of extra geometry to simulate the results of human visual perception, while interesting, has its own challenges.

As depicted in Figure 8-26, the illusion of shadow breaks down when the shadow caster geometry extends beyond the bounds of the shadow receiver. Such situations must be properly resolved in order for shadow to appear genuine. An example of proper handling of this situation can be observed in Figure 8-24; the top portion of the hero helmet shadow extends beyond the bounds of the minion and is not drawn.

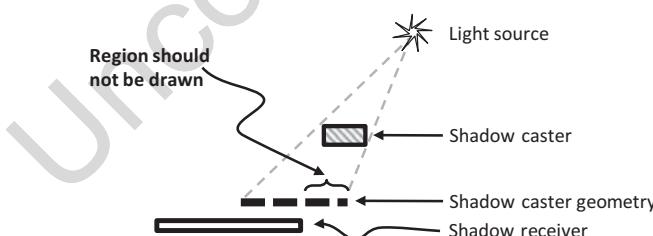


Figure 8-26. Shadow caster extends beyond the bounds of shadow receiver

Fortunately, the WebGL stencil buffer is designed specifically to resolve these types of situations. The WebGL stencil buffer can be configured as a 2D array of on/off switches with the same pixel resolution as the canvas that is displayed on the web

browser. With this configuration, when stencil buffer checking is enabled, the pixels in the canvas that can be drawn on will be only those with corresponding stencil buffer pixels that are switched on.

Figure 8-27 uses an example to illustrate this functionality. In this example, the middle layer is the stencil buffer with all pixels initialized to off except for the pixels in the white triangular region being initialized to on. When the stencil buffer checking is enabled, the drawing of the top layer image will result in only a triangular region appearing in the canvas (bottom layer). This triangular region is formed by pixels that correspond to the on positions of the triangle in the stencil buffer. In this way, the stencil buffer acts exactly like a stencil over the canvas where only the on regions can be drawn on.

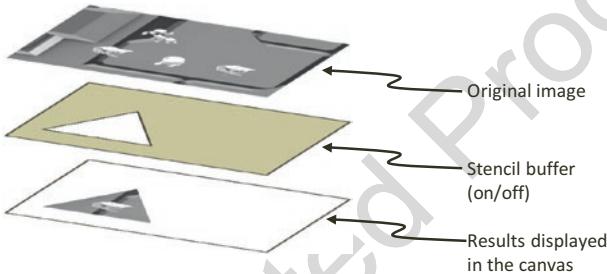


Figure 8-27. The WebGL stencil buffer

With the support of the WebGL stencil buffer, shadow simulation can now be specified accordingly by identifying all shadow receivers and by grouping corresponding shadow casters with each receiver. In the Figure 8-24 example, the Hero object is grouped as the shadow caster of the minion shadow receiver. In this example, for the background object to receive a shadow from the hero, it must be explicitly identified as a shadow receiver, and the Hero object must be grouped with it as a shadow caster. Notice that without explicitly grouping the minion object as a shadow caster of the background shadow receiver, the minion will not cast a shadow on the background.

As will be detailed in the following implementation discussion, the transparencies of the shadow casters and receivers and the intensity of the casting light source can all affect the generation of shadows. It is important to recognize that this shadow simulation is actually an algorithmic creation with effects that can be used to approximate human perception. This procedure does not describe how shadows are formed in the real world, and it is entirely possible to create unrealistic dramatic effects such as casting transparent or blue-colored shadows.

2433 The Shadow Simulation Algorithm

2434 The shadow simulation and rendering algorithm can now be outlined as follows:

2435 Given a shadowReceiver

2436 A: Draw the shadowReceiver to the canvas as usual

2437 // Stencil op to enable the region for drawing on the shadowCaster

2438 B1: Initialize all stencil buffer pixels to off

2439 B2: Switch on stencil buffer pixels correspond to shadowReceiver

2440 B3: Enable stencil buffer checking

2441 // Compute shadowCaster geometries and draw them on shadowReceiver

2442 C: For each shadowCaster of this shadowReceiver

2443 D: For each shadow casting light source

2444 D1: Compute the shadowCaster geometry

2445 D2: Draw the shadowCaster geometry

2446 The listed code renders the shadow receiver and all the shadow caster geometries

2447 without rendering the actual shadow caster objects. The B1, B2, and B3 steps switch

2448 on the stencil buffer pixels that correspond to the shadow receiver. This is similar

2449 to switching on the pixels that are associated with the white triangle in Figure 8-27,

2450 enabling the region that can be drawn. The loops of steps C and D point out that a

2451 separate geometry must be computed for each shadow casting light source. By the time

2452 step D1 draws the shadow caster geometry, with the stencil buffer containing the shadow

2453 receiver imprint and checking enabled, only pixels occupied by the shadow receiver will

2454 be enabled to be drawn on in the canvas.

2455 The Shadow Shaders Project

2456 This project demonstrates how to implement and integrate the shadow simulation

2457 algorithm into your game engine. You can see an example of the project running in

2458 Figure 8-28. The source code of this project is located in the chapter8/8.7.shadow_

2459 shaders folder.



Figure 8-28. Running the Shadow Shaders project

The controls of this project are identical to the previous project:

- **WASD keys:** Move the hero character on the screen

Lighting controls:

- **Number keys 0, 1, 2, and 3:** Select the corresponding light source
- **Arrow keys:** Move the currently selected light; note that this has no effect on the directional light (light 1).
- **Arrow keys with spacebar pressed:** Change the direction of the currently selected light; note that this has no effect on the point light (light 0).
- **Z/X key:** Increases/decreases the light z position; note that this has no effect on the directional light (light 1).

CHAPTER 8 IMPLEMENTING ILLUMINATION AND SHADOW

- **C/V and B/N keys:** Increase/decrease the inner and outer cone angles of the selected light; note that these only affect the two spotlights in the scene (lights 2 and 3).
- **K/L key:** Increases/decreases the intensity of the selected light.
- **H key:** Toggles the selected light on/off.

Material property controls:

- **Number keys 5 and 6:** Select the left minion and the hero
- **Number keys 7, 8, and 9:** Select the K_a , K_d , and K_s material properties of the selected character (left minion or the hero)
- **E/R, T/Y, and U/I keys:** Increase/decrease the red, green, and blue channels of the selected material property
- **O/P keys:** Increase/decrease the shininess of the selected material property

The goals of the project are as follows:

- Understand shadows can be approximated by algorithmically defining and rendering explicit geometries
- Appreciate the basic operations of the WebGL stencil buffer
- Understand the simulation of shadows with shadow caster and receiver
- Implement the shadow simulation algorithm based on the WebGL stencil buffer

Creating GLSL Fragment Shaders

Two separate GLSL fragment shaders are required to support the rendering of shadow, one for drawing the shadow caster geometry onto the canvas and one for drawing the shadow receiver into the stencil buffer.

Defining the GLSL Shadow Caster Fragment Shader

2496

The GLSL shadow_caster_fs fragment shader supports the drawing of the shadow caster geometries. Refer to Figure 8-25; the shadow caster geometry is the piece of geometry that fakes being the shadow of the shadow caster. This geometry is typically scaled by the engine according to its distance from the shadow caster; the further from the caster, the larger this geometry.

2497

2498

2499

2500

2501

In the fragment shader, this geometry should be rendered as a dark-colored object to create the illusion of it being a shadow. Note that one shadow caster geometry is required for each shadow casting light source; as such, the fragment shader only supports one light source. Finally, the degree of darkness of this object depends on the effective strength of the shadow casting light sources, and thus, the fragment shader must define functionality to compute strengths from each type of light sources.

2502

2503

2504

2505

2506

2507

1. In the src/glsl_shaders folder, create a file shadow_caster_fs.glsl. Since all light types can cast shadow, existing light structures must be supported. Now, copy the Light struct and light type constants from light_fs (not shown). These data structure and constants must be exactly the same such that the corresponding interfacing shader in the engine can reuse existing utilities that support LightShader. The only difference is since a shadow caster geometry must be defined for each light source, the uLight array size is exactly 1 in this case.
2. Define constants for shadow rendering. The kMaxShadowOpacity is how opaque shadows should be, and kLightStrengthCutOff is a cutoff threshold where a light with intensity less than this value will not cast shadows.

2508

2509

2510

2511

2512

2513

2514

2515

2516

2517

2518

2519

2520

```
#define kMaxShadowOpacity 0.7 // max of shadow opacity
#define kLightStrengthCutOff 0.05 // any less will not cause shadow
```

2521

2522

3. To properly support shadow casting from the three different light types, AngularDropOff() and DistanceDropOff(), functions must also be defined in exactly the same manner as those in light_fs (and illum_fs). You can copy these functions from light_fs. Note that since there is only one light source in the uLight array, you can remove the light parameter from these functions and

2523

2524

2525

2526

2527

2528

refer directly to `uLight[0]` in the computation. This parameter replacement is the only modification required, and thus, the code is not shown here.

4. Remember that shadow is observed because of light occlusion and is independent from the color of the light source. Now, modify the `LightStrength()` function to compute the light strength arriving at the position to be illuminated instead of a shaded color.

```

2529     refer directly to uLight[0] in the computation. This parameter
2530     replacement is the only modification required, and thus, the code
2531     is not shown here.

2532     4. Remember that shadow is observed because of light occlusion
2533         and is independent from the color of the light source. Now, modify
2534         the LightStrength() function to compute the light strength
2535         arriving at the position to be illuminated instead of a shaded color.

2536 float LightStrength() {
2537     float aStrength = 1.0, dStrength = 1.0;
2538     vec3 lgtDir = -normalize(uLights[0].Direction.xyz);
2539     vec3 L; // light vector
2540     float dist; // distance to light
2541     if (uLights[0].LightType == eDirectionalLight) {
2542         L = lgtDir;
2543     } else {
2544         L = uLights[0].Position.xyz - gl_FragCoord.xyz;
2545         dist = length(L);
2546         L = L / dist;
2547     }
2548     if (uLights[0].LightType == eSpotLight) {
2549         // spotlight: do angle dropoff
2550         aStrength = AngularDropOff(lgtDir, L);
2551     }
2552     if (uLights[0].LightType != eDirectionalLight) {
2553         // both spot and point light has distance dropoff
2554         dStrength = DistanceDropOff(dist);
2555     }
2556     float result = aStrength * dStrength;
2557     return result;
2558 }
```

Comparing the listed `LightStrength()` with the same function in `light_fs`, there are two main differences. First, the function does not consider the color of the light and returns a float, the aggregated strength of the light source. Second, since the `uLight` array is of size 1, the function removed the `Light` parameter and referred to `uLight[0]` in the computation.

5. Compute the color of the shadow in the `main()` function based
on the strength of the light source. Notice that no shadows will be
cast if the light intensity is less than `kLightStrengthCutOff` and
that the actual color of the shadow is not exactly black or opaque.
Instead, it is a blend of the programmer-defined `uPixelColor` and
the sampled transparency from the texture map.

```
void main(void) 2569
{
    vec4 texFragColor = texture2D(uSampler, vTexCoord); 2570
    float lgtStrength = LightStrength(); 2571
    if (lgtStrength < kLightStrengthCutOff) 2572
        discard; 2573
    vec3 shadowColor = lgtStrength * uPixelColor.rgb; 2574
    shadowColor *= uPixelColor.a * texFragColor.a; 2575
    gl_FragColor = vec4(shadowColor, 2576
                        kMaxShadowOpacity * lgtStrength * texFragColor.a); 2577
} 2578
```

Defining the GLSL Shadow Receiver Fragment Shader

The GLSL `shadow_receiver_fs` fragment shader is the shader for drawing the shadow receiver into the stencil buffer. Take note that the stencil buffer is configured as an on/off buffer where any value returned in `gl_FragColor` will switch the corresponding pixel to on. For this reason, transparent receiver fragments must be discarded.

- Under the `src/glsl_shaders` folder, create `shadow_receiver_fs.glsl`, and define a `sampler2D` object to sample the color texture map of the shadow receiver object. In addition, define the constant `kSufficientlyOpaque` to be the threshold where fragments with less opacity will be treated as transparent and discarded. Stencil buffer pixels that correspond to discarded fragments will remain off and thus will not be able to receive shadow geometries.

```
// The object that fetches data from texture. 2593
// Must be set outside the shader. 2594
uniform sampler2D uSampler; 2595
```

```

2596 uniform vec4 uPixelColor;
2597 // "varying" signifies that the texture coordinate will be
2598 // interpolated and thus varies.
2599 varying vec2 vTexCoord;
2600 #define kSufficientlyOpaque      0.1
2601
2602     Note that to facilitate engine shader class code reuse, the variable names of uSampler
2603 and vTexCoord must not be changed. These correspond to the variable names defined in
2604 texture_fs.glsl, and the game engine can use the existing SpriteShader to facilitate
2605 the loading of information to this shader.

```

- 2605 2. Implement the `main()` function to sample the texture of shadow

2606 receiver object and test for opacity threshold in determining if

2607 shadow could be received:

```

2608 void main(void)
2609 {
2610     vec4 texFragColor = texture2D(uSampler, vTexCoord);
2611     if (texFragColor.a < kSufficientlyOpaque)
2612         discard;
2613     else
2614         gl_FragColor = vec4(1, 1, 1, 1);
2615 }

```

2616 **Interfacing the GLSL Shadow Shaders to the Engine**

2617 With two new GLSL shaders defined, you may expect that it is necessary to define two

2618 corresponding SimpleShader/Renderable pairs to facilitate the communications. This is

2619 not the case for two reasons:

- 2620 • First, only one new engine shader type is required for supporting

2621 shadow_caster_fs. With the strategic variable naming in the shadow_

2622 receiver_fs shader, the existing SpriteShader object can be used to

2623 communicate with the shadow_receiver_fs GLSL fragment shader.

- Second, no new Renderable classes are required. The Renderable classes are designed to support the drawing and manipulation of game objects with the corresponding shaders. In this way, Renderable objects are visible to the players. In the case of shadow shaders, `shadow_caster_fs` draws shadow caster geometries, and `shadow_receiver_fs` draws the shadow receiver geometry into the stencil buffer. Notice that neither of the shaders is designed to support drawing of objects that are visible to the players. For these reasons, there is no need for the corresponding Renderable objects.

Creating the Shadow Caster Shader

A JavaScript `SimpleShader` subclass must be defined to facilitate the loading of information from the game engine to the GLSL shader. In this case, a `ShadowCasterShader` needs to be defined to communicate with the GLSL `shadow_caster_fs` fragment shader.

1. Under the `src/engine/shaders` folder, create `shadow_caster_shader.js`; define the `ShadowCasterShader` class to inherit from `SpriteShader`. Since each shadow caster geometry is created by one casting light source, define a single light source for the shader.

```
import SpriteShader from "./sprite_shader.js";
import ShaderLightAt from "./shader_light_at.js";

class ShadowCasterShader extends SpriteShader {
    // constructor
    constructor(vertexShaderPath, fragmentShaderPath) {
        super(vertexShaderPath, fragmentShaderPath);

        this.mLight = null; // The light that casts the shadow
        this.mCamera = null;

        // GLSL Shader must define uLights[1] (size of 1)!!
        this.mShaderLight = new ShaderLightAt(this.mCompiledShader, 0);
    }
    ... implementation to follow ...
}
export default ShadowCasterShader;
```

2656 2. Override the activate() function to ensure the single light source
 2657 is properly loaded to the shader:

```
2658 // Overriding the activation of the shader for rendering
2659 activate(pixelColor, trsMatrix, cameraMatrix) {
2660     // first call the super class's activate
2661     super.activate(pixelColor, trsMatrix, cameraMatrix);
2662     this.mShaderLight.loadToShader(this.mCamera, this.mLight);
2663 }
```

2664 3. Define a function to set the current camera and light source for
 2665 this shader:

```
2666 setCameraAndLights(c, l) {
2667     this.mCamera = c;
2668     this.mLight = l;
2669 }
```

2670 Instantiating Default Shadow Caster and Receiver Shaders

2671 Default instances of engine shaders must be created to connect to the newly defined
 2672 GLSL shader caster and receiver fragment shaders:

2673 1. Modify `shader_resources.js` in the `src/engine/core` folder
 2674 to import `ShadowCasterShader`, and define the constants and
 2675 variables for the two new shadow-related shaders.

```
2676 import ShadowCasterShader from "../shaders/shadow_caster_shader.js";
2677 let kShadowReceiverFS = "src/glsl_shaders/shadow_receiver_fs.glsl";
2678 let mShadowReceiverShader = null;
2679 let kShadowCasterFS = "src/glsl_shaders/shadow_caster_fs.glsl";
2680 let mShadowCasterShader = null;
```

2681 2. Edit the `createShaders()` function to define engine shaders to
 2682 interface to the new GLSL fragment shaders. Notice that both
 2683 of the engine shaders are based on the `texture_vs` GLSL vertex
 2684 shader. In addition, as discussed, a new instance of the engine
 2685 `SpriteShader` is created to interface to the `shadow_receiver_fs`
 2686 GLSL fragment shader.

```

function createShaders() { 2687
    ... identical to previous code ... 2688
    mIllumShader = new IllumShader(kTextureVS, kIllumFS); 2689
mShadowCasterShader = new ShadowCasterShader(2690
    kTextureVS, kShadowCasterFS); 2691
mShadowReceiverShader = new SpriteShader(2692
    kTextureVS, kShadowReceiverFS); 2693
}

```

3. The rest of the modifications to the `shader_resources.js` file are routine, including defining accessors, loading and unloading the GLSL source code files, cleaning up the shaders, and exporting the accessors. The detailed listings of these are not included here because you saw similar changes on many occasions. Please refer to the source code file for the actual implementations. 2695
2696
2697
2698
2699
2700

Configuring and Supporting WebGL Buffers

Three modifications are necessary to integrate WebGL stencil buffer into the game engine. First, the WebGL stencil buffer must be enabled and properly configured. Second, functions must be defined to support the drawing with stencil buffer. Third, buffer must be properly cleared before each drawing cycle. 2701
2702
2703
2704
2705

1. Edit the `gl.js` file in the `src/engine/core` folder to enable and configure WebGL stencil buffer during engine initialization. In the `init()` function, add the request for the allocation and configuration of stencil and depth buffers during WebGL initialization. Notice that the depth buffer, or z buffer, is also allocated and configured. This is necessary for proper shadow caster support, where a shadow caster must be in front of a receiver, or with a larger z depth in order to cast shadow on the receiver. 2706
2707
2708
2709
2710
2711
2712
2713
2714

CHAPTER 8 IMPLEMENTING ILLUMINATION AND SHADOW

```
2715 function init(htmlCanvasID) {  
2716     ... identical to previous code ...  
2717     mGL = mCanvas.getContext("webgl2",  
2718                             {alpha: false, depth: true, stencil: true} ||  
2719                             mCanvas.getContext("experimental-webgl2",  
2720                             {alpha: false, depth: true, stencil: true});  
2721     ... identical to previous code ...  
2722     // make sure depth testing is enabled  
2723     mGL.enable(mGL.DEPTH_TEST);  
2724     mGL.depthFunc(mGL.LEQUAL);  
2725 }  
2726 2. Continue working with gl.js; define functions to begin, end, and  
2727      disable drawing with the stencil buffer. Remember to export these  
2728      new stencil buffer support functions.  
2729 function beginDrawToStencil(bit, mask) {  
2730     mGL.clear(mGL.STENCIL_BUFFER_BIT);  
2731     mGL.enable(mGL.STENCIL_TEST);  
2732     mGL.colorMask(false, false, false, false);  
2733     mGL.depthMask(false);  
2734     mGL.stencilFunc(mGL.NEVER, bit, mask);  
2735     mGL.stencilOp(mGL.REPLACE, mGL.KEEP, mGL.KEEP);  
2736     mGL.stencilMask(mask);  
2737 }  
2738 function endDrawToStencil(bit, mask) {  
2739     mGL.depthMask(mGL.TRUE);  
2740     mGL.stencilOp(mGL.KEEP, mGL.KEEP, mGL.KEEP);  
2741     mGL.stencilFunc(mGL.EQUAL, bit, mask);  
2742     mGL.colorMask(true, true, true, true);  
2743 }  
2744 function disableDrawToStencil() { mGL.disable(mGL.STENCIL_TEST); }
```

3. Edit the engine access file, `index.js`, in the `src/engine` folder to
2745
clear the stencil and depth buffers when clearing the canvas in the
2746
`clearCanvas()` function:
2747

```
function clearCanvas(color) {  
    ... identical to previous code ...  
    gl.clear(gl.COLOR_BUFFER_BIT | gl.STENCIL_BUFFER_BIT |  
        gl.DEPTH_BUFFER_BIT);  
}
```

Defining Shadow Support for Game Developers

As described when defining `ShadowCasterShader`, `Renderable` classes should not be
2754
defined to pair with the shadow caster and receiver shaders as that would allow game
2755
developers the capabilities to manipulate the algorithmically created objects as regular
2756
game objects. Instead, the `ShadowCaster` and `ShadowReceiver` classes are introduced to
2757
allow the game developers to create shadows without granting access to manipulate the
2758
underlying geometries.
2759

Defining the Shadow Caster Class

Instead of the familiar `Renderable` class hierarchy, the `ShadowCaster` class is defined to
2761
encapsulate the functionality of the implicitly defined shadow caster geometry. Recall
2762
from Figure 8-25, the shadow caster geometry is derived algorithmically for each shadow
2763
casting light sources based on the positions of the shadow caster, a `Renderable`, and the
2764
shadow receiver, another `Renderable`, objects.
2765

To support receiving shadows on an animated sprite element, the shadow receiver
2766
must be of `SpriteRenderable` or its subclasses. The shadow casting `Renderable`
2767
object must be able to receive light sources and thus is must be of `LightRenderable`
2768
or its subclasses. A `ShadowCaster` object maintains references to the actual shadow
2769
casting and receiving `Renderable` objects and defines the algorithm to compute and
2770
render shadow caster geometries for each of the light sources referenced by the caster
2771
`LightRenderable` object. The details of the `ShadowCaster` class are as follows:
2772

- Create the `src/engine/shadows` folder for organizing shadow-
2773
related support files and the `shadow_caster.js` file in the folder.
2774

CHAPTER 8 IMPLEMENTING ILLUMINATION AND SHADOW

```
2775      2. Define the ShadowCaster class and the constructor to initialize  
2776          the instance variables and constants required for caster geometry  
2777          computations:  
  
2778 import * as shaderResources from "../core/shader_resources.js";  
2779 import SpriteRenderable from "../renderables/sprite_renderable.js";  
2780 import Transform from "../utils/transform.js";  
2781 import { eLightType } from "../lights/light.js";  
  
2782 // shadowCaster: GameObject referencing at least a LightRenderable  
2783 // shadowReceiver: GameObject referencing at least a SpriteRenderable  
2784 class ShadowCaster {  
2785     constructor(shadowCaster, shadowReceiver) {  
2786         this.mShadowCaster = shadowCaster;  
2787         this.mShadowReceiver = shadowReceiver;  
2788         this.mCasterShader = shaderResources.getShadowCasterShader();  
2789         this.mShadowColor = [0, 0, 0, 0.2];  
2790         this.mSaveXform = new Transform();  
  
2791         this.kCasterMaxScale = 3;    // Max amount a caster will be scaled  
2792         this.kVerySmall = 0.001;    //  
2793         this.kDistanceFudge = 0.01; // to avoid caster-receiver overlap  
2794         this.kReceiverDistanceFudge = 0.6;  
2795             // Factor to reduce the projected caster geometry size  
2796     }  
  
2797     setShadowColor(c) {  
2798         this.mShadowColor = c;  
2799     }  
  
2800     ... implementation to follow ...  
2801 }  
  
2802 export default ShadowCaster;
```

2803 The `mShadowCaster` is a reference to the shadow caster `GameObject` with at
2804 least a `LightRenderable`, and the `mShadowReceiver` is a `GameObject` with at least
2805 a `SpriteRenderable` render component. As will be detailed in the next step,
2806 `mCasterShader`, `mShadowColor`, and `mSaveXform` are variables to support the rendering of
2807 shadow caster geometries.

3. Implement the draw() function to compute and draw a shadow
 caster geometry for each of the light sources that illuminates the
 Renderable object of mShadowCaster:

```

draw(aCamera) {
    let casterRenderable = this.mShadowCaster.getRenderable();
    // Step A: save caster xform/shader/color. Set caster to shadow color
    this.mShadowCaster.getXform().cloneTo(this.mSaveXform);
    let s = casterRenderable.swapShader(this.mCasterShader);
    let c = casterRenderable.getColor();
    casterRenderable.setColor(this.mShadowColor);
    let l, lgt;
    // Step B: loop through each light, if shadow casting is on
    //           compute the proper shadow offset
    for (l = 0; l < casterRenderable.getNumLights(); l++) {
        lgt = casterRenderable.getLightAt(l);
        if (lgt.isLightOn() && lgt.isLightCastShadow()) {
            // Step C: turn caster into caster geometry
            //           draws as SpriteRenderable
            this.mSaveXform.cloneTo(this.mShadowCaster.getXform());
            if (this._computeShadowGeometry(lgt)) {
                this.mCasterShader.setCameraAndLights(aCamera, lgt);
                SpriteRenderable.prototype.draw.call(
                    casterRenderable, aCamera);
            }
        }
    }
    // Step D: restore the original shadow caster
    this.mSaveXform.cloneTo(this.mShadowCaster.getXform());
    casterRenderable.swapShader(s);
    casterRenderable.setColor(c);
}

```

CHAPTER 8 IMPLEMENTING ILLUMINATION AND SHADOW

2839 The `casterRenderable` is the `Renderable` object that is actually casting the shadow.

2840 The following are the four main steps of the `draw()` function:

- 2841 a. Step A saves the `caster Renderable` state, transform, shader,
2842 and color, and sets it into a shadow `caster geometry` by setting
2843 its shader to a `ShadowCasterShader` (`mCasterShader`) and its
2844 color to that of shadow color.
- 2845 b. Step B iterates through all light sources illuminating the
2846 `casterRenderable` and looks for lights that are switched on
2847 and casting shadow.
- 2848 c. Step C, for each shadow producing light, calls the
2849 `_computeShadowGeometry()` function to compute
2850 an appropriately size and positioned shadow `caster`
2851 geometry and renders it as a `SpriteRenderable`. With
2852 the replaced `ShadowCasterShader` and shadow color, the
2853 rendered geometry appears as the shadow of the actual
2854 `casterRenderable`.
- 2855 d. Step D restores the state of the `casterRenderable`.
- 2856 4. Define the `_computeShadowGeometry()` function to compute
2857 the shadow `caster geometry` based on the `mShadowCaster`, the
2858 `mShadowReceiver`, and a casting light source. Although slightly
2859 intimidating in length, the following function can be logically
2860 separated into four regions. The first region declares and
2861 initializes the variables. The second and third regions are the
2862 two cases of the `if` statement that handle the computation of
2863 transform parameters for directional and point/spotlights. The
2864 last region sets the computed parameters to the transform of the
2865 `caster geometry`, `cxf`.

```
2866 _computeShadowGeometry(aLight) {  
2867     // Region 1: declaring variables  
2868     let cxf = this.mShadowCaster.getXform();  
2869     let rxf = this.mShadowReceiver.getXform();  
2870     // vector from light to caster  
2871     let lgtToCaster = vec3.create();
```

```

let lgtToReceiverZ; 2872
let receiverToCasterZ; 2873
let distToCaster, distToReceiver; // along the lgtToCaster vector 2874
let scale; 2875
let offset = vec3.fromValues(0, 0, 0); 2876

receiverToCasterZ = rxf.getZPos() - cxf.getZPos(); 2877
if (aLight.getLightType() === eLightType.eDirectionalLight) { 2878
    // Region 2: Processing a directional light 2879
    if (((Math.abs(aLight.getDirection())[2]) < this.kVerySmall) || 2880
        ((receiverToCasterZ * (aLight.getDirection())[2]) < 0)) { 2881
        return false; // direction light casting side way or 2882
        // caster and receiver on different sides of light in Z 2883
    }
    vec3.copy(lgtToCaster, aLight.getDirection()); 2885
    vec3.normalize(lgtToCaster, lgtToCaster); 2886

    distToReceiver = Math.abs(receiverToCasterZ / lgtToCaster[2]); 2887
                                // measured along lgtToCaster 2888
    scale = Math.abs(1 / lgtToCaster[2]); 2889
} else { 2890
    // Region 3: Processing a point or spot light 2891
    vec3.sub(lgtToCaster, cxf.get3DPosition(), aLight.getPosition()); 2892
    lgtToReceiverZ = rxf.getZPos() - (aLight.getPosition())[2]; 2893

    if ((lgtToReceiverZ * lgtToCaster[2]) < 0) { 2894
        return false; // caster and receiver 2895
                    // on different sides of light in Z 2896
    }
    if ((Math.abs(lgtToReceiverZ) < this.kVerySmall) || 2898
        ((Math.abs(lgtToCaster[2]) < this.kVerySmall))) { 2899
        // almost the same Z, can't see shadow 2900
        return false; 2901
    }
    distToCaster = vec3.length(lgtToCaster); 2903
    vec3.scale(lgtToCaster, lgtToCaster, 1 / distToCaster); 2904
                                // normalize lgtToCaster 2905
}

```

CHAPTER 8 IMPLEMENTING ILLUMINATION AND SHADOW

```

2906     distToReceiver = Math.abs(receiverToCasterZ / lgtToCaster[2]);
2907                         // measured along lgtToCaster
2908     scale = (distToCaster +
2909             (distToReceiver * this.kReceiverDistanceFudge)) /
2910             distToCaster;
2911 }
2912 vec3.scaleAndAdd(offset, cxf.get3DPosition(),
2913                     lgtToCaster, distToReceiver + this.kDistanceFudge);

2914 // Region 4: Setting casterRenderable xform
2915 cxf.setRotationInRad(cxf.getRotationInRad());
2916 cxf.setPosition(offset[0], offset[1]);
2917 cxf.setZPos(offset[2]);
2918 cxf.setWidth(cxf.getWidth() * scale);
2919 cxf.setHeight(cxf.getHeight() * scale);

2920 return true;
2921 }
```

The `aLight` parameter is the casting light source. The goals of this function are to compute and set the shadow caster geometry transform, `cxf`, by using the `aLight` to project the shadow caster onto the shadow receiver. As illustrated in Figure 8-29, there are two cases to consider for the size of the projected caster geometry. First, for a directional light source, the projected size is a constant. Second, for a point or spotlight, the projected size is a function of distance to the receiver. These are the two cases, Regions 2 and 3, of the `if` statement with the following details:

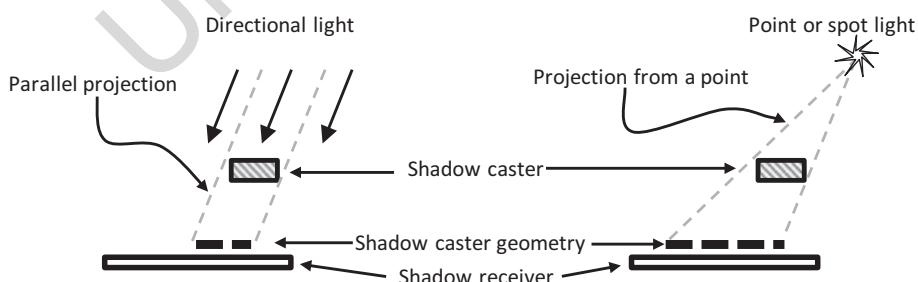


Figure 8-29. Computing the shadow caster geometry

- a. **Region 2:** Computes parallel projection according to the directional light. The if statement within this region is to ensure no shadow is computed when the light direction is parallel to the xy plan or when the light is in the direction from the shadow receiver toward the shadow caster. Notice that for dramatic effects, the shadow caster geometry will be moderately scaled. 2929
2930
2931
2932
2933
2934
2935
- b. **Region 3:** Computes projection from the point or spotlight position. The two if statements within this region are to ensure that the shadow caster and receiver are on the same side of the light position, and for the purpose of maintaining mathematical stability, neither is very close to the light source. 2936
2937
2938
2939
2940
- c. **Region 4:** Uses the computed distToReceiver and scale to set the transform of the shadow caster or cxf. 2941
2942

The ShadowCaster object is meant for game developers to define and work with shadow. So, remember to update the engine access file, index.js, to forward the newly defined functionality to the client. 2943
2944
2945

Defining the Shadow Receiver Class

Recall from Figure 8-25, ShadowReceiver is the object where the shadow of the caster object will appear. As illustrated in Figure 8-26, the ShadowReceiver must draw itself into the stencil buffer to ensure shadow caster geometry will only appear over pixels that are occupied by the ShadowReceiver object. 2947
2948
2949
2950

1. Create a new file, shadow_receiver.js, in the src/engine/shadows folder; define the ShadowReceiver class. In the constructor, initialize the constants and variables necessary for receiving shadows. As discussed, the mReceiver is a GameObject with at least a SpriteRenderable reference and is the actual receiver of the shadow. Notice that mShadowCaster is an array of ShadowCaster objects. These objects will cast shadows on the mReceiver. 2951
2952
2953
2954
2955
2956
2957
2958

```
import * as shaderResources from "../core/shader_resources.js";
import ShadowCaster from "./shadow_caster.js";
import * as glSys from "../core/gl.js"; 2959  
2960  
2961
```

CHAPTER 8 IMPLEMENTING ILLUMINATION AND SHADOW

```
2962 class ShadowReceiver {  
2963     constructor(theReceiverObject) {  
2964         this.kShadowStencilBit = 0x01;    // stencil bit for shadow  
2965         this.kShadowStencilMask = 0xFF;  // The stencil mask  
2966         this.mReceiverShader = shaderResources.getShadowReceiverShader();  
2967         this.mReceiver = theReceiverObject;  
2968         // To support shadow drawing  
2969         this.mShadowCaster = [];           // array of ShadowCasters  
2970     }  
2971     ... implementation to follow ...  
2972 }  
2973 export default ShadowReceiver;
```

2. Define the `addShadowCaster()` function to add a game object as a shadow caster for this receiver:

```
2976 addShadowCaster(lgtRenderable) {  
2977     let c = new ShadowCaster(lgtRenderable, this.mReceiver);  
2978     this.mShadowCaster.push(c);  
2979 }  
2980 // for now, cannot remove shadow casters
```

3. Define the `draw()` function to draw the receiver and all the shadow caster geometries:

```
2983 draw(aCamera) {  
2984     let c;  
2985     // Step A: draw receiver as a regular renderable  
2986     this.mReceiver.draw(aCamera);  
2987     // Step B: draw receiver into stencil to enable corresponding pixels  
2988     glSys.beginDrawToStencil(this.kShadowStencilBit,  
2989                             this.kShadowStencilMask);  
2990     //      Step B1: swap receiver shader to a ShadowReceiverShader  
2991     let s = this.mReceiver.getRenderable().swapShader(  
2992                                     this.mReceiverShader);
```

```

//           Step B2: draw the receiver again to the stencil buffer      2993
this.mReceiver.draw(aCamera);                                         2994
this.mReceiver.getRenderable().swapShader(s);                         2995
glSys.endDrawToStencil(this.kShadowStencilBit,                      2996
                      this.kShadowStencilMask);                     2997

// Step C: draw shadow color to pixels with stencil switched on      2998
for (c = 0; c < this.mShadowCaster.length; c++) {                   2999
    this.mShadowCaster[c].draw(aCamera);                            3000
}

// switch off stencil checking                                       3002
glSys.disableDrawToStencil();                                     3003
}

```

3004

This function implements the outlined shadow simulation algorithm and does not draw the actual shadow caster. Notice that the `mReceiver` object is drawn twice, in steps A and B2. Step A, the first `draw()` function, renders the `mReceiver` to the canvas as usual. Step B enables the stencil buffer for drawing where all subsequent drawings will be directed to switching on stencil buffer pixels. For this reason, the `draw()` function at step B2 uses the `ShadowReceiverShader` and switches on all pixels in the stencil buffer that corresponds to the `mReceiver` object. With the proper stencil buffer setup, in step C, the `draw()` function calls to the `mShadowCaster` will draw the corresponding shadow caster geometries only into the pixels that are covered by the receiver.

Lastly, once again, the `ShadowReceiver` object is designed for the client game developers to create shadows. So, remember to update the engine access file, `index.js`, to forward the newly defined functionality to the client.

Updating Engine Support

With the new objects defined and engine configured, some of the existing engine classes must be modified to support the new shadow operations. The following summarizes the required changes without listing the straightforward changes. Please refer to the source code file for the actual implementation details.

- `renderable.js`: Both the `ShadowCaster` and `ShadowReceiver` objects require the ability to swap the shaders to render the objects for shadow simulation purpose. This `swapShader()` function is best realized in the root of the `Renderable` hierarchy.

- `light.js`: The Light source now defines `mCastShadow`, a boolean variable, and the associated getter and setter, indicating if the light should cast shadow.
- `camera_main.js`: The Camera WC center must now be located at some z distance away. A `kCameraZ` constant is defined for this purpose and used in the `mCameraMatrix` computation in the `setViewAndCameraMatrix()` function.
- `transform.js`: The Transform class must be modified to support being `cloneTo()` and the manipulation of a z-depth value.

Testing the Shadow Algorithm

There are two important aspects to testing the shadow simulation. First, you must understand how to program and create shadow effects based on the implementation. Second, you must verify that Renderable objects can serve as shadow casters and receivers. The `MyGame` level test case is similar to the previous project with the exception of the shadow setup and drawing.

Setting Up the Shadow

The proper way of setting up the shadow system is to create all `ShadowCaster` objects and then create and add to the `ShadowReceiver` objects. The `my_game_shadow.js` file defines the `_setupShadow()` function to demonstrate this.

```
MyGame.prototype._setupShadow = function () {
    // mLgtMinion has a LightRenderable
    this.mLgtMinionShadow = new engine.ShadowReceiver(this.mLgtMinion);
    this.mLgtMinionShadow.addShadowCaster(this.mIllumHero);
    this.mLgtMinionShadow.addShadowCaster(this.mLgtHero);

    // mIllumMinion has a SpriteAnimateRenderable
    this.mMinionShadow = new engine.ShadowReceiver(this.mIllumMinion);
    this.mMinionShadow.addShadowCaster(this.mIllumHero);
    this.mMinionShadow.addShadowCaster(this.mLgtHero);
    this.mMinionShadow.addShadowCaster(this.mLgtMinion);
```

```
// mBg has a IllumRenderable
3055
this.mBgShadow = new engine.ShadowReceiver(this.mBg);
3056
this.mBgShadow.addShadowCaster(this.mLgtHero);
3057
this.mBgShadow.addShadowCaster(this.mIllumMinion);
3058
this.mBgShadow.addShadowCaster(this.mLgtMinion); }
```

The `_setupShadow()` function is called at the end of the `MyGame.init()` function, when all other `GameObject` instances are properly created and initialized. This function demonstrates that different types of `Renderable` objects can serve as shadow receivers.

- `LightRenderable`: `mLgtMinionShadow` is created with `mLgtMinon` as a receiver, which has a reference to a `LightRenderable` object.
- `IllumRenderable`: `mBgShadow` and `mMinionShadow` are created with `mBg` and `mIllumMinion` being the receivers where both have references to `IllumRenderable` object.

Note that in order to observe shadow on an object, an explicit corresponding `ShadowReceiver` must be created and followed by explicitly adding `ShadowCaster` objects to the receiver. For example, `mLgtMinionShadow` defines the `mLgtMinion` object to be a receiver, where only the `mIllumHero` and `mLgtHero` will cast shadows on this object. Lastly, notice that `mLgtMinon` and `mIllumMinion` are both receivers and casters of shadows.

Drawing the Shadow

In 2D drawings, objects are drawn by overwriting the previously drawn objects. For this reason, it is important to draw the shadow receivers and the shadow caster geometries before drawing the shadow casters. The following `my_game.draw()` function in `my_game_main.js` illustrates the important drawing order of objects:

```
draw() {
  // Clear the canvas
  engine.clearCanvas([0.9, 0.9, 0.9, 1.0]); // clear to light gray
  // Set up the camera and draw
  this.mCamera.setViewAndCameraMatrix();
```

```
3084 // always draw shadow receivers first!
3085 this.mBgShadow.draw(this.mCamera); // also draws the receiver object
3086 this.mMinionShadow.draw(this.mCamera);
3087 this.mLgtMinionShadow.draw(this.mCamera);

3088 this.mBlock1.draw(this.mCamera);
3089 this.mIllumHero.draw(this.mCamera);
3090 this.mBlock2.draw(this.mCamera);
3091 this.mLgtHero.draw(this.mCamera);

3092 this.mMsg.draw(this.mCamera); // draw last
3093 this.mMatMsg.draw(this.mCamera);
3094 }
```

3095 It is important to note the draw ordering. All three shadow receivers are drawn first.
3096 Additionally, among the three receivers, the mBgShadow object is the actual background
3097 and thus is the first being drawn. Recall that in the definition of the ShadowReceiver
3098 class, the draw() function also draws the receiver object. For this reason, there is no need
3099 to call the draw() function of mLgtMinion, mIllumMinion, and mBg objects.

3100 The rest of the MyGame level is largely similar to previous projects and is not listed
3101 here. Please refer to the source code for the details.

3102 Observations

3103 You can now run the project and observe the shadows. Notice the effect of the stencil
3104 buffer where the shadow from the mIllumHero object is cast onto the minion and yet not
3105 on the background. Press the WASD keys to move both of the Hero objects. Observe how
3106 the shadows offer depth and distance cues as they move with the two hero objects. The
3107 mLgtHero on the right is illuminated by all four lights and thus casts many shadows. Try
3108 selecting and manipulating each of the lights, such as moving or changing the direction
3109 or switching the light on/off to observe the effects on the shadows. You can even try
3110 changing the color of the shadow (in shadow_caster.js) to something dramatic, such as
3111 to bright blue [0, 0, 5, 1], and observe shadows that could never exist in the real world.

Summary

3112

This chapter guided you to develop a variation of the simple yet complete Phong illumination model for the game engine. The examples were organized to follow the three terms of the Phong illumination model: ambient, diffuse, and specular. The coverage of light sources was strategically intermixed to ensure proper illumination can be observed for every topic discussed.

3113

3114

3115

3116

3117

The first example in this chapter on ambient illumination introduced the idea of interactively controlling and fine-tuning the color of the scene. The following two examples on light sources presented the notion that illumination, an algorithmic approach to color manipulation, can be localized and developed in the engine infrastructure for supporting the eventual Phong illumination model. The example on diffuse reflection and normal mapping was critical because it enabled illumination computation based on simple physical models and simulation of an environment in 3D.

3118

3119

3120

3121

3122

3123

3124

The Phong illumination model and the need for a per-object material property were presented in the specular reflection example. The halfway vector version of the Phong illumination model was implemented to avoid computing the light source reflection vector for each pixel. The light source type project demonstrated how subtle but important illumination variations can be accomplished by simulating different light sources in the real world. Finally, the last example explained that accurate shadow computation is nontrivial and introduced an approximation algorithm. The resulting shadow simulation, though inaccurate from a real-world perspective and with limitations, can be aesthetically appealing and is able to convey many of the same vital visual cues.

3125

3126

3127

3128

3129

3130

3131

3132

3133

3134

The first four chapters of this book introduced the basic foundations and components of a game engine. Chapters 5, 6, and 7 extended the core engine functionality to support drawing, game object behaviors, and camera controls, respectively. This chapter complements Chapter 5 by bringing the engine's capability in rendering higher-fidelity scenes to a new level. Over the next three chapters, this complementary pattern will be repeated. Chapter 9 will introduce physical behavior simulation, Chapter 10 will discuss particle effects, and Chapter 11 will complete the engine development with more advanced support for the camera including tiling and parallax.

3135

3136

3137

3138

3139

3140

3141

3142

3143

3144 Game Design Considerations

3145 The work you did in the “*Game Design Considerations*” section of Chapter 7 to create
3146 a basic well-formed game mechanic will ultimately need to be paired with the other
3147 elements of game design to create something that feels satisfying for players. In addition
3148 to the basic game loop, you’ll need to think about your game’s systems, setting, and
3149 meta-game and how they’ll help determine the kinds of levels you design. As you begin
3150 to define the setting, you’ll begin exploring ideas for visual and audio design.

3151 As is the case with most visual art, games rely in no small measure on effectively using
3152 lighting to convey setting. A horror game taking place in a graveyard at midnight will
3153 typically use a very different lighting model and color palette than a game focusing on
3154 upbeat, happy themes. Many people think that lighting applies primarily to games created
3155 in 3D engines that are capable of simulating realistic light and shadow, but the notion of
3156 lighting applies to most 2D game environments as well; consider the example presented by
3157 Playdead studio’s 2D side-scrolling platform game *Limbo*, as shown in Figure 8-30.

this figure will be printed in b/w



Figure 8-30. Playdead and Double Eleven’s *Limbo*, a 2D side-scrolling game making clever use of background lighting and chiaroscuro techniques to convey tension and horror. Lighting can be both programmatically generated and designed into the color palettes of the images themselves by the visual artist and is frequently a combination of the two (image copyright Playdead media; please see www.playdead.com/limbo for more information)

Lighting is also often used as a core element of the game loop in addition to setting the mood; a game where the player is perhaps navigating in the dark with a virtual flashlight is an obvious example, but lights can also indirectly support game mechanics by providing important information about the game environment. Red pulsing lights often signal dangerous areas, certain kinds of green environment lights might signal either safe areas or areas with deadly gas, flashing lights on a map can help direct players to important locations, and the like.

In the *Simple Global Ambient* project, you saw the impact that colored environment lighting has on the game setting. In that project, the hero character moves in front of a background of metallic panels, tubes, and machinery, perhaps the exterior of a space ship. The environment light is red and can be pulsed—notice the effect on mood when the intensity is set to a comparatively low 1.5 vs. when it's set to something like a supersaturated 3.5, and imagine how the pulsing between the two values might convey a story or increase tension. In the *Simple Light Shader: One Light Source* project, a light was attached to the hero character (a point light in this case), and you can imagine that the hero must navigate the environment to collect objects to complete the level that are visible only when illuminated by the light (or perhaps activate objects that switch on only when illuminated).

The *Diffuse Shader with Multiple Light Sources* project illustrated how various light sources and colors can add considerable visual interest to an environment (sometimes referred to as localized environment lighting). Varying the types, intensities, and color values of lights often makes environments appear more alive and engaging because the light you encounter in the real world typically originates from many different sources. The other projects in this chapter all served to similarly enhance the sense of presence in the game level; as you work with diffuse shaders, normal maps, specularity, different light types, and shadows, consider how you might integrate some or all of these techniques into a level's visual design to make game objects and environments feel more vibrant and interesting.

Before you begin thinking about how lighting and other design elements might enhance the game setting and visual style, let us return for a moment to the simple game mechanic project from the “Game Design Considerations” section of Chapter 7 and consider how you might think about adding lighting to the mechanic to make the puzzle more engaging. Figure 8-31 begins with the basic mechanic from the end of the exercise.

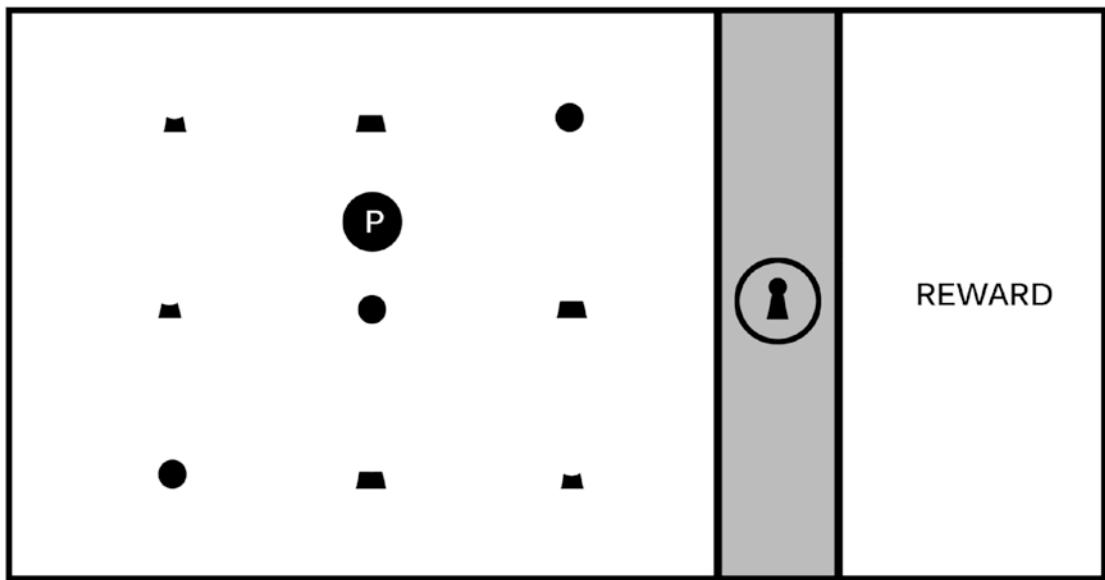


Figure 8-31. The simple game mechanic project, without lighting. Recall that the player controls the circle labeled with a P and must activate each of the three sections of the lock in proper sequence to disengage the barrier and reach the reward

3191 For the next phase of the simple game mechanic project, how might you integrate
3192 light directly into the game loop so that it becomes part of gameplay? As with the
3193 previous exercise, minimizing complexity and limiting yourself to one addition or
3194 evolution to the current game loop at a time will help prevent the design from becoming
3195 overburdened or too complex. Start this phase of the exercise by considering all the
3196 different ways that light might impact the current game screen. You might choose to
3197 have a dark environment where the player sees only shadowy shapes unless illuminating
3198 an area with a flashlight, you might use colored light to change the visible color of
3199 illuminated objects, or you might use something like an X-ray or ultraviolet beam to
3200 reveal information about the objects that wouldn't be seen with the naked eye. For this
3201 example, you'll add one additional dimension to the simple sequence mechanic: a light
3202 beam that reveals hidden information about the objects, as shown in Figure 8-32.

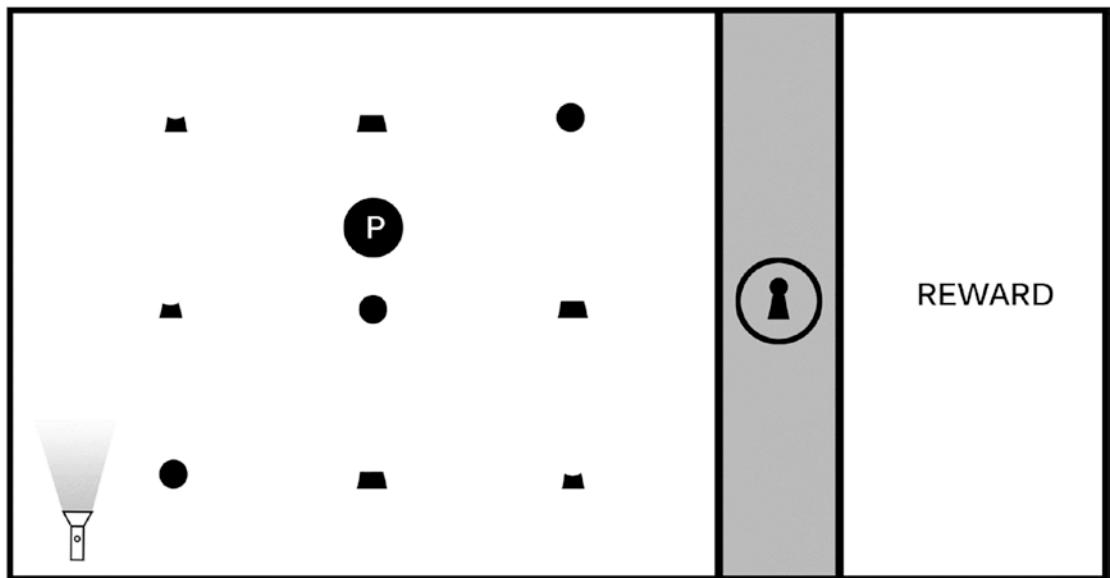


Figure 8-32. The addition of a movable “flashlight” that shines a special beam

In the first iteration of this game loop, the design required players to activate each segment of the lock in both the correct relative position (top on top, middle in the middle, bottom on bottom) and the correct order (top-middle-bottom). The interaction design provided consistent visual feedback for both correct and incorrect moves that allowed the player to understand the rules of play, and with some experimentation, astute players will deduce the proper sequence required to unlock the barrier. Now imagine how the addition of a special light beam might take the gameplay in a new direction: building on the basic notion of sequencing, you can create an increasingly clever puzzle requiring players to first discover the flashlight in the environment and experiment with it as a tool before making any progress on the lock. Imagine perhaps that the player can still directly activate the shapes when the hero character touches them even without the flashlight (triggering the highlight ring around the object as was the case in the first iteration, as shown in Figure 8-33), but that direct interaction is insufficient to activate the corresponding area of the lock unless the flashlight first reveals the secret clues required to understand the puzzle. Figure 8-34 shows the flashlight moved to illuminate one of the objects with its beam, revealing a single white dot.

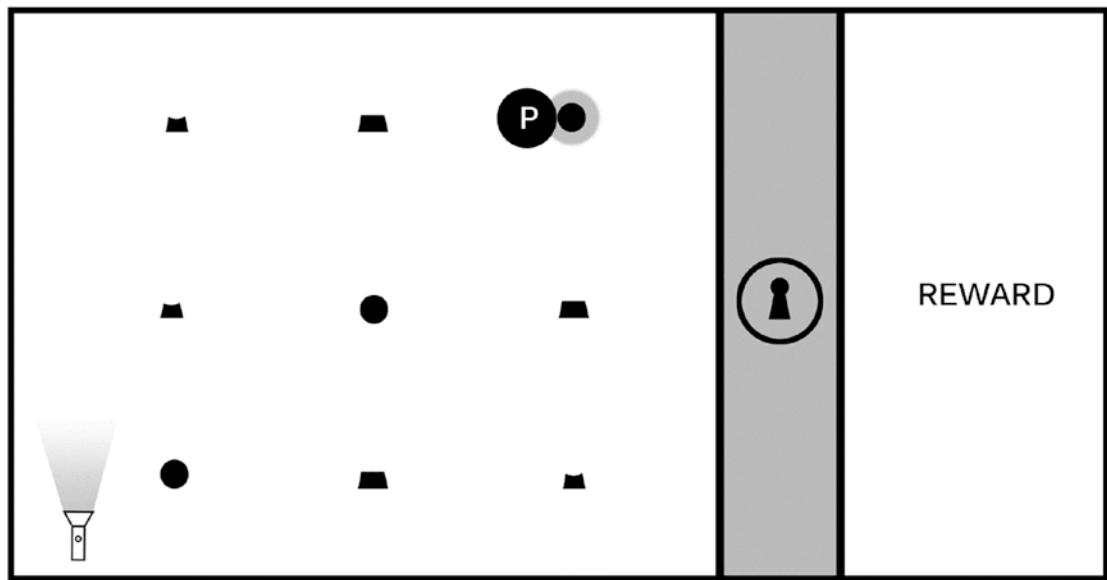


Figure 8-33. The player is able to directly activate the objects as in the first iteration of the mechanic, but the corresponding section of the lock now remains inactive

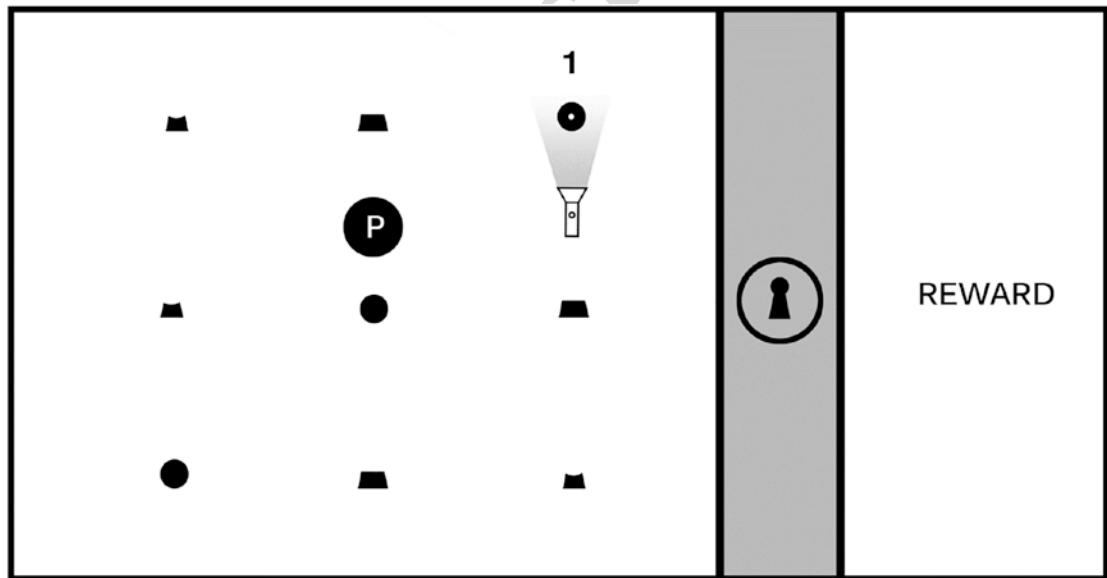
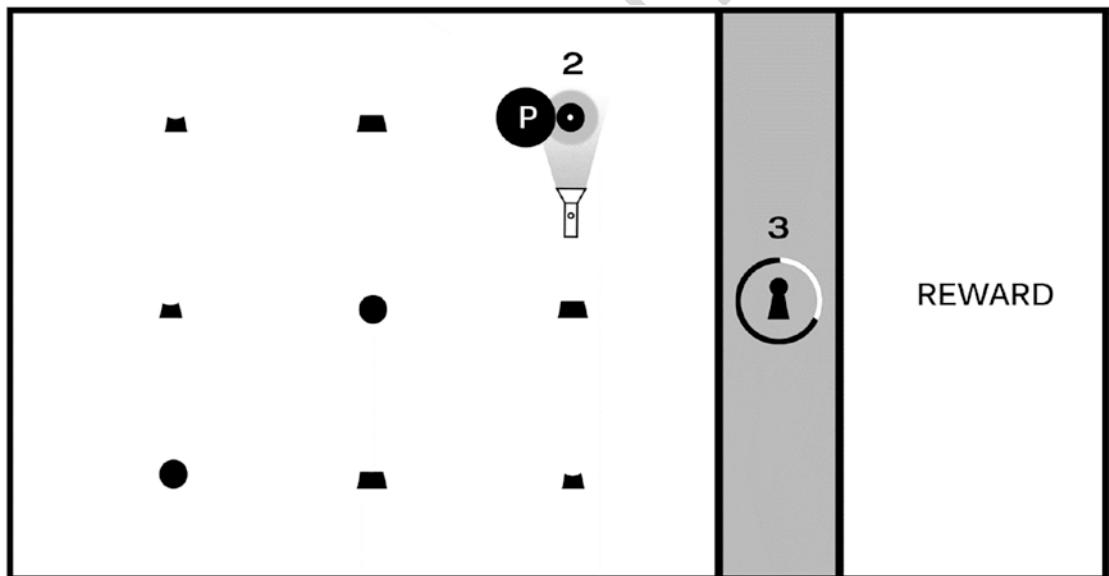


Figure 8-34. The player moves the flashlight under one of the shapes to reveal a hidden clue (#1)

From the gameplay point of view, any object in a game environment can be used as a tool; your job as a designer is to ensure the tool follows consistent, logical rules the player can first understand and then predictively apply to achieve their goal. In this case, it's reasonable to assume that players will explore the game environment looking for tools or clues; if the flashlight is an active object, players will attempt to learn how it functions in the context of the level.

The game loop in our sample project is evolving with the flashlight but uses the same basic sequencing principles and feedback metaphors. When the player reveals the secret symbol on the object with the flashlight, the player can begin the unlocking sequence by activating the object only when the symbol is visible. The new design requires players to activate each of the three objects corresponding to each section of the lock in the correct order, in this case from one dot to three dots; when all objects in a section are activated in order, that section of the lock will light up just as it did in the first iteration. Figures 8-35 to 8-37 show the new sequence using the flashlight beam.



this figure will be printed in b/w

Figure 8-35. With the flashlight revealing the hidden symbol, the player can now activate the object (#2), and a progress bar (#3) on the lock indicates the player is on the right track to complete a sequence

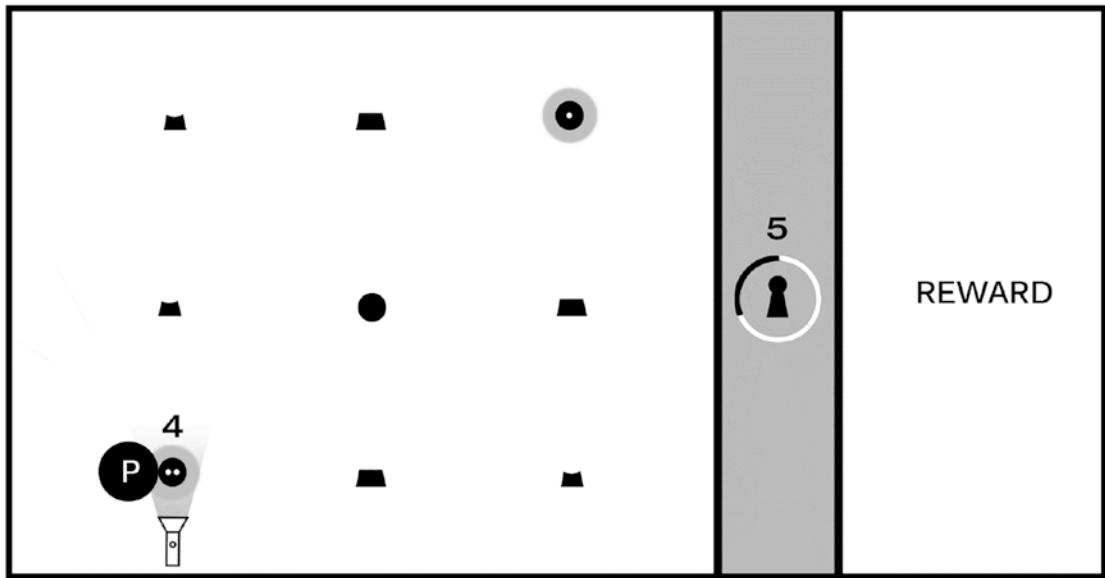


Figure 8-36. The player activates the second of the three top sections in the correct order (#4), and the progress bar confirms the correct sequence by lighting another section (#5). In this implementation, the player would not be able to activate the object with two dots before activating the object with one dot (the rules require activating like objects in order from one to three dots)

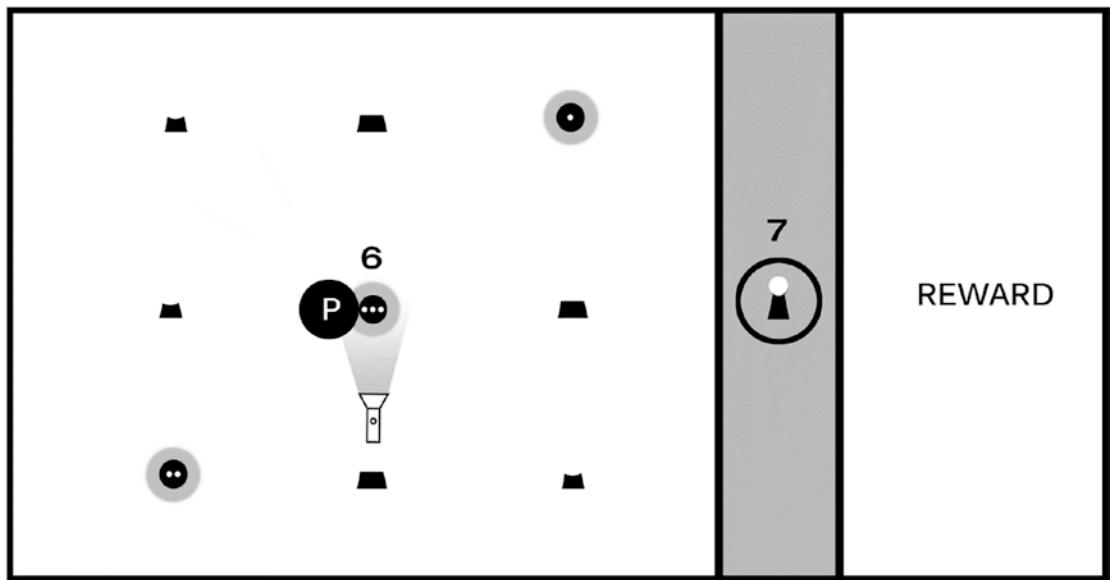


Figure 8-37. The third of the three top sections is revealed with the flashlight beam and activated by the player (#6), thereby activating the top section of the lock (#7). Once the middle and lower sections of the lock have been similarly activated, the barrier is disabled and players can claim the reward

Note that you've changed the feedback players receive slightly from the first iteration of the game loop: you originally used the progress bar to signal overall progress toward unlocking the barrier, but you're now using it to signal overall progress toward unlocking each section of the lock. The flashlight introduces an extra step into the causal chain leading to the level solution, and you've now taken a one-step elemental game loop and made something considerably more complex and challenging while maintaining logical consistency and following a set of rules that players can first learn and then predictively apply. In fact, the level is beginning to typify the kind of puzzle found in many adventure games: the game screen was a complex environment filled with a number of movable objects; finding the flashlight and learning that its beam reveals hidden information about objects in the game world would become part of the game setting itself.

3233
3234
3235
3236
3237
3238
3239
3240
3241
3242
3243

CHAPTER 8 IMPLEMENTING ILLUMINATION AND SHADOW

3244 It's important to be aware that as gameplay complexity increases, so also increases
3245 the complexity of the interaction model and the importance of providing players with
3246 proper audiovisual feedback to help them make sense of their actions (recall from
3247 Chapter 1 that the interaction model is the combination of keys, buttons, controller
3248 sticks, touch gestures, and the like that the player uses to accomplish game tasks). In
3249 the current example, the player is now capable of controlling not just the hero character
3250 but also the flashlight. Creating intuitive interaction models is a critical component of
3251 game design and often much more complex than designers realize; as one example,
3252 consider the difficulty in porting many PC games designed for a mouse and keyboard
3253 to a game console using buttons and thumb sticks or a mobile device using only touch.
3254 Development teams often pour thousands of hours of research and testing into control
3255 schemes, yet they still frequently miss the mark; interaction design is tricky to get right
3256 and often requires thousands of hours of testing and refinement even for fairly basic
3257 actions, so when possible, you should make use of established conventions. There are
3258 many books dedicated to examining interaction design in detail, but for the purposes of
3259 this book, you should keep the two golden rules in mind when you design interactions:
3260 first, use known and tested patterns when possible unless you have a compelling reason
3261 to ask players to learn something new; second, keep the number of unique actions
3262 players must remember to a minimum. Decades of user testing have clearly shown that
3263 players don't enjoy relearning basic key combinations for tasks that are similar across
3264 titles (which is why so many games have standardized on WASD for movement, e.g.),
3265 and similar data is available showing how easily players can become overwhelmed when
3266 you ask them to remember more than a few simple unique button combinations. There
3267 are exceptions, of course; many classic arcade fighting games, for example, use dozens
3268 of complex combinations, but those genres are targeted to a specific kind of player who
3269 considers mastering button combinations to be a fundamental component of what
3270 makes an experience fun. As a general rule, most players prefer to keep interaction
3271 complexity as streamlined and simple as possible if it's not an intentional component of
3272 play.

3273 There are a number of ways to deal with controlling multiple objects. The most
3274 common pattern for our flashlight would likely be for the player to "equip" it; perhaps
3275 if the player moves over the flashlight and clicks the left mouse button, it becomes a
3276 new ability for the player that can be activated by pressing one of the keyboard keys or
3277 by clicking the right mouse button. Alternately, perhaps the hero character can move
3278 around the game screen freely with the WASD keys, while other active objects like the

flashlight are first selected with a left-mouse-click and then moved by holding the left mouse button and dragging them into position. There are similarly a variety of ways to provide the player with contextual feedback that will help teach the puzzle logic and rules (in this case, we're using the ring around the lock as a progress bar to confirm players are following the correct sequence). As you experiment with various interaction and feedback models, it's always a good idea to review how other games have handled similar tasks, paying particular attention to things you believe to work especially well.

In the next chapter, you'll investigate how your game loop can evolve once again by applying simple physics to objects in the game world.

Author Queries

Chapter No.: 8 0005244101

Queries	Details Required	Author's Response
AU1	Please check if edit to sentence starting “As you can...” is okay.	
AU2	Please check if edit to sentence starting “The fonts folder ...” is okay.	
AU3	Please check if edit to sentence starting “In concert with...” is okay.	
AU4	Please check if edit to sentence starting “As illustrated in...” is okay.	
AU5	Projects, namely, “ <i>Light Source Project</i> ”, “ <i>Simple Light Shader: One Light Source project</i> ” and “ <i>Diffuse Shader with Multiple Light Sources project</i> ” do not exactly match to any of projects mentioned in the text nor in the previous chapters. Please check and confirm.	
AU6	Please check if edit to sentence starting “Type the 2 or 3...” is okay.	
AU7	Please check if edit to sentence starting “The controls of this ...” is okay.	
AU8	Please check if “must be of SpriteRenderable or its subclasses” is okay as edited.	

CHAPTER 9

1

Simulating the World with RigidShapes

2

3

After completing this chapter, you will be able to

- Recognize the significant computational complexity and cost of simulating real-world physical interactions
- Understand that typical game engine physics components approximate physical interaction based on simple geometries such as circles and rectangles
- Implement accurate collisions of circle and rectangular geometric shapes
- Approximate Newtonian motion formulation with Symplectic Euler Integration
- Resolve interpenetrating collisions based on a numerically stable relaxation method
- Compute and implement responses to collisions that resemble the behavior of rigid bodies in the real world

4

5

6

7

8

9

10

11

12

13

14

15

16

17

Introduction

18

In a game engine, the functionality of simulating energy transfer is often referred to as the physics, physics system, physics component, or physics engine. Game engine physics components play an important role in many types of games. The range of topics within physics for games is broad and includes but is not limited to areas such as rigid body, soft body, fluid dynamics, and vehicle physics. Believable physical behaviors

19

20

21

22

23 AUI

CHAPTER 9 SIMULATING THE WORLD WITH RIGIDSHAPES

and interactions of game objects have become key elements of many modern PC and console games, as well as, more recently, browser and smartphone games, for example, the bouncing of a ball, the wiggling of a jelly block, the ripples on a lake, or the skidding of a car. The proper simulation and realistic renditions of these are becoming common expectations.

Unfortunately, accurate simulations of the real world can involve details that are overwhelming and require in-depth disciplinary knowledge where the underlying mathematical models can be complicated and the associated computational costs prohibitive. For example, the skid of a car depends on its speed, the tire properties, etc.; the ripples on a lake depend on its cause, the size of the lake, etc.; the wiggle of a jelly block depends on its density, the initial deformation, etc. Even in the very simple case, the bounce of a ball depends on its material, the state of inflation, and, theoretically, even on the particle concentrations of the surrounding air. Modern game engine physics components address these complexities by restricting the types of physical interaction and simplifying the requirements for the simulation computation.

Physics engines typically restrict and simulate isolated types of physical interaction and do not support general combinations of interaction types. For example, the proper simulation of a ball bouncing (rigid body) often will not support the ball colliding and jiggling a jelly block (soft body) or accurately simulate the ripple effects caused by the ball interaction with fluid (fluid dynamics). That is, typically a rigid body physics engine does not support interactions with soft body objects, fluids, or vehicles. In the same manner, a soft body physics engine usually does not allow interactions with rigid body or other types of physical objects.

Additionally, physics engines typically approximate a vastly simplified interaction model while focusing mainly on attaining visually convincing results. The simplifications are usually in the forms of assumptions on object geometry and physical properties with restrictive interaction rules applied to a selective subset in the game world. For example, a rigid body physics engine typically simplifies the interactions of objects in the following ways:

- Assumes objects are continuous geometries with uniformly distributed mass where the center of mass is located at the center of the geometric shape
- Approximates object material properties with straightforward bounciness and friction

- Dictates that objects do not change shape during interactions 58
- Limits the simulation to a selective subset of objects in the game scene 59

Based on this set of assumptions, a rigid body physics simulation, or a rigid body simulation, is capable of capturing and reproducing many familiar real-world physical interactions such as objects bouncing, falling, and colliding, for example, a fully inflated bouncing ball or a simple *Lego* block bouncing off of a desk and landing on a hardwood floor. These types of rigid body physical interactions can be reliably simulated in real time as long as deformation does not occur during collisions. 61 AU2
62
63
64
65
66

Objects with uniformly distributed mass that do not change shape during interactions can be applicable to many important and useful scenarios in games. In general, rigid body physics engines are excellent for simulating moving objects coming into contact with one another such as a bowling ball colliding with pins or a cannon ball hitting an armored plate. However, it is important to recognize that with the given set of assumptions, a rigid body physics simulation does not support the following: 67
68
69
70
71
72

- Objects consisting of multiple geometric parts, for example, an arrow 73
- Objects with nontrivial material properties, for example, magnetism 74
- Objects with nonuniform mass distribution, for example, a baseball bat 75
- Objects that change shapes during collision, for example, rubber balls 76

Of all real-world physical object interaction types, rigid body interaction is the best understood, most straightforward to approximate solutions for, and least challenging to implement. This chapter focuses only on rigid body simulation. 78
79
80

Chapter Overview

Similar to illumination functionality, the physics component of a game engine is also a large and complex area of game engine design, architecture, and implementation. With this in mind, you will develop the rigid body physics component based on the same approach for all the previous game engine components. That is analyzing, understanding, and implementing individual steps to gradually realize the core functionality of the component. In the case of the physics component, the main ideas that encompass the rigid body simulation include the following: 81
82
83
84
85
86
87
88

CHAPTER 9 SIMULATING THE WORLD WITH RIGIDSHAPES

- **Rigid shape and bounds:** Define the RigidShape class to support an optimized simulation by performing computation on separate and simple geometries instead of the potentially complex Renderable objects. This topic will be covered by the first project, the Rigid Shapes and Bounds project.
- **Collision detection:** Examines and implements the mathematics to accurately collide circle and rectangle RigidShape objects. An important concept is that in the digital world, rigid shapes can and often do overlap, and it is essential to retain the details of this overlapping event in a CollisionInfo object. The topics covered on collision detection will be discussed by three separate projects, each focusing on a unique collision interaction. They include
 - The collisions between circle shapes: the Circle Collisions and Collision Info project
 - The collisions between rectangle shapes: the Rectangle Collisions project
 - The collisions between rectangle and circle shapes: the Rectangle and Circle Collisions project
- **Movement:** Approximates integrals that describe motions in a world that is updated at fixed intervals. The topic on motion will be covered by the Rigid Shape Movements project.
- Interpenetration of colliding objects: Addresses the interpenetration between colliding rigid shapes with a numerically stable solution to incrementally correct the situation. This topic is presented in the Collision Position Correction project.
- **Collision resolution:** Models the responses to collision with the Impulse Method. The Impulse Method will be covered in two projects, first the simpler case without rotations in the Collision Resolution project and finally with considerations for rotation in the Collision Angular Resolution project.

Rigid Shapes and Bounds

119

The computation involved in simulating the interactions between arbitrary rigid shapes can be algorithmically complicated and computationally costly. For these reasons, rigid body simulations are typically based on a limited set of simple geometric shapes, for example, rigid circles and rectangles. In typical game engines, these simple rigid shapes can be attached to geometrically complex game objects for an approximated simulation of the physical interactions between those game objects, for example, attaching rigid circles on spaceships and performing rigid body physics simulations on the rigid circles to approximate the physical interactions between the spaceships.

120

121^{AU4}

122

123^{AUS}

124

125

126

127

From real-world experience, you know that simple rigid shapes can interact with one another only when they come into physical contact. Algorithmically, this observation is translated into detecting collisions between rigid shapes. For a proper simulation, every shape must be tested for collision with every other shape. In this way, the collision testing is an $O(n^2)$ operation, where n is the number of shapes that participate in the simulation. As an optimization for this costly operation, rigid shapes are often bounded by a simple geometry, for example, a circle, where the potentially expensive collision computation is only invoked when the bounds of the shapes overlap.

128

129

130

131

132

133

134

135

The Rigid Shapes and Bounds Project

136

This project introduces the `RigidBody` classes with a simple circular bound for collision optimization. The defined `RigidBody` class will be integrated into the game engine where each `GameObject` object will have references to both a `Renderable` and a `RigidBody` object. The `Renderable` object will be drawn showing the players a visually pleasing gaming element, while the `RigidBody` will be processed in the rigid shape simulation approximating the behavior of the `GameObject` object. You can see an example of this project running in Figure 9-1. The source code to this project is defined in chapter9/9.1.rigid_shapes_and_bounds.

137

138

139

140

141

142

143

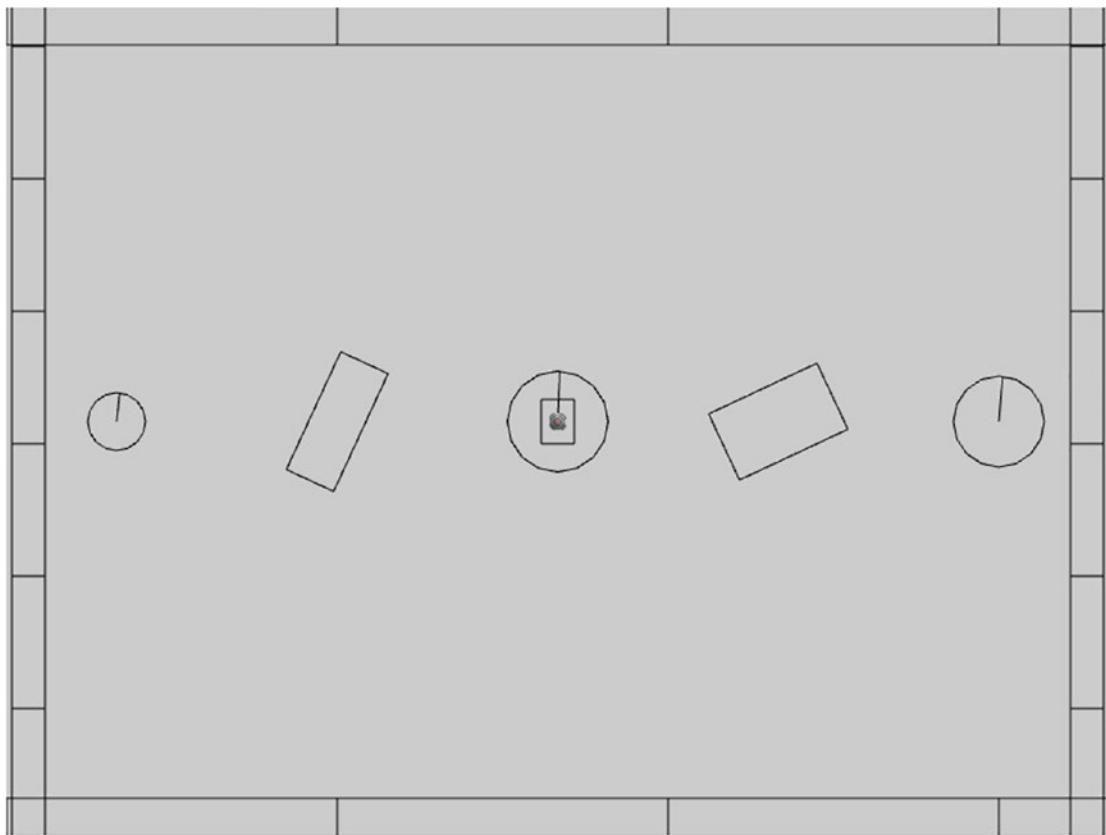


Figure 9-1. Running the Rigid Shapes and Bounds project

145 The controls of the project are as follows:

146 • **Behavior control:**

- 147 • **G key:** Randomly create a new rigid circle or rectangle

148 • **Draw control:**

- 149 • **T key:** Toggles textures on all objects

- 150 • **R key:** Toggles the drawing of RigidShape

- 151 • **B key:** Toggles the drawing of the bound on each RigidShape

152 • **Object control:**

- 153 • **Left-/right-arrow key:** Sequences through and selects an object

- 154 • **WASD keys:** Move the selected object.

- **Z/X key:** Rotates the selected object. 155
- **Y/U key:** Increases/decreases RigidShape size of the selected object; this does not change the size of the corresponding Renderable object. 156
157
158

The goals of the project are as follows: 159

- To define the RigidShape classes and integrate with GameObject 160
- To demonstrate that a RigidShape represents a corresponding 161
Renderable geometry on the same GameObject 162
- To lay the foundation for building a rigid shape physics simulator 163
- To define an initial scene for testing the physics component 164

In addition to the system font folder, you can find the following external resource files in the assets folder: 165
166

- minion_sprite.png is for the minion and hero objects. 167
- platform.png and wall.png are the horizontal and vertical boarder 168
objects in the test scene. 169
- target.png is displayed over the currently selected object. 170

Setting up Implementation Support 171

You will begin building this project by first setting up implementation support. First, organize the engine source code structure with new folders for anticipation of increases in complexity. Second, define debugging utilities for visualization and verification of correctness. Third, extend library support for rotating rigid shapes. 172
173
174
175

Organizing the Engine Source Code 176

In anticipation for the new components, in the src/engine folder, create the components folder and move the input.js component source code file into this folder. This folder will contain the source code for physics and other components to be introduced in later chapters. You will have to edit camera_input.js, loop.js, and index.js to update the source code file location change of input.js. 177
178
179
180
181

182 Supporting Debug Drawing

183 It is important to note that only a Renderable object, typically referenced by a
 184 GameObject, is actually visible in the game world. Rigid shapes do not actually exist in
 185 the game world; they are defined to approximate the simulation of physical interactions
 186 of corresponding Renderable objects. In order to support proper debugging and
 187 verification of correctness, it is important to be able to draw and visualize the rigid
 188 shapes.

- 189 1. In the `src/core` folder, create `debug_draw.js`, import from
 190 `LineRenderable`, and define supporting constants and variables
 191 for drawing simple shapes as line segments:

```
192 import LineRenderable from "../renderables/line_renderable.js";
193 let kDrawNumCircleSides = 16;      // approx circumference as line segments
194 let mUnitCirclePos = [];
195 let mLine = null;
```

- 196 2. Define the `init()` function to initialize the objects for drawing.
 197 The `mUnitCirclePos` are positions on the circumference of a unit
 198 circle, and `mLine` variable is the line object that will be used for
 199 drawing.

```
200 function init() {
201     mLine = new LineRenderable();
202     mLine.setPointSize(5); // make sure when shown, its visible
203     let deltaTheta = (Math.PI * 2.0) / kDrawNumCircleSides;
204     let theta = deltaTheta;
205     let i, x, y;
206     for (i = 1; i <= kDrawNumCircleSides; i++) {
207         let x = Math.cos(theta);
208         let y = Math.sin(theta);
209         mUnitCirclePos.push([x, y]);
210         theta = theta + deltaTheta;
211     }
212 }
```

3. Define the `drawLine()`, `drawCrossMarker()`, `drawRectangle()`,
and `drawCircle()` functions to draw the corresponding shape
based on the defined `mLine` object. The source code for these
functions is not relevant to the physics simulation and is not
shown. Please refer to the project source code folder for details. 213
214
215
216
217
4. Remember to export the defined functions: 218

```
export {  
    init,  
    drawLine, drawCrossMarker, drawCircle, drawRectangle  
}  
219  
220  
221  
222
```

Initialing the Debug Drawing Functionality 223

Edit `loop.js`, import from `debug_draw.js`, and call the `init()` function after all
asynchronous loading promises are fulfilled in `start()`: 224
225

```
import * as debugDraw from "./debug_draw.js";  
226  
  
... identical to previous code ...  
227  
  
async function start(scene) {  
    ... identical to previous code ...  
228  
    ...  
    // Wait for any async requests before game-load  
    await map.waitOnPromises();  
230  
231  
    // With all resources loaded, it is now possible to initialize  
    // system internal functions that depend on shaders, etc.  
    debugDraw.init(); // drawing support for rigid shapes, etc.  
232  
233  
234  
    ... identical to previous code ...  
235  
}  
236
```

237 **Note** A valid alternative for initializing debug drawing is in the
238 createShaders() function of the shader_resources module after all the
239 shaders are created. However, importing from debug_draw.js in shader_
240 resources.js would create a circular import: debug_draw imports from
241 LineRenderable that attempts to import from shader_resources.

242 **Updating the gl-matrix Library**

243 The gl-matrix library supports vertex translations with its vec2 addition and vertex
244 scaling with its vec2 scalar multiplication but does not support vertex rotations. Edit
245 src/lib/gl-matrix.js file, and define the vec2.rotateWRT() function to support
246 rotating a vertex position, pt, by angle with respect to the ref position. Following the
247 convention of gl-matrix, the first parameter of the function, out, returns the results of
248 the operation.

```
249 vec2.rotateWRT = function(out, pt, angle, ref) {  
250     var r=[];  
251     vec2.subtract(r, pt, ref);  
252     vec2.rotate(r, r, angle);  
253     vec2.add(r, r, ref);  
254     out[0] = r[0];  
255     out[1] = r[1];  
256     return r;  
257 };
```

258 **Defining the RigidShape Base Class**

259 You are now ready to define RigidShape to be the base class for the rectangle and circle
260 rigid shapes. This base class will encapsulate all the functionality that is common to the
261 two shapes.

- 262 1. Start by creating a new subfolder, rigid_shapes, in src/engine. In
263 this folder, create rigid_shape.js, import from debug_draw, and
 define drawing colors and the RigidShape class.

```
import * as debugDraw from "../core/debug_draw.js";          264  
let kShapeColor = [0, 0, 0, 1];                                265  
let kBoundColor = [1, 1, 1, 1];                                266  
  
class RigidBody {                                              267  
    ... implementation to follow ...                           268  
}                                                               269  
  
export default RigidBody;                                       270  
  
2. Define the constructor to include instance variables shared by 271  
all subclasses. The xf parameter is typically a reference to the 272  
Transform of the Renderable represented by this RigidBody.        273  
The mType variable will be initialized by subclasses to differentiate 274  
between shape types, for example, circle vs. rectangle. The 275  
mBoundRadius is the radius of the circular bound for collision 276  
optimization, and mDrawBounds indicates if the circular bound 277  
should be drawn.                                              278  
  
constructor(xf) {                                              279  
    this.mXform = xf;                                         280  
    this.mType = "";                                         281  
    this.mBoundRadius = 0;                                    282  
    this.mDrawBounds = false;                                 283  
}                                                               284  
  
3. Define appropriate getter and setter functions for the instance 285  
variables:                                                 286  
  
getType() { return this.mType; }                               287  
getCenter() { return this.mXform.getPosition(); }             288  
getBoundRadius() { return this.mBoundRadius; }                289  
toggleDrawBound() { this.mDrawBounds = !this.mDrawBounds; }   290  
setBoundRadius(r) { this.mBoundRadius = r; }                  291
```

CHAPTER 9 SIMULATING THE WORLD WITH RIGIDSHAPES

```
292 setTransform(xf) { this.mXform = xf; }
293 setPosition(x, y) { this.mXform.setPosition(x, y); }
294 adjustPositionBy(v, delta) {
295     let p = this.mXform.getPosition();
296     vec2.scaleAndAdd(p, p, v, delta);
297 }
298 _shapeColor() { return kShapeColor; }
299 _boundColor() { return kBoundColor; }
```

300 4. Define the `boundTest()` function to determine if the circular
301 bounds of two shapes have overlapped. As illustrated in
302 Figure 9-2, a collision between two circles can be determined by
303 comparing the sum of the two radii, `rSum`, with the distance, `dist`,
304 between the centers of the spheres. Once again, this is a relatively
305 efficient operation designed to precede the costlier accurate
306 collision computation between two shapes.

```
307 boundTest(otherShape) {
308     let vFrom1to2 = [0, 0];
309     vec2.subtract(vFrom1to2, otherShape.mXform.getPosition(),
310                   this.mXform.getPosition());
311     let rSum = this.mBoundRadius + otherShape.mBoundRadius;
312     let dist = vec2.length(vFrom1to2);
313     if (dist > rSum) {
314         // not overlapping
315         return false;
316     }
317     return true;
318 }
```

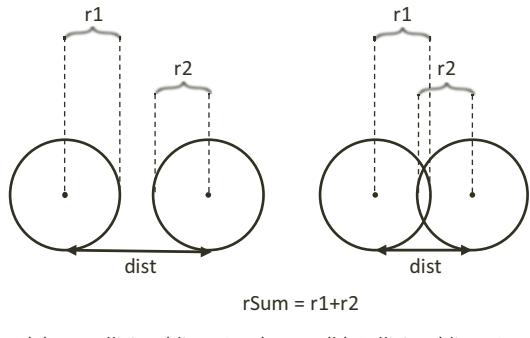


Figure 9-2. Circle collision detection: (a) No collision. (b) Collision detected

5. Define the update() and draw() functions. For now, update() is empty. When enabled, the draw() function draws the circular bound and an “X” marker at the center of the bound.

```

update() { // nothing for now }

draw(aCamera) {
    if (!this.mDrawBounds)
        return;
    debugDraw.drawCircle(aCamera, this.mXform.getPosition(),
                         this.mBoundRadius, this._boundColor());
    debugDraw.drawCrossMarker(aCamera, this.mXform.getPosition(),
                           this.mBoundRadius * 0.2, this._boundColor());
}

```

Defining the RigidRectangle Class

Renderable objects encode geometric information of a shape based on a Transform operator being applied on the unit square. For example, a rotated rectangle is encoded as a scaled and rotated unit square. As you have experienced, this representation, where vertices of the unit square remain constant together with the matrix transformation support from the GLSL vertex shader, is effective and efficient for supporting the drawing of transformed shapes.

338 RigidShapes are Renderable objects designed for interactions where the underlying
 339 representation must support extensive mathematical computations. In this case, it is
 340 more efficient to explicitly represent and update the vertices of the underlying geometric
 341 shape. For example, instead of a scaled and rotated square, the vertex positions of
 342 the rectangle can be explicitly computed and stored. In this way, the actual vertex
 343 positions are always readily available for mathematical computations. For this reason,
 344 RigidRectangle will define and maintain the vertices of a rectangle explicitly.

345 With the abstract base class for rigid shapes defined, you can now create the first
 346 concrete rigid shape, the RigidRectangle class. In anticipation of complex collision
 347 functions, the implementation source code will be separated into multiple files. For
 348 now, create the `rigid_rectangle.js` as the access file, and import from the `rigid_`
 349 `rectangle_main.js` which will implement the core RigidRectangle functionality.

- 350 1. In the `src/rigid_shapes` folder, create `rigid_rectangle.js`
 351 to import from `rigid_rectangle_main.js` and to export the
 352 RigidRectangle class. This is the RigidRectangle class access file
 353 where users of this class should import from.

```
354 import RigidRectangle from "./rigid_rectangle_main.js";
355 export default RigidRectangle;
```

- 356 2. Now, create `rigid_rectangle_main.js` in the `src/rigid_`
 357 `shapes` folder to import RigidBody and debugDraw, and define
 358 RigidRectangle to be a subclass of RigidBody.

```
359 import RigidBody from "./rigid_shape.js";
360 import * as debugDraw from "../core/debug_draw.js";
361 class RigidRectangle extends RigidBody {
362     ... implementation to follow ...
363 }
364 export default RigidRectangle;
```

- 365 3. Define the constructor to initialize the rectangle dimension,
 366 `mWidth` by `mHeight`, and `mType`. It is important to recognize
 367 that the vertex positions of the rigid rectangle are controlled
 368 by the `Transform` referenced by `mXform`. In contrast, the width
 369 and height dimensions are defined independently by `mWidth`

and `mHeight`. This dimension separation allows the designer
370
to determine how tightly a `RigidRectangle` should wrap the
371
corresponding `Renderable`. Notice that the actual vertex and face
372
normal of the shape are computed in the `setVertices()` and
373
`computeFaceNormals()` functions. The definition of face normal
374
will be detailed in the following steps:
375

```
constructor(xf, width, height) {  
    376  
    super(xf);  
    377  
    this.mType = "RigidRectangle";  
    378  
    this.mWidth = width;  
    379  
    this.mHeight = height;  
    380  
    this.mBoundRadius = 0;  
    381  
    this.mVertex = [];  
    382  
    this.mFaceNormal = [];  
    383  
  
    this.setVertices();  
    384  
    this.computeFaceNormals();  
    385  
}  
386
```

4. Define the `setVertices()` function to set the vertex positions
387
based on the dimension defined by `mXform`. As illustrated in
388
Figure 9-3, the vertices on the rectangle is defined as index 0 being
389
the top-left, 1 being top-right, 2 being bottom-right, and index 3
390
corresponds to the bottom-left vertex position.
391

```
setVertices() {  
    392  
    this.mBoundRadius = Math.sqrt(this.mWidth * this.mWidth +  
        393  
        this.mHeight * this.mHeight) / 2;  
    394  
    let center = this.mXform.getPosition();  
    395  
    let hw = this.mWidth / 2;  
    396  
    let hh = this.mHeight / 2;  
    397  
    // 0--TopLeft;1--TopRight;2--BottomRight;3--BottomLeft  
    398  
    this.mVertex[0] = vec2.fromValues(center[0] - hw, center[1] - hh);  
    399  
    this.mVertex[1] = vec2.fromValues(center[0] + hw, center[1] - hh);  
    400  
    this.mVertex[2] = vec2.fromValues(center[0] + hw, center[1] + hh);  
    401  
    this.mVertex[3] = vec2.fromValues(center[0] - hw, center[1] + hh);  
    402  
}  
403
```

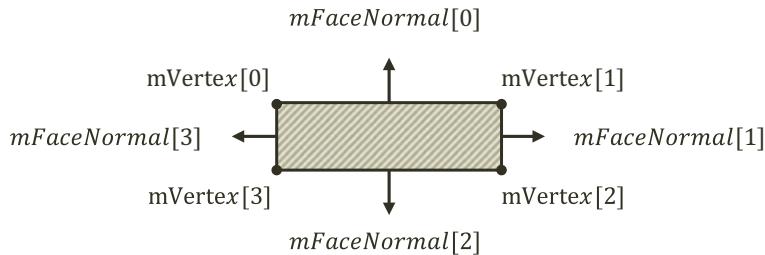


Figure 9-3. The vertices and face normals of a rectangle

- 404 5. Define the `computeFaceNormals()` function. Figure 9-3 shows that
 405 the face normals of a rectangle are vectors that are perpendicular
 406 to the edges and point away from the center of the rectangle.
 407 In addition, notice the relationship between the indices of the
 408 face normals and the corresponding vertices. Face normal index
 409 0 points in the same direction as the vector from vertex 2 to 1.
 410 This direction is perpendicular to the edge formed by vertices 0
 411 and 1. In this way, the face normal of index 0 is perpendicular to
 412 the first edge, and so on. Notice that the face normal vectors are
 413 normalized with length of 1. The face normal vectors will be used
 414 later for determining collisions.

```

415 computeFaceNormals() {
416   // 0--Top;1--Right;2--Bottom;3--Left
417   // mFaceNormal is normal of face toward outside of rectangle
418   for (let i = 0; i < 4; i++) {
419     let v = (i + 1) % 4;
420     let nv = (i + 2) % 4;
421     this.mFaceNormal[i] = vec2.clone(this.mVertex[v]);
422     vec2.subtract(this.mFaceNormal[i],
423                   this.mFaceNormal[i], this.mVertex[nv]);
424     vec2.normalize(this.mFaceNormal[i], this.mFaceNormal[i]);
425   }
426 }
```

6. Define the dimension and position manipulation functions. In
all cases, the vertices and face normals must be recomputed
(`rotateVertices()` calls `computeFaceNormals()`). In all cases, it is
critical to ensure that the vertex positions and the state of `mXform`
are consistent. 427

```

incShapeSizeBy(dt) { 432
    this.mHeight += dt; 433
    this.mWidth += dt; 434
    this.setVertices(); 435
    this.rotateVertices(); 436
}

setPosition(x, y) { 438
    super.setPosition(x, y); 439
    this.setVertices(); 440
    this.rotateVertices(); 441
}

adjustPositionBy(v, delta) { 443
    super.adjustPositionBy(v, delta); 444
    this.setVertices(); 445
    this.rotateVertices(); 446
}

setTransform(xf) { 448
    super.setTransform(xf); 449
    this.setVertices(); 450
    this.rotateVertices(); 451
}

rotateVertices() { 453
    let center = this.mXform.getPosition(); 454
    let r = this.mXform.getRotationInRad(); 455
    for (let i = 0; i < 4; i++) { 456
        vec2.rotateWRT(this.mVertex[i], this.mVertex[i], r, center); 457
    } 458
    this.computeFaceNormals(); 459
}

```

}

- 461 7. Now, define the `draw()` function to draw the edges of the
 462 rectangle as line segments, and the `update()` function to update
 463 the vertices of the rectangle. The vertices and face normals must
 464 be recomputed because, as you may recall from the `RigidShape`
 465 base class constructor discussion, the `mXfrom` is a reference
 466 to the `Transform` of a `Renderable` object; the game may have
 467 manipulated the position or the rotation of the `Transform`.
 468 To ensure `RigidRectangle` consistently reflect the potential
 469 `Transform` changes, the vertices and face normals must be
 470 recomputed at each update.

```
471 draw(aCamera) {  

472     super.draw(aCamera); // the cross marker at the center  

473     debugDraw.drawRectangle(aCamera, this.mVertex, this._shapeColor());  

474 }  

475 update() {  

476     super.update();  

477     this.setVertices();  

478     this.rotateVertices();  

479 }
```

480 Lastly, remember to update the engine access file, `index.js`, to forward the newly
 481 defined functionality to the client.

482 Defining the `RigidCircle` Class

483 You can now implement the `RigidCircle` class with a similar overall structure to that of
 484 `RigidRectangle`:

- 485 1. In the `src/rigid_shapes` folder, create `rigid_circle.js`
 486 to import from `rigid_circle_main.js` and to export the
 487 `RigidCircle` class. This is the `RigidCircle` class access file where
 488 users of this class should import from.

```
489 import RigidCircle from "./rigid_circle_main.js";  

490 export default RigidCircle;
```

2. Now, create `rigid_circle_main.js` in the `src/rigid_shapes` folder to import `RigidBody` and `debugDraw`, and define `RigidCircle` to be a subclass of `RigidBody`:

```
import RigidBody from "./rigid_shape.js";
import * as debugDraw from "../core/debug_draw.js";

class RigidCircle extends RigidBody {
    ... implementation to follow ...
}

export default RigidCircle;
```

3. Define the constructor to initialize the circle radius, `mRadius`, and `mType`. Similar to the dimension of a `RigidRectangle`, the radius of `RigidCircle` is defined by `mRadius` and is independent from the size defined by the `mXfrom`. Note that the radii of the `RigidCircle`, `mRadius`, and the circular bound, `mBoundRadius`, are defined separately. This is to ensure future alternatives to separate the two.

```
constructor(xf, radius) {
    super(xf);
    this.mType = "RigidCircle";
    this.mRadius = radius;
    this.mBoundRadius = radius;
}
```

4. Define the getter and setter of the dimension:

```
getRadius() { return this.mRadius; }

incShapeSizeBy(dt) {
    this.mRadius += dt;
    this.mBoundRadius = this.mRadius;
}
```

- 518 5. Define the function to draw the circle as a collection of line
 519 segments along the circumference. To properly visualize the
 520 rotation of the circle, a bar is drawn from the center to the rotated
 521 vertical circumference position.

```
522 draw(aCamera) {  

  523   let p = this.mXform.getPosition();  

  524   debugDraw.drawCircle(aCamera, p, this.mRadius,  

  525                         this._shapeColor());  

  526   let u = [p[0], p[1] + this.mBoundRadius];  

  527   // angular motion  

  528   vec2.rotateWRT(u, u, this.mXform.getRotationInRad(), p);  

  529   debugDraw.drawLine(aCamera, p, u,  

  530                      false, this._shapeColor()); // show rotation  

  531   super.draw(aCamera); // draw last to be on top  

  532 }
```

533 Lastly, remember to update the engine access file, `index.js`, to forward the newly
 534 defined functionality to the client.

535 **Modifying the GameObject Class to Integrate RightShape**

536 Recall from the discussions in Chapter 6, the `GameObject` class is designed to encapsulate
 537 the visual appearance and behaviors of objects in the game scene. The visual appearance
 538 of a `GameObject` is defined by the referenced `Renderable` object. Thus far, the behaviors
 539 of a `GameObject` have been defined and implemented as part of the `GameObject` class
 540 in the forms of an ad hoc traveling speed, `mSpeed`, and simple autonomous behavior,
 541 `rotateObjPointTo()`. You can now replace these ad hoc parameters with the upcoming
 542 systematic physics component support.

- 543 1. Edit `GameObject.js` to remove the support for speed, `mSpeed`,
 544 as well as the corresponding setter and getter functions and the
 545 `rotateObjPointTo()` function. Through the changes in the rest
 546 of this chapter, the game object behaviors will be supported
 547 by the rigid body physics simulation. Make sure to leave the
 548 other variables and functions alone; they are defined to support
 549 appearance and to detect texture overlaps, `pixelTouches()`.

2. In the constructor, define new instance variables to reference to a RigidShape and to provide drawing options: 551

```

class GameObject {
    constructor(renderable) {
        this.mRenderComponent = renderable;
        this.mVisible = true;
        this.mCurrentFrontDir = vec2.fromValues(0, 1); // front direction
        this.mRigidBody = null; 557
        this.mDrawRenderable = true; 558
        this.mDrawRigidShape = false; 559
    }
    ... implementation to follow ...
}

```

3. Define getter and setter for mRigidBody and functions for toggling drawing options: 563 564

```

getRigidBody() { return this.mRigidBody; } 565
setRigidBody(r) { this.mRigidBody = r; } 566

```

```

toggleDrawRenderable() { this.mDrawRenderable = !this.mDrawRenderable; } 567
toggleDrawRigidShape() { this.mDrawRigidShape = !this.mDrawRigidShape; } 568

```

4. Refine the draw() and update() functions to respect the drawing options and to delegate GameObject behavior update to the RigidShape class: 569 570 571

```

draw(aCamera) { 572
    if (this.isVisible()) {
        if (this.mDrawRenderable) 573
            this.mRenderComponent.draw(aCamera);
        if ((this.mRigidBody !== null) && (this.mDrawRigidShape)) 575
            this.mRigidBody.draw(aCamera);
    }
}

```

CHAPTER 9 SIMULATING THE WORLD WITH RIGIDSHAPES

```
580 update() {  
581     // simple default behavior  
582     if (this.mRigidBody !== null)  
583         this.mRigidBody.update();  
584 }
```

- 585 5. Edit the `game_object_set.js` file to modify the `GameObjectSet`
586 class to support the toggling of different drawing options for the
587 entire set:

```
588 ... identical to previous code ...
```

```
589 toggleDrawRenderable() {  
590     let i;  
591     for (i = 0; i < this.mSet.length; i++) {  
592         this.mSet[i].toggleDrawRenderable();  
593     }  
594 }  
  
595 toggleDrawRigidShape() {  
596     let i;  
597     for (i = 0; i < this.mSet.length; i++) {  
598         this.mSet[i].toggleDrawRigidShape();  
599     }  
600 }  
  
601 toggleDrawBound() {  
602     let i;  
603     for (i = 0; i < this.mSet.length; i++) {  
604         let r = this.mSet[i].getRigidBody()  
605         if (r !== null)  
606             r.toggleDrawBound();  
607     }  
608 }
```

Testing of RigidShape Functionality

RigidBody is designed to approximate and to participate on behalf of a Renderable object in the rigid shape simulation. For this reason, it is essential to create and test different combinations of RigidBody types, which includes circles and rectangles, with all combinations of Renderable types, more specifically, TextureRenderable, SpriteRenderable, and SpriteAnimateRenderable. The proper functioning of these combinations can demonstrate the correctness of the RigidBody implementation and allow you to visually examine the suitability as well as the limitations of approximating Renderable objects with simple circles and rectangles.

The overall structure of the test program, MyGame, is largely similar to previous projects where the details of the source code can be distracting and is not listed here. Instead, the following describes the tested objects and how these objects fulfill the specified requirements. As always, the source code files are located in `src/my_game` folder, and the supporting object classes are located in `src/my_game/objects` folder.

The testing of imminent collisions requires the manipulation of the positions and rotations of each object. The `WASDObj` class, implemented in `wasd_obj.js`, defines the WASD keys movement and Z/X keys rotation control of a `GameObject`. The `Hero` class, a subclass of `WASDObj` implemented in `hero.js`, is a `GameObject` with a `SpriteRenderable` and a `RigidRectangle`. The `Minion` class, also a subclass of `WASDObj` in `minion.js`, is a `GameObject` with `SpriteAnimateRenderable` and is wrapped by either a `RigidCircle` or a `RigidRectangle`. Based on these supporting classes, the created `Hero` and `Minion` objects encompass different combinations of Renderable and RigidBody types allowing you to visually inspect the accuracy of representing complex textures with different RigidShapes.

The vertical and horizontal bounds in the game scene are `GameObject` instances with `TextureRenderable` and `RigidRectangle` created by the `wallAt()` and `platformAt()` functions defined in `my_game_bounds.js` file. The constructor, `init()`, `draw()`, `update()`, etc., of `MyGame` are defined in the `my_game_main.js` file with largely identical functionality as in previous testing projects.

Observations

You can now run the project and observe the created RigidBody objects. Notice that by default, only RigidBody objects are drawn. You can verify this by typing the T key to toggle on the drawing of the Renderable objects. Notice how the textures of the Renderable objects are bounded by the corresponding RigidBody instances.

CHAPTER 9 SIMULATING THE WORLD WITH RIGIDSHAPES

642 You can type the R key to toggle off the drawing of the RigidBody objects. Normally,
643 this is what the players of a game will observe, with only the Renderable and without
644 the RigidBody objects being drawn. Since the focus of this chapter is on the rigid
645 shapes and the simulation of their interactions, the default is to show the RigidBody
646 and not the Renderable objects.

647 Now type the T and R keys again to toggle back the drawing of RigidBody objects.
648 The B key shows the circular bounds of the shapes. The more accurate and costly
649 collision computations to be discussed in the next few sections will only be incurred
650 between objects when these bounds overlap.

651 You can try using the WASD key to move the currently selected object around, by
652 default with the Hero in the center. The Z/X and Y/U keys allow you to rotate and change
653 the dimension of the Hero. Toggle on the texture, with the T key, to verify that rotation
654 and movement are applied to both the Renderable and its corresponding RigidBody
655 and that the Y/U keys only change the dimension of the RigidBody. This allows the
656 designer to control over how tightly to wrap the Renderable with the corresponding
657 RigidBody. You can type the left-/right-arrow keys to select and work with any of
658 the objects in the scene. Finally, the G key creates new Minion objects with either a
659 RigidCircle or a RigidRectangle.

660 Lastly, notice that you can move any selected object to any location, including
661 overlapping with another RigidBody object. In the real world, the overlapping, or
662 interpenetration, of rigid shape objects can never occur, while in the simulated digital
663 world, this is an issue that must be addressed. With the functionality of the RigidBody
664 classes verified, you can now examine how to compute the collision between these shapes.

665 Collision Detection

666 In order to simulate the interactions of rigid shapes, you must first detect which of
667 the shapes are in physical contact with one another or which are the shapes that have
668 collided. In general, there are two important issues to be addressed when working with
669 rigid shape collisions: computation cost and the situations when the shapes overlap, or
670 interpenetrate. In the following, the broad and narrow phase methods are explained as
671 an approach to alleviate the computational cost, and collision information is introduced
672 to record interpenetration conditions such that they can be resolved. This and the next
673 two subsections detail the collision detection algorithms and implementations of circle-
674 circle, rectangle-rectangle, and circle-rectangle collisions.

Broad and Narrow Phase Methods

675

As discussed when introducing the circular bounds for RigidShape objects, in general, every object must be tested for collision with every other object in the game scene. For example, if you want to detect the collisions between five objects, A, B, C, D, and E, you must perform four detection computations for the first object, A, against objects B, C, D, and E. With A and B's results computed, next you must perform three collision detections between the second object B, against objects C, D, and E; followed by two collisions for the third object, C; and then, finally, one for the fourth object, D. The fifth object, E, has already been tested against the other four. This testing process, while thorough, has its drawbacks. Without dedicated optimizations, you must perform $O(n^2)$ operations to detect the collisions between n objects.

676

677

678

679

680

681

682

683

684

685

In rigid shape simulation, a detailed collision detection algorithm involving intensive computations is required. This is because accurate results must be computed to support effective interpenetration resolution and realistic collision response simulation. A broad phase method optimizes this computation by exploiting the proximity of objects to rule out those that are physically far apart from each other and thus, clearly, cannot possibly collide. This allows the detailed and computationally intensive algorithm, or the narrow phase method, to be deployed for objects that are physically close to each other.

686

687

688

689

690

691

692

A popular broad phase method uses axis-aligned bounding boxes (AABBS) or bounding circles to approximate the proximity of objects. As detailed in Chapter 6, AABBS are excellent for approximating objects that are aligned with the major axes, but have limitations when objects are rotated. As you have observed from running the previous project with the B key typed, a bounding circle is a circle that centers around and completely bounds an object. By performing the straightforward bounding box/circle intersection computations, it becomes possible to focus only on objects with overlapping bounds as the candidates for narrow phase collision detection operations.

693

694

695

696

697

698

699

700

701

There are other broad phase methods that organize objects either with a spatial structure such as a uniform grid or a quad-tree or into coherent groups such as hierarchies of bounding colliders. Results from broad phase methods are typically fed into mid phase and finally narrow phase collision detection methods. Each phase narrows down candidates for the eventual collision computation, and each subsequent phase is incrementally more accurate and more expensive.

702

703

704

705

706

707

708

Collision Information

709 In addition to reporting if objects have collided, a collision detection algorithm should
 710 also compute and return information that can be used to resolve and respond to the
 711 collision. As you have observed when testing the previous project, it is possible for
 712 RigidShape objects to overlap in space, or interpenetrate. Since real-world rigid shape
 713 objects cannot interpenetrate, recording the details and resolving RigidShape overlaps is
 714 of key importance.

715 As illustrated in Figure 9-4, the essential information of a collision and
 716 interpenetration include collision depth, normal, start, and end. The collision depth is
 717 the smallest amount that the objects interpenetrated where the collision normal is the
 718 direction along which the collision depth is measured. The start and end are beginning
 719 and end positions of the interpenetration defined for the convenience of drawing the
 720 interpenetration as a line segment. It is always true that any interpenetration of convex
 721 objects can be resolved by moving the colliding objects along the collision normal by the
 722 collision depth magnitude or the distance from the start to the end position.

this figure will be printed in b/w

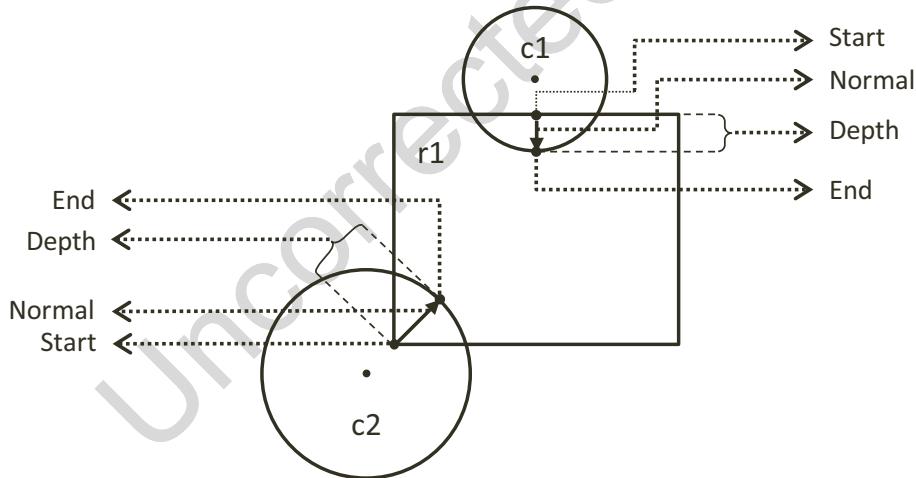


Figure 9-4. Collision information

The Circle Collisions and CollisionInfo Project

723

This project builds the infrastructure for computing and working with collision information based on collisions between circles. You can see an example of this project running in Figure 9-5. The source code to this project is defined in chapter9/9.2.circle_collisions_and_collision_info.

724

725

726

727

this figure will be printed in b/w

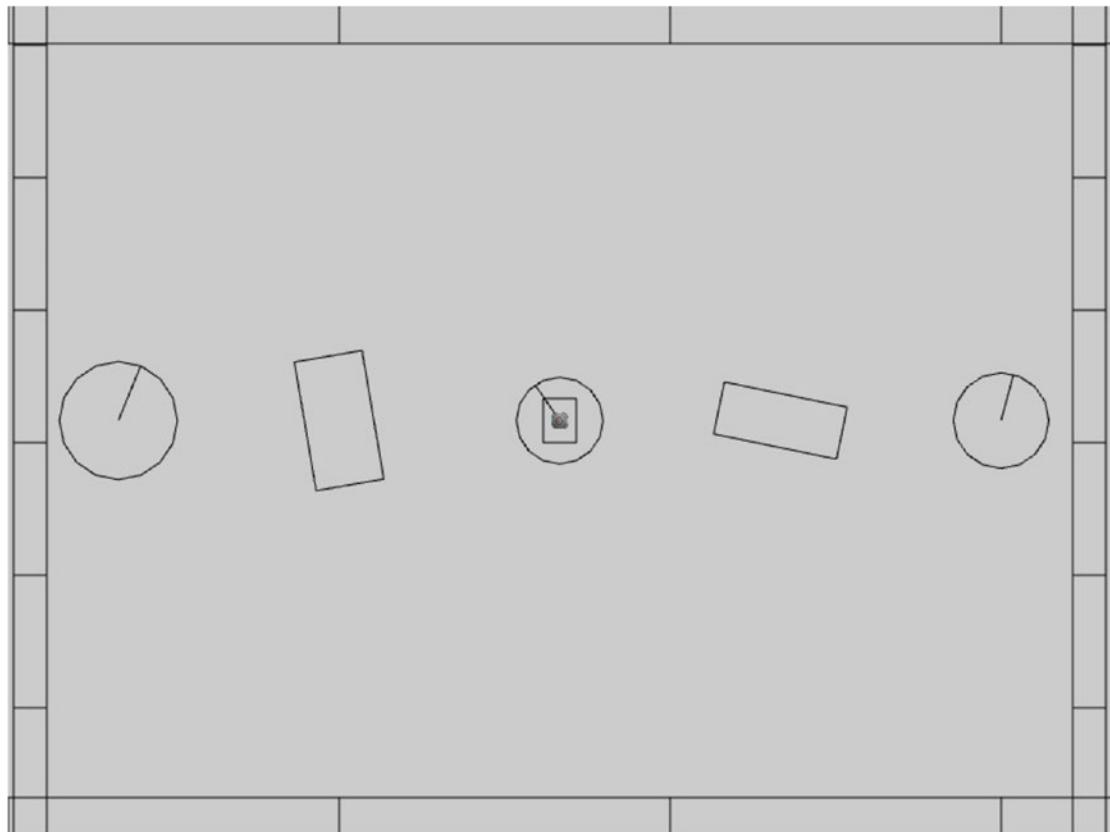


Figure 9-5. Running the CollisionInfo and Circle Collisions project

The controls of the project are identical to the previous project with a single addition of C key command in draw control:

728

729

- **Behavior control:**

730

- **G key:** Randomly create a new rigid circle or rectangle

731

- 732 • **Draw control:**
- 733 • **C key:** Toggles the drawing of all CollisionInfo
- 734 • **T key:** Toggles textures on all objects
- 735 • **R key:** Toggles the drawing of RigidBody
- 736 • **B key:** Toggles the drawing of the bound on each RigidBody
- 737 • **Object control:**
- 738 • **Left-/right-arrow key:** Sequences through and selects an object.
- 739 • **WASD keys:** Move the selected object.
- 740 • **Z/X key:** Rotates the selected object.
- 741 • **Y/U key:** Increases/decreases RigidBody size of the selected
- 742 object; this does not change the size of the corresponding
- 743 Renderable object.

744 The goals of the project are as follows:

- 745 • To understand the strengths and weaknesses of broad phase collision
- 746 detection
- 747 • To build the infrastructure for computing inter-circle collisions
- 748 • To define and work with collision conditions via the
- 749 CollisionInfo class
- 750 • To understand and implement circle collision detection algorithm

751 Defining the CollisionInfo Class

752 A new class must be defined to record RigidBody interpenetration situations as
 753 illustrated in Figure 9-4:

- 754 1. In the src/engine/rigid_shape folder, create the collision_
- 755 info.js file, import from debugDraw, declare the drawing color to
- 756 be magenta, and define the CollisionInfo class:

```
757 import * as debugDraw from "../core/debug_draw.js";
758 let kInfoColor = [1, 0, 1, 1]; // draw the info in magenta
```

```

class CollisionInfo {
    ... implementation to follow ...
}

```

```
export default CollisionInfo;
```

2. Define the constructor with instance variables that correspond to those illustrated in Figure 9-4 for collision depth, normal, and start and end positions:

```

constructor() {
    this.mDepth = 0;
    this.mNormal = vec2.fromValues(0, 0);
    this.mStart = vec2.fromValues(0, 0);
    this.mEnd = vec2.fromValues(0, 0);
}

```

3. Define the getter and setter for the variables:

```

getDepth() { return this.mDepth; }
setDepth(s) { this.mDepth = s; }

```

```

getNormal() { return this.mNormal; }
setNormal(s) { this.mNormal = s; }

```

```

getStart() { return this.mStart; }
getEnd() { return this.mEnd; }

```

```

setInfo(d, n, s) {
    this.mDepth = d;
    this.mNormal[0] = n[0];
    this.mNormal[1] = n[1];
    this.mStart[0] = s[0];
    this.mStart[1] = s[1];
    vec2.scaleAndAdd(this.mEnd, s, n, d);
}

```

4. Create a function to flip the direction of the collision normal. This function will be used to ensure that the normal is always pointing toward the object that is being tested for collision.

```

790 changeDir() {
791     vec2.scale(this.mNormal, this.mNormal, -1);
792     let n = this.mStart;
793     this.mStart = this.mEnd;
794     this.mEnd = n;
795 }

```

- 796 5. Define a `draw()` function to visualize the start, end, and collision
 797 normal in magenta:

```

798 draw(aCamera) {
799     debugDraw.drawLine(aCamera, this.mStart, this.mEnd, true, kInfoColor);
800 }

```

801 Lastly, remember to update the engine access file, `index.js`, to forward the newly
 802 defined functionality to the client.

803 **Modifying the RigidShape Classes**

804 RigidShape classes must be updated to support collisions. Since the abstract base
 805 shape, `RigidShape`, does not contain actual geometric information, the actual collision
 806 functions must be implemented in the rectangle and circle classes.

807 **Modifying the RigidRectangle Class**

808 For readability, collision support will be implemented in a separate source code file,
 809 `rigid_rectangle_collision.js`:

- 810 1. Modify `rigid_rectangle.js` to import from the new source
 811 code file:

```

812 import RigidRectangle from "./rigid_rectangle_collision.js";
813 export default RigidRectangle;

```

- 814 2. In the `src/engine/rigid_shapes` folder, create the `rigid_`
 815 `rectangle_collision.js` file, import `CollisionInfo` and
 816 `RigidRectangle`, and define the `collisionTest()` function
 817 to always return a collision failed status. Collisions with
 818 `RigidRectangle` shape will always fail until the next subsection.

```
RigidRectangle.prototype.collisionTest =
function (otherShape, collisionInfo) {
    let status = false;
    if (otherShape.mType === "RigidCircle") {
        status = false;
    } else {
        status = false;
    }
    return status;
}
```

3. Remember to export the extended RigidRectangle class for the clients:

```
export default RigidRectangle;
```

Modifying the RigidCircle Class

Modify the RigidCircle source code files in exactly the same manner as that of RigidRectangle: edit rigid_circle.js to import from rigid_circle_collision.js. Now, you are ready to implement circle-circle collision detection.

1. In the src/engine/rigid_shape folder, create the rigid_circle_collision.js file, import RigidCircle, and define the collisionTest() function to always return a collision failed status if the otherShape is not a RigidCircle; otherwise, call and return the status of collideCircCirc(). For now, a RigidCircle does not know how to collide with a RigidRectangle.

```
import RigidCircle from "./rigid_circle_main.js";

RigidCircle.prototype.collisionTest =
function (otherShape, collisionInfo) {
    let status = false;
    if (otherShape.mType === "RigidCircle") {
        status = this.collideCircCirc(this, otherShape, collisionInfo);
    } else {
```

```

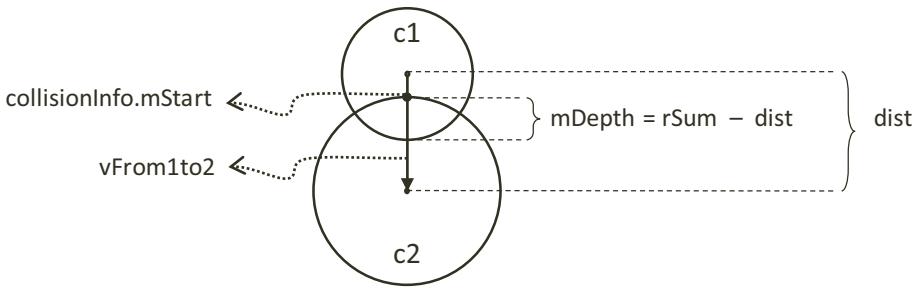
849         status = false;
850     }
851     return status;
852 }
```

- 853 2. Define the `collideCircCirc()` function to detect the collision
 854 between two circles and to compute the corresponding collision
 855 information when a collision is detected. There are three cases
 856 to the collision detection: no collision (step 1), collision with
 857 centers of the two circles located at different positions (step 2),
 858 and collision with the two centers located at exactly the same
 859 position (step 3). The following code shows step 1, the detection of
 860 no collision; notice that this code also corresponds to the cases as
 861 illustrated in Figure 9-2.

```

862 RigidCircle.prototype.collideCircCirc= function (c1, c2, collisionInfo) {
863     let vFrom1to2 = [0, 0];
864     // Step 1: Determine if the circles overlap
865     vec2.subtract(vFrom1to2, c2.getCenter(), c1.getCenter());
866     let rSum = c1.mRadius + c2.mRadius;
867     let dist = vec2.length(vFrom1to2);
868     if (dist > Math.sqrt(rSum * rSum)) {
869         // not overlapping
870         return false;
871     }
872     ... implementation of Steps 2 and 3 to follow ...
873 }
```

- 874 3. When a collision is detected, if the two circle centers are located
 875 at different positions (step 2), the collision depth and normal can
 876 be computed as illustrated in Figure 9-6. Since `c2` is the reference
 877 to the other `RigidBody`, the collision normal is a vector pointing
 878 from `c1` toward `c2` or in the same direction as `vFrom1to2`. The
 879 collision depth is the difference between `rSum` and `dist`, and the
 880 start position for `c1` is simply `c2-radius` distance away from the
 881 center of `c2` along the negative `mFrom1to2` direction.

**Figure 9-6.** Details of a circle-circle collision

```

// Step 1: refer to previous step           882
if (dist !== 0) {                         883
    // Step 2: Colliding circle centers are at different positions 884
    vec2.normalize(vFrom1to2, vFrom1to2);          885
    let vToC2 = [0, 0];                          886
    vec2.scale(vToC2, vFrom1to2, -c2.mRadius);    887
    vec2.add(vToC2, c2.getCenter(), vToC2);        888
    collisionInfo.setInfo(rSum - dist, vFrom1to2, vToC2); 889
}
... implementation of Step 3 to follow ...      890
4. The last case for two colliding circles is when both circle centers 892
   are located at exactly the same position (step 3). In this case, the 893
   collision normal is defined to be the negative y direction, and the 894
   collision depth is simply the larger of the two radii. 895

// Step 1: refer to previous step           896
if (dist !== 0) {                         897
    // Step 2: refer to previous step       898
} else {                                899
    let n = [0, -1];                      900
    // Step 3: Colliding circle centers are at exactly the same position 901
    if (c1.mRadius > c2.mRadius) {        902
        let pC1 = c1.getCenter();          903
        let ptOnC1 = [pC1[0], pC1[1] + c1.mRadius]; 904
        collisionInfo.setInfo(rSum, n, ptOnC1); 905
    }
}

```

```

906     } else {
907         let pC2 = c2.getCenter();
908         let ptOnC2 = [pC2[0], pC2[1]+ c2.mRadius];
909         collisionInfo.setInfo(rSum, n, ptOnC2);
910     }
911 }
```

Defining the Physics Component

You can now define the physics component to trigger the collision detection computations:

1. In the `src/engine/components` folder, create the `physics.js` file, import `CollisionInfo` and declare variables to support computations that are local to this file.
2. Define the `collideShape()` function to trigger the collision detection computation. Take note the two tests prior to the actual calling of shape `collisionTest()`. First, check to ensure the two shapes are not actually the same object. Second, call to the broad phase `boundTest()` method to determine the proximity of the shapes. Notice that the last parameter, `infoSet`, when defined will contain all `CollisionInfo` objects for all successful collisions. This is defined to support visualizing the `CollisionInfo` objects for verification and debugging purposes.

```

927 function collideShape(s1, s2, infoSet = null) {
928     let hasCollision = false;
929     if (s1 !== s2) {
930         if (s1.boundTest(s2)) {
931             hasCollision = s1.collisionTest(s2, mCInfo);
932             if (hasCollision) {
933                 // make sure mCInfo is always from s1 towards s2
934                 vec2.subtract(mS1toS2, s2.getCenter(), s1.getCenter());
935                 if (vec2.dot(mS1toS2, mCInfo.getNormal()) < 0)
936                     mCInfo.changeDir();
```

```

    // for showing off collision mCInfo!
937
    if (infoSet !== null) {
938        infoSet.push(mCInfo);
939        mCInfo = new CollisionInfo();
940    }
941
942    }
943
944    return hasCollision;
945
946}
947

```

3. Define utility functions to support the game developer:

processSet() to perform collision determination between all objects in the same GameObjectSet, processObjToSet() to check between a given GameObject and objects of a GameObjectSet, and processSetToSet() to check between all objects in two different GameObjectSets.

```

// collide all objects in the GameObjectSet with themselves
953
function processSet(set, infoSet = null) {
954
    let i = 0, j = 0;
955
    let hasCollision = false;
956
    for (i = 0; i < set.size(); i++) {
957        let s1 = set.getObjectAt(i).getRigidBody();
958        for (j = i + 1; j < set.size(); j++) {
959            let s2 = set.getObjectAt(j).getRigidBody();
960            hasCollision = collideShape(s1, s2, infoSet) || hasCollision;
961        }
962    }
963
    return hasCollision;
964
}
965
// collide a given GameObject with a GameObjectSet
966
function processObjToSet(obj, set, infoSet = null) {
967
    let j = 0;
968
}
969

```

CHAPTER 9 SIMULATING THE WORLD WITH RIGIDSHAPES

```
969     let hasCollision = false;
970     let s1 = obj.getRigidBody();
971     for (j = 0; j < set.size(); j++) {
972         let s2 = set.getObjectAt(j).getRigidBody();
973         hasCollision = collideShape(s1, s2, infoSet) || hasCollision;
974     }
975     return hasCollision;
976 }

977 // collide between all objects in two different GameObjectSets
978 function processSetToSet(set1, set2, infoSet = null){
979     let i = 0, j = 0;
980     let hasCollision = false;
981     for (i = 0; i < set1.size(); i++) {
982         let s1 = set1.getObjectAt(i).getRigidBody();
983         for (j = 0; j < set2.size(); j++) {
984             let s2 = set2.getObjectAt(j).getRigidBody();
985             hasCollision = collideShape(s1, s2, infoSet) || hasCollision;
986         }
987     }
988     return hasCollision;
989 }
```

990 4. Now, export all the defined functionality:

```
991 export {
992     // collide two shapes
993     collideShape,
994     // Collide
995     processSet, processObjToSet, processSetToSet
996 }
```

997 Lastly, remember to update the engine access file, `index.js`, to forward the newly
998 defined functionality to the client.

Modifying the MyGame to Test Circle Collisions

999

The modifications required for testing the newly defined collision functionality is rather straightforward:

1. Edit my_game_main.js; in the constructor, define the array for storing CollisionInfo and a new flag indicating if CollisionInfo should be drawn:

```
constructor() {  
    super();  
    ... identical to previous code ...  
    this.mCollisionInfos = [];  
    ... identical to previous code ...  
  
    // Draw controls  
    this.mDrawCollisionInfo = true; // showing of collision info  
    ... identical to previous code ...  
}  
1005  
1006  
1007  
1008  
1009  
1010  
1011  
1012  
1013
```

2. Modify the update() function to trigger the collision tests:

```
update() {  
    ... identical to previous code ...  
    if (this.mDrawCollisionInfo)  
        this.mCollisionInfos = [];  
    else  
        this.mCollisionInfos = null;  
    engine.physics.processObjToSet(this.mHero,  
        this.mPlatforms, this.mCollisionInfos);  
    engine.physics.processSetToSet(this.mAllObjs,  
        this.mPlatforms, this.mCollisionInfos);  
    engine.physics.processSet(this.mAllObjs, this.mCollisionInfos);  
}  
1015  
1016  
1017  
1018  
1019  
1020  
1021  
1022  
1023  
1024  
1025  
1026
```

1027 3. Modify the `draw()` function to draw the created `CollisionInfo`
1028 array when defined:

```
1029 draw() {  
1030     ... identical to previous code ...  
  
1031     if (this.mCollisionInfos !== null) {  
1032         for (let i = 0; i < this.mCollisionInfos.length; i++)  
1033             this.mCollisionInfos[i].draw(this.mCamera);  
1034         this.mCollisionInfos = [];  
1035     }  
1036     ... identical to previous code ...  
1037 }
```

1038 4. Remember to update the `drawControlUpdate()` function to
1039 support the C key for toggling of the drawing of the `CollisionInfo`
1040 objects:

```
1041 drawControlUpdate() {  
1042     let i;  
1043     if (engine.input.isKeyPressed(engine.input.keys.C)) {  
1044         this.mDrawCollisionInfo = !this.mDrawCollisionInfo;  
1045     }  
1046     ... identical to previous code ...  
1047 }
```

1048 Observations

1049 You can now run the project to examine your collision implementation between
1050 `RigidCircle` shapes in the form of the resulting `CollisionInfo` objects. Remember
1051 that you have only implemented circle-circle collisions. Now, use the left-/right-arrow
1052 keys to select and work with a `RigidCircle` object. Use the WASD keys to move this
1053 object around to observe the magenta line segment representing the collision normal
1054 and depth when it overlaps with another `RigidCircle`. Try typing the Y/U keys to verify
1055 the correctness of `CollisionInfo` for shapes with different radii. Now, type the G key to
1056 create a few more `RigidCircle` objects. Try moving the selected object and increase its

size such that it collides with multiple `RigidCircle` objects simultaneously and observe
that a proper `CollisionInfo` is computed for every collision. Finally, note that you can
toggle the drawing of `CollisionInfo` with the C key.

You have now implemented circle collision detection, built the required engine
infrastructure to support collisions, and verified the correctness of the system. You are
now ready to learn about Separating Axis Theorem (SAT) and implement a derived
algorithm to detect collisions between rectangles.

Separating Axis Theorem

The Separating Axis Theorem (SAT) is the foundation for one of the most popular
algorithms used for detecting collision between general convex shapes in 2D. Since the
derived algorithm can be computationally intensive, it is typically preceded with an
initial pass of the broad phase method. The SAT states that

*Two convex polygons are not colliding if there exists a line (or axis) that is
perpendicular to one of the given edges of the two polygons that when pro-
jecting all edges of the two polygons onto this axis results in no overlaps of
the projected edges.*

In other words, given two convex shapes in 2D space, iterate through all of the
edges of the convex shapes, one at a time. For each of the edges, derive a line (or axis)
that is perpendicular to the edge, project all the edges of the two convex shapes onto
this line, and compute for the overlaps of the projected edges. If you can find one of the
perpendicular lines where none of the projected edges overlaps, then the two convex
shapes do not collide.

Figure 9-7 illustrates this description using two axis-aligned rectangles. In this case,
there are two lines that are perpendicular to the edges of the two given shapes, the X and
Y axes.

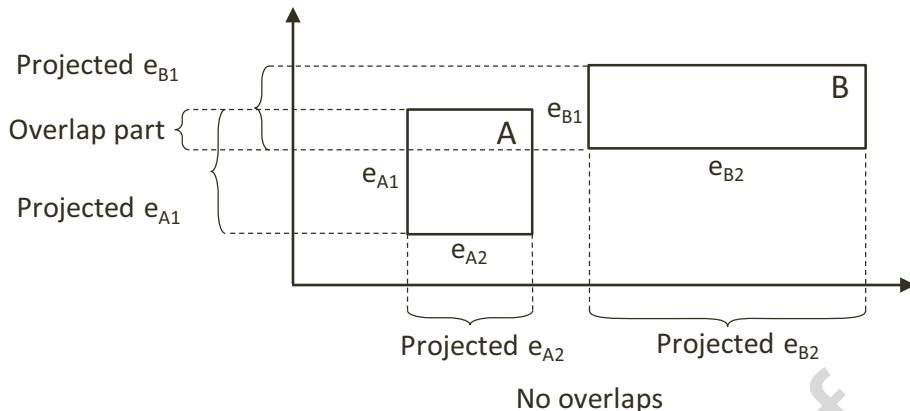


Figure 9-7. A line where projected edges do not overlap

When projecting all of the edges of the shapes onto these two lines/axes, note that the projection results on the Y axis overlap, while there is no overlap on the X axis. Since there exists one line that is perpendicular to one of the rectangle edges where the projected edges do not overlap, the SAT concludes that the two given rectangles do not collide.

The main strength of algorithms derived from the SAT is that for non-colliding shapes it has an early exit capability. As soon as an axis with no overlapping projected edge is detected, an algorithm can report no collision and does not need to continue with the testing for other axes. In the case of Figure 9-7, if the algorithm began with processing the X axis, there would be no need to perform the computation for the Y axis.

A Simple SAT-Based Algorithm

Algorithms derived based on the SAT typically consist of four steps. Note that this algorithm is applicable for detecting collisions between any convex shapes. For clarity, in the following explanation, each step is accompanied with a simple example consisting of two rectangles:

- **Step 1. Compute face normals:** Compute the perpendicular axes or face normals for projecting the edges. Using rectangles as an example, Figure 9-8 illustrates that there are four edges, and each edge has a corresponding perpendicular axis. For example, A1 is the corresponding axis for and thus is perpendicular to the edge e_{A1} . Note that in your `RigidRectangle` class, `mFaceNormal`, or face normals, are the perpendicular axes A1, A2, A3, and A4.

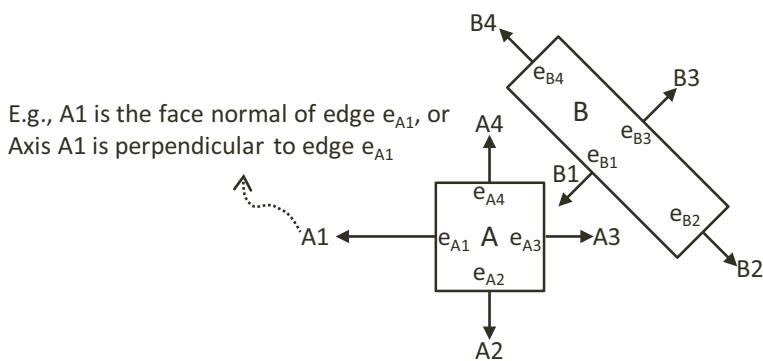


Figure 9-8. Rectangle edges and face normals

- **Step 2. Project vertices:** Project each of the vertices of the two convex shapes onto the face normals. For the given rectangle example, Figure 9-9 illustrates projecting all vertices onto the A1 axis from Figure 9-8.

1103
1104
1105
1106

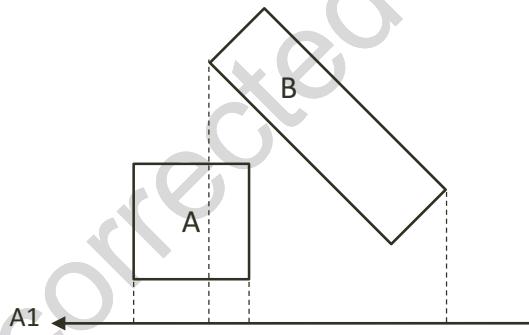


Figure 9-9. Project each vertex onto face normals (shows A1)

- **Step 3. Identify bounds:** Identifies the min and max bounds for the projected vertices of each convex shape. Continue with the rectangle example; Figure 9-10 shows the min and max positions for each of the two rectangles. Notice that the min/max positions are defined with respect to the direction of the given axis.

this figure will be printed in b/w

1107
1108
1109
1110
1111

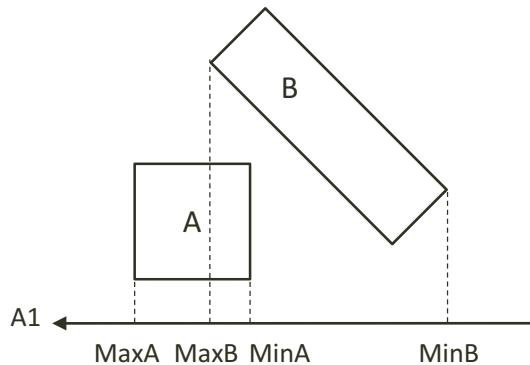


Figure 9-10. Identify the min and max bound positions for each rectangle

- **Step 4. Determine overlaps:** Determines if the two min/max bounds overlap. Figure 9-11 shows that the two projected bounds do indeed overlap. In this case, the algorithm cannot conclude and must proceed to process the next face normal. Notice that as illustrated in Figure 9-8, processing of face normal B1 or B3 will result in a deterministic conclusion of no collision.

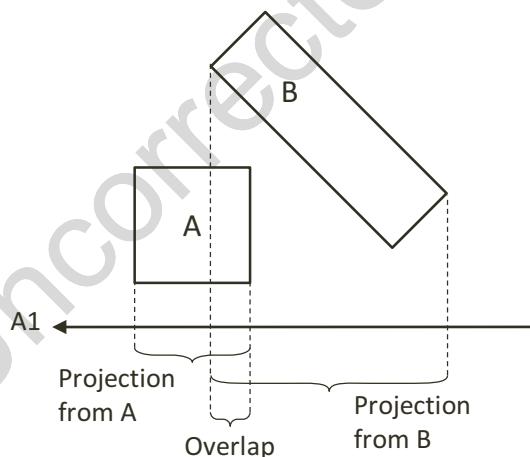


Figure 9-11. Test for overlaps of projected edges (shows A1)

- 1118 The given algorithm is capable of determining if a collision has occurred with no
 1119 additional information. Recall that after detecting a collision, the physics engine must
 1120 also resolve potential interpenetration and derive a response for the colliding shapes.
 1121 Both of these computations require additional information—the collision information as

introduced in Figure 9-4. The next section introduces an efficient SAT-based algorithm
1122 that computes support points to both inform the true/false outcome of the collision
1123 detection and serve as the basis for deriving collision information.
1124

An Efficient SAT Algorithm: The Support Points

As illustrated in Figure 9-12, a support point for a face normal of shape A is defined to be
1126 the vertex position on shape B where the vertex has the most negative distant from the
1127 corresponding edge of shape A. The vertex S_{A1} on shape B has the largest negative distant
1128 from edge e_{A1} when measured along the $A1$ direction, and thus, S_{A1} is the support point
1129 for face normal $A1$. The negative distance signifies that the measurement is directional
1130 and that a support point must be in the reversed direction from the face normal.
1131

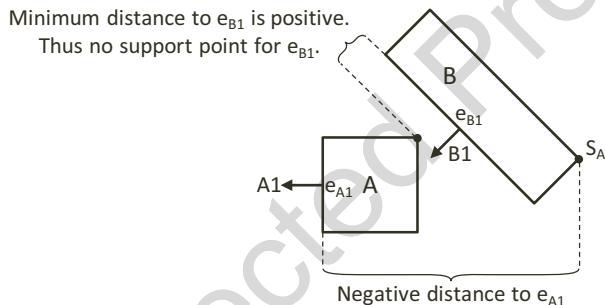


Figure 9-12. Support points of face normals

In general, the support point for a given face normal may be different during
1132 every update cycle and thus must be recomputed during each collision invocation. In
1133 addition, and very importantly, it is entirely possible for a face normal to not have a
1134 defined support point.
1135

Support Point May Not Exist for a Face Normal

A support point is defined only when the measured distance along the face normal has a
1137 negative value. For example, in Figure 9-12, the face normal $B1$ of shape B does not have
1138 a corresponding support point on shape A. This is because all vertices on shape A are
1139 positive distances away from the corresponding edge e_{B1} when measured along $B1$. The
1140 positive distances signify that all vertices of shape A are in front of the edge e_{B1} . In other
1141 words, the entire shape A is in front of the edge e_{B1} of shape B; and thus, the two shapes
1142 are not physically touching; and thus, they are not colliding.
1143

1144 It follows that, when computing the collision between two shapes, if any of the face
 1145 normals does not have a corresponding support point, then the two shapes are not
 1146 colliding. Once again, the early exit capability is an important advantage—the algorithm
 1147 can return a decision as soon as the first case of undefined support point is detected.

1148 For convenience of discussion and implementation, the distance between a support
 1149 point and the corresponding edge is referred to as the support point distance, and this
 1150 distance is computed as a positive number. In this way, the support point distance is
 1151 actually measured along the negative face normal direction. This will be the convention
 1152 followed in the rest of the discussions in this book.

1153 The Axis of Least Penetration and Collision Information

1154 When support points are defined for all face normals of a convex shape, the face normal
 1155 of the smallest support point distance is the axis leading to the least interpenetration.
 1156 Figure 9-13 shows the collision between two shapes where support points for all of the
 1157 face normals of shape B are defined: vertex S_{B1} on shape A is the corresponding support
 1158 point for face normal B1, S_{B2} for face normal B2, and so on. In this case, S_{B1} has the smallest
 1159 corresponding support point distance, and thus, the face normal B1 is the axis that leads
 1160 to the least interpenetration. The illustration on the right on Figure 9-13 shows that in this
 1161 case, support point distance is the collision depth, face normal B1 is collision normal,
 1162 support point S_{B1} is the start of the collision, and the end of the collision can be readily
 1163 computed; it is simply S_{B1} offset by collision depth in the collision normal direction.

this figure will be printed in b/w

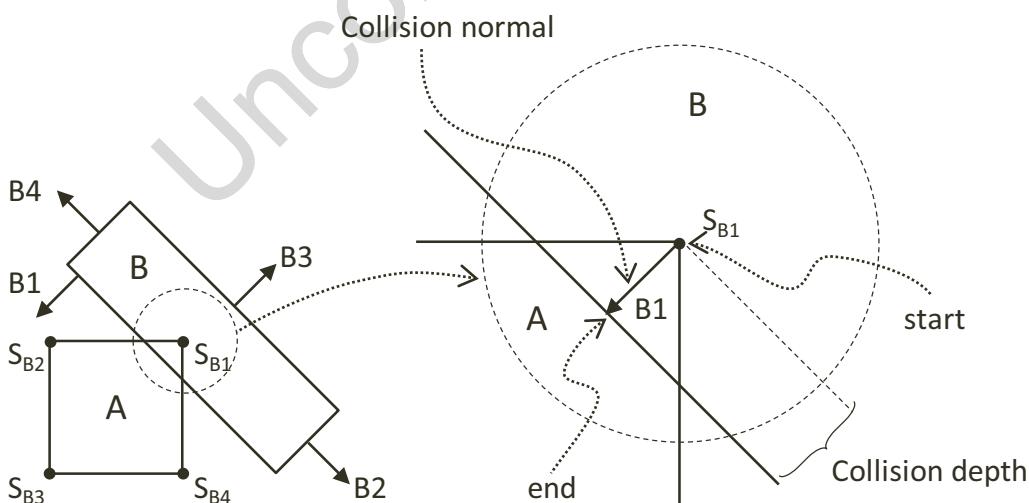


Figure 9-13. Axis of least penetration and the corresponding collision information

The Algorithm

1164

With the background description, the efficient SAT-based algorithm to compute the collision between two convex shapes, A and B, can be summarized as

1165

1166

Compute the support points for all the face normals on shape-A

1167

If any of the support points is not defined, there is no collision

1168

If all support points defined, compute the axis of least penetration

1169

Compute the support points for all the face normals on shape-B

1170

If any of the support points is not defined, there is no collision

1171

If all support points defined, compute the axis of least penetration

1172

The collision information is simply the smaller collision depth from the earlier two results. You are now ready to implement the support point SAT algorithm.

1173

1174

The Rectangle Collisions Project

1175

This project will guide you through the implementation of the support point SAT algorithm. You can see an example of this project running in Figure 9-14. The source code to this project is defined in chapter9/9.3.rectangle_collisions.

1176

1177

1178

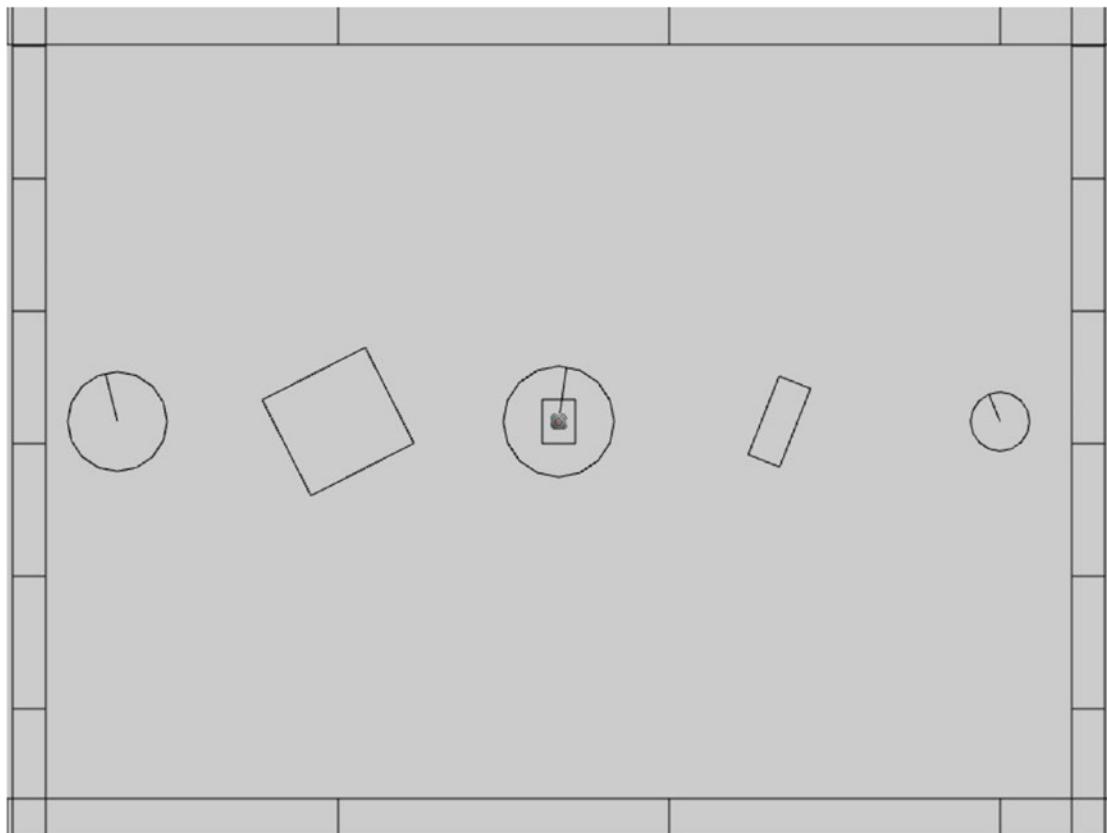


Figure 9-14. Running the Rectangle Collisions project

1179 The controls of the project are identical to the previous project:

1180 • **Behavior control:**

- **G key:** Randomly create a new rigid circle or rectangle

1182 • **Draw control:**

- **C key:** Toggles the drawing of all CollisionInfo

- **T key:** Toggles textures on all objects

- **R key:** Toggles the drawing of RigidShape

- **B key:** Toggles the drawing of the bound on each RigidShape

- **Object control:**
 - **Left-/right-arrow key:** Sequences through and selects an object. 1188
 - **WASD keys:** Move the selected object. 1189
 - **Z/X key:** Rotates the selected object. 1190
 - **Y/U key:** Increases/decreases RigidShape size of the selected object; this does not change the size of the corresponding Renderable object. 1191
1192
1193

The goals of the project are as follows:

- To gain insights into and implement the support point SAT algorithm 1195
- To continue with completing narrow phase collision detection 1196
implementation. 1197

After this project, your game engine will be able to collide between circle shapes and between rectangle shapes while still not supporting collisions between circle and rectangle shapes. This will be one step closer to completing the implementation of narrow phase collision detection for rigid shapes. The remaining functionality, detecting circle-rectangle collisions, will be covered in the next subsection.

Implementing the Support Point SAT

With the collision detection infrastructure from the previous project completed, the only modification required is to append the new functionality to the `RigidRectangle` class. Recall that the source code file `rigid_rectangle_collision.js` was created for the implementation of rectangle collision.

1. In the `src/engine/rigid_shapes` folder, edit `rigid_rectangle_collision.js` to define local variables. These are temporary storage during computations; they are statically allocated and reused to avoid the cost of repeated dynamic allocation during each invocation. 1208
1209
1210
1211
1212

```
class SupportStruct {
  constructor() {
    this.mSupportPoint = null;
```

CHAPTER 9 SIMULATING THE WORLD WITH RIGIDSHAPES

```
1216         this.mSupportPointDist = 0;  
1217     }  
1218 }  
1219 // temp work area to save memory allocations  
1220 let mTmpSupport = new SupportStruct();  
1221 let mCollisionInfoR1 = new CollisionInfo();  
1222 let mCollisionInfoR2 = new CollisionInfo();  
  
1223 2. Create a new function findSupportPoint() to compute a support  
1224 point based on, dir, the negated face normal direction, ptOnEdge,  
1225 a position on the given edge (e.g., a vertex). The listed code  
1226 marches through all the vertices; compute vToEdge, the vector  
1227 from vertices to ptOnEdge; project this vector onto the input dir;  
1228 and record the largest positive projected distant. Recall that dir is  
1229 the negated face normal direction, and thus, the largest positive  
1230 distant corresponds to the furthest vertex position. Note that it is  
1231 entirely possible for all of the projected distances to be negative. In  
1232 such cases, all vertices are in front of the input dir, a support point  
1233 does not exist for the given edge, and thus, the two rectangles do  
1234 not collide.  
  
1235 RigidRectangle.prototype.findSupportPoint = function (dir, ptOnEdge) {  
1236     // the longest project length  
1237     let vToEdge = [0, 0];  
1238     let projection;  
1239     mTmpSupport.mSupportPointDist = -Number.MAX_VALUE;  
1240     mTmpSupport.mSupportPoint = null;  
1241     // check each vector of other object  
1242     for (let i = 0; i < this.mVertex.length; i++) {  
1243         vec2.subtract(vToEdge, this.mVertex[i], ptOnEdge);  
1244         projection = vec2.dot(vToEdge, dir);  
1245         // find the longest distance with certain edge  
1246         // dir is -n direction, so the distance should be positive  
1247         if ((projection > 0) &&  
1248             (projection > mTmpSupport.mSupportPointDist)) {
```

```

        mTmpSupport.mSupportPoint = this.mVertex[i];
1249
        mTmpSupport.mSupportPointDist = projection;
1250
    }
1251
}
1252
}
1253

```

3. With the ability to locate a support point for any face normal,
 the next step is the find the axis of least penetration with the
`findAxisLeastPenetration()` function. Recall that the axis of
 least penetration is the support point with the least support point
 distant. The listed code loops over the four face normals, finds
 the corresponding support point and support point distance, and
 records the shortest distance. The while loop signifies that if a
 support point is not defined for any of the face normals, then the
 two rectangles do not collide.

```

RigidRectangle.prototype.findAxisLeastPenetration = function (otherRect,
1263
collisionInfo) {
1264
    let n;
1265
    let supportPoint;
1266
    let bestDistance = Number.MAX_VALUE;
1267
    let bestIndex = null;
1268
    let hasSupport = true;
1269
    let i = 0;
1270
    let dir = [0, 0];
1271
    while ((hasSupport) && (i < this.mFaceNormal.length)) {
1272
        // Retrieve a face normal from A
1273
        n = this.mFaceNormal[i];
1274
        // use -n as direction and the vertex on edge i as point on edge
1275
        vec2.scale(dir, n, -1);
1276
        let ptOnEdge = this.mVertex[i];
1277
        // find the support on B
1278
        // the point has longest distance with edge i
1279
        otherRect.findSupportPoint(dir, ptOnEdge);
1280
        hasSupport = (mTmpSupport.mSupportPoint !== null);
1281
    }
}

```

CHAPTER 9 SIMULATING THE WORLD WITH RIGIDSHAPES

```
1282     // get the shortest support point depth
1283     if ((hasSupport) && (mTmpSupport.mSupportPointDist < bestDistance)) {
1284         bestDistance = mTmpSupport.mSupportPointDist;
1285         bestIndex = i;
1286         supportPoint = mTmpSupport.mSupportPoint;
1287     }
1288     i = i + 1;
1289 }
1290 if (hasSupport) {
1291     // all four directions have support point
1292     let bestVec = [0, 0];
1293     vec2.scale(bestVec, this.mFaceNormal[bestIndex], bestDistance);
1294     let atPos = [0, 0];
1295     vec2.add(atPos, supportPoint, bestVec);
1296     collisionInfo.setInfo(bestDistance, this.mFaceNormal[bestIndex],
1297     atPos);
1298 }
1299 return hasSupport;
1300 }
```

- 1301 4. You can now implement the `collideRectRect()` function by
- 1302 computing the axis of least penetration with respect to each of the
- 1303 two rectangles and choosing the smaller of the two results:

```
1304 Rectangle.prototype.collideRectRect = function (r1, r2, collisionInfo) {
1305     var status1 = false;
1306     var status2 = false;
1307     // find Axis of Separation for both rectangle
1308     status1 = r1.findAxisLeastPenetration(r2, collisionInfoR1);
1309     if (status1) {
1310         status2 = r2.findAxisLeastPenetration(r1, collisionInfoR2);
1311         if (status2) {
1312             // if rectangles overlap, the shorter normal is the normal
1313             if (collisionInfoR1.getDepth() < collisionInfoR2.getDepth()) {
1314                 var depthVec = collisionInfoR1.getNormal().scale(
1315                     collisionInfoR1.getDepth());
```

```

    collisionInfo.setInfo(collisionInfoR1.getDepth(),           1316
                           collisionInfoR1.getNormal(),          1317
                           collisionInfoR1.mStart.subtract(depthVec)); 1318
} else {
    collisionInfo.setInfo(collisionInfoR2.getDepth(),           1320
                           collisionInfoR2.getNormal().scale(-1), 1321
                           collisionInfoR2.mStart);            1322
}
}
}
return status1 && status2;
}

```

5. Complete the implementation by modifying the `collisionTest()` function to call the newly defined `collideRectRect()` function to compute the collision between two rectangles:

```

RigidRectangle.prototype.collisionTest =           1331
function (otherShape, collisionInfo){             1332
    let status = false;                          1333
    if (otherShape.mType === "RigidCircle") {     1334
        status = false;                         1335
    } else {                                     1336
        status = this.collideRectRect(this, otherShape, collisionInfo); 1337
    }
    return status;                            1339
}

```

Observations

You can now run the project to test your implementation. You can use the left-/right-arrow keys to select any rigid shape and use the WASD keys to move the selected object. Once again, you can observe the magenta collision information between overlapping rectangles or overlapping circles. Remember that this line shows the least amount of positional correction needed to ensure that there is no overlap between the shapes. Type the Z/X keys to rotate and the Y/U keys to change the size of the selected object, and observe how the collision information changes accordingly.

At this point, only circle-circle and rectangle-rectangle collisions are supported, so when circles and rectangles overlap, there are no collision information shown. This will be resolved in the next project.

Collision Between Rectangles and Circles

The support point algorithm does not work with circles because a circle does not have identifiable vertex positions. Instead, you will implement an algorithm that detects collisions between a rectangle and a circle according to the relative position of the circle's center with respect to the rectangle.

Before discussing the actual algorithm, as illustrated in Figure 9-15, it is convenient to recognize that the area outside an edge of a rectangle can be categorized into three distinct regions by extending the connecting edges. In this case, the dotted lines separated the area outside the given edge into RG1, the region to the left/top; RG2, the region to the right/bottom; and RG3, the region immediately outside of the given edge.

With this background, the collision between a rectangle and a circle can be detected as follows:

- **Step A:** Compute the edge on the rectangle that is closest to the circle center.
- **Step B:** If the circle center is inside the rectangle, collision is detected.
- **Step C:** If circle center is outside
 - **Step C1:** If in Region RG1, distance between the circle center and top vertex determines if a collision has occurred.
 - **Step C2:** If in Region RG2, distance between the circle center and bottom vertex determines if a collision has occurred.
 - **Step C3:** If in Region RG3, perpendicular distance between the center and the edge determines if a collision has occurred.

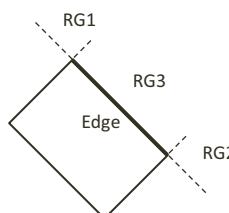


Figure 9-15. The three regions outside a given edge of a rectangle

The Rectangle and Circle Collisions Project

1374

This project guides you in implementing the described rectangle-circle collision detection algorithm. You can see an example of this project running in Figure 9-16. The source code to this project is defined in chapter9/9.4.rectangle_and_circle_collisions.

this figure will be printed in b/w

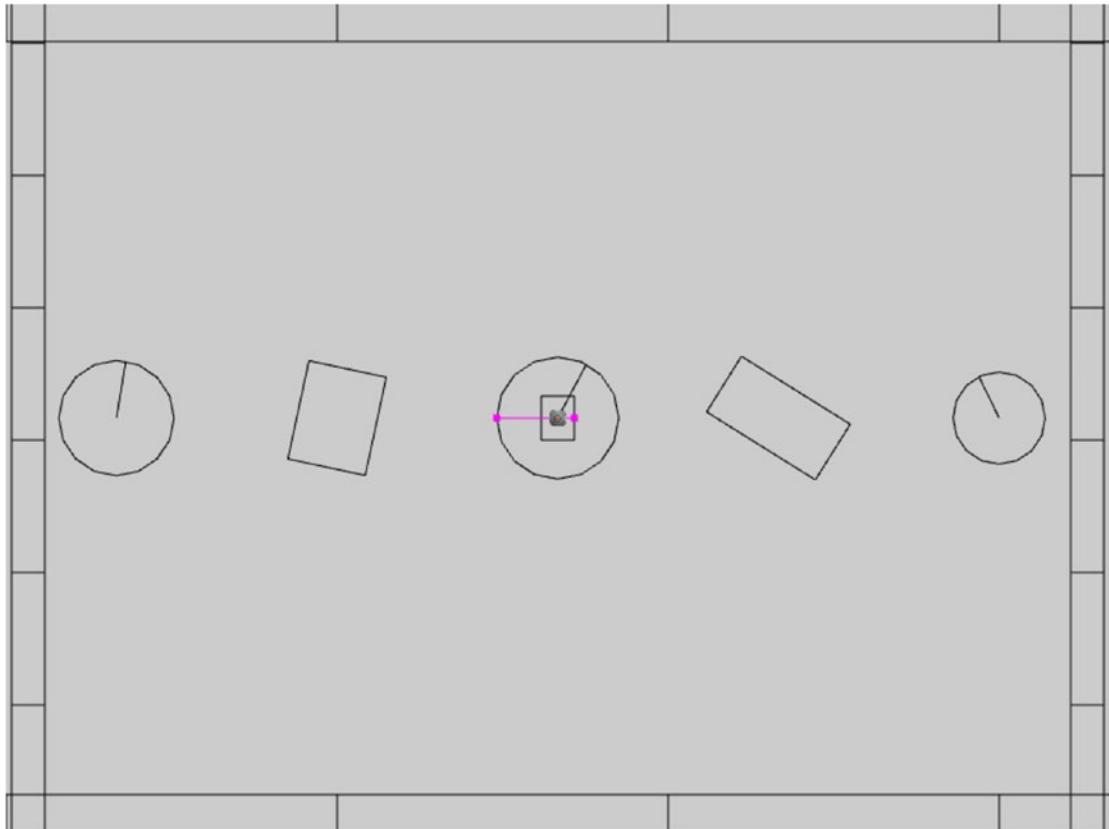


Figure 9-16. Running the Rectangle and Circle Collisions project

The controls of the project are identical to the previous project:

1379

- **Behavior control:**
 - **G key:** Randomly creates a new rigid circle or rectangle
- **Draw control:**
 - **C key:** Toggles the drawing of all CollisionInfo
 - **T key:** Toggles textures on all objects

1380

1381

1382

1383

1384

CHAPTER 9 SIMULATING THE WORLD WITH RIGIDSHAPES

- 1385 • **R key:** Toggles the drawing of RigidShape
- 1386 • **B key:** Toggles the drawing of the bound on each RigidShape
- 1387 • **Object control:**
 - 1388 • **Left-/right-arrow key:** Sequences through and selects an object.
 - 1389 • **WASD keys:** Move the selected object.
 - 1390 • **Z/X key:** Rotates the selected object.
 - 1391 • **Y/U key:** Increases/decreases RigidShape size of the selected
1392 object; this does not change the size of the corresponding
1393 Renderable object.

1394 The goals of the project are as follows:

- 1395 • To understand and implement the rectangle circle collision detection
1396 algorithm
- 1397 • To complete the narrow phase collision detection implementation for
1398 circle and rectangle shapes

1399 Defining Rectangle-Circle Collision

1400 Once again, with the completed collision detection infrastructure, the only modification
1401 required is to append the new functionality. This will be implemented in the
1402 RigidRectangle class. For readability of the rather involved algorithm, a new source
1403 code file, `rigid_rectangle_circle_collision.js`, will be created for implementation.

- 1404 1. Update the RigidRectangle access file to import from the latest
1405 source code file. In the `src/engine/rigid_shapes` folder, edit
1406 `rigid_rectangle.js` to replace the import to be from the latest
1407 source code file.

```
1408 import RigidRectangle from "./rigid_rectangle_circle_collision.js";  
1409 export default RigidRectangle;
```

- 1410 2. In the same folder, create the `rigid_rectangle_circle_`
1411 `collision.js` file to import from `rigid_rectangle_collision.`
1412 `js` such that new collision function can be appended to the class:

```
1413 import RigidRectangle from "./rigid_rectangle_collision.js";
```

3. Define a new function, `checkCircRectVertex()`, to process regions RG1 and RG2. As illustrated in the left diagram of Figure 9-17, the parameter $v1$ is the vector from vertex position to circle center. The right diagram of Figure 9-17 shows that a collision occurs when $dist$, the length of $v1$, is less than r , the radius. In this case, the collision depth is simply the difference between r and $dist$.

```
RigidRectangle.prototype.checkCircRectVertex = 1421
function(v1, cirCenter, r, info) { 1422
    // the center of circle is in corner region of mVertex[nearestEdge] 1423
    let dist = vec2.length(v1); 1424
    // compare the distance with radius to decide collision 1425
    if (dist > r) 1426
        return false; 1427
    let radiusVec = [0, 0]; 1428
    let ptAtCirc = [0, 0]; 1429
    vec2.scale(v1, v1, 1/dist); // normalize 1430
    vec2.scale(radiusVec, v1, -r); 1431
    vec2.add(ptAtCirc, cirCenter, radiusVec); 1432
    info.setInfo(r - dist, v1, ptAtCirc); 1433
    return true; 1434
}
```

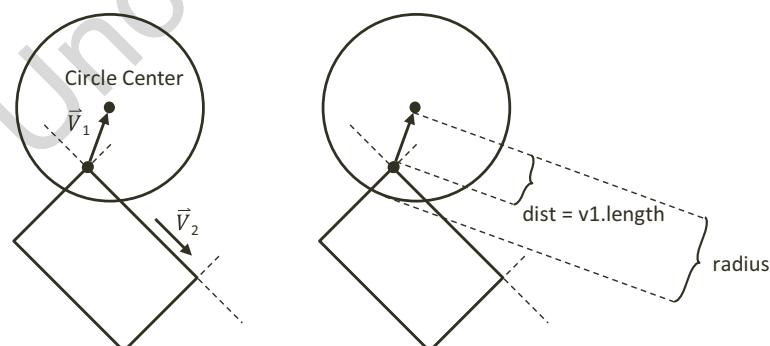


Figure 9-17. Left: condition when center is in region RG1. Right: the corresponding collision information

CHAPTER 9 SIMULATING THE WORLD WITH RIGIDSHAPES

- 1436 4. Define `collideRectCirc()` function to detect the collision
1437 between a rectangle and a circle. The following code shows the
1438 declaration of local variables and the five major steps, A to C3, that
1439 must be performed. The details of each steps are discussed in the
1440 rest of this subsection.

```
1441 RigidRectangle.prototype.collideRectCirc =  
1442 function (otherCir, collisionInfo) {  
1443     let outside = false;  
1444     let bestDistance = -Number.MAX_VALUE;  
1445     let nearestEdge = 0;  
1446     let vToC = [0, 0];  
1447     let projection = 0;  
1448     let i = 0;  
1449     let cirCenter = otherCir.getCenter();  
1450     ... Step A: Compute nearest edge, handle if center is inside ...  
1451     if (!outside) {  
1452         ... Step B: Circle center is inside rectangle ...  
1453         return;  
1454     }  
1455     ... Steps C1 to C3: Circle center is outside rectangle ...  
1456     return true;  
1457 };
```

- 1458 5. Step A, compute the nearest edge. The nearest edge can be found
1459 by computing the perpendicular distances between the circle
1460 center and each edge of the rectangle. This distance is simply
1461 the projection of the vector, from each vertex to the circle center,
1462 onto the corresponding face normal. The listed code iterates
1463 through all of the vertices computing the vector from the vertex
1464 to the circle center and projects the computed vector to the
1465 corresponding face normal.

```

// Step A: Compute the nearest edge 1466
while ((!outside) && (i<4)) { 1467
    // find the nearest face for center of circle 1468
    vec2.subtract(vToC, cirCenter, this.mVertex[i]); 1469
    projection = vec2.dot(vToC, this.mFaceNormal[i]); 1470
    if (projection > bestDistance) { 1471
        outside = (projection > 0); // if projection < 0, inside 1472
        bestDistance = projection; 1473
        nearestEdge = i; 1474
    } 1475
    i++; 1476
}
1477

```

As illustrated in the left diagram of Figure 9-18, when the circle center is inside the rectangle, all vertex to center vectors will be in the opposite directions of their corresponding face normals and thus will result in negative projected lengths. This is in contrast to the right diagram of Figure 9-18, when the center is outside of the rectangle. In this case, at least one of the projected lengths will be positive. For this reason, the “nearest projected distance” is the one with the least negative value and thus is actually the largest number.

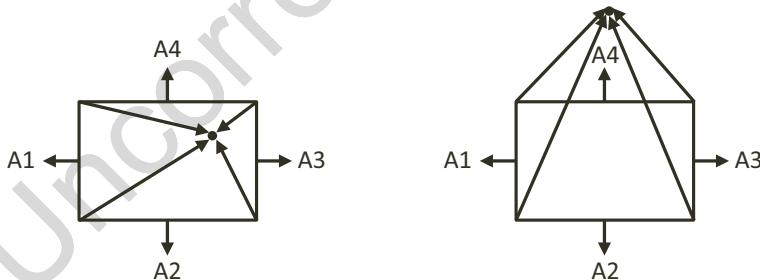


Figure 9-18. Left: center inside the rectangle will result in all negative projected length. Right: center outside the rectangle will result in at least one positive projected length

6. Step B, if the circle center is inside the rectangle, then collision is detected and the corresponding collision information can be computed and returned:

this figure will be printed in b/w

CHAPTER 9 SIMULATING THE WORLD WITH RIGIDSHAPES

```

1488 if (!outside) { // inside
1489     // Step B: The center of circle is inside of rectangle
1490     vec2.scale(radiusVec, this.mFaceNormal[nearestEdge], otherCir.mRadius);
1491     dist = otherCir.mRadius - bestDistance; // bestDist is -ve
1492     vec2.subtract(ptAtCirc, cirCenter, radiusVec);
1493     collisionInfo.setInfo(dist, this.mFaceNormal[nearestEdge], ptAtCirc);
1494     return true;
1495 }

```

- 1496 7. Step C1, determine and process if the circle center is in Region
 1497 RG1. As illustrated in the left diagram of Figure 9-17, Region RG1
 1498 can be detected when v1, the vector between the center and
 1499 vertex, is in the opposite direction of v2, the direction of the edge.
 1500 This condition is computed in the following listed code:

```

1501 let v1 = [0, 0], v2 = [0, 0];
1502 vec2.subtract(v1, cirCenter, this.mVertex[nearestEdge]);
1503 vec2.subtract(v2, this.mVertex[(nearestEdge + 1) % 4],
1504                 this.mVertex[nearestEdge]);
1505 let dot = vec2.dot(v1, v2);
1506 if (dot < 0) {
1507     // Step C1: In Region RG1
1508     return this.checkCircRectVertex(v1, cirCenter,
1509                                     otherCir.mRadius, collisionInfo);
1510 } else {
1511     ... implementation of Steps C2 and C3 to follow ...
1512 }

```

- 1513 8. Steps C2 and C3, differentiate and process for Regions RG2 and
 1514 RG3. The listed code performs complementary computation
 1515 for the other vertex on the same rectangle edge for Region RG2.
 1516 The last region for the circle center to be located in would be
 1517 the area immediately outside the nearest edge. In this case, the
 1518 bestDistance computed previously in step A is the distance
 1519 between the circle center and the given edge. If this distance is
 1520 less than the circle radius, then a collision has occurred.

```

if (dot < 0) {                                1521
    // Step C1: In Region RG1                  1522
    ... identical to previous code ...        1523
} else {                                         1524
    // Either in Region RG2 or RG3            1525
    // v1 is from right vertex of face to center of circle 1526
    // v2 is from right vertex of face to left vertex of face 1527
    vec2.subtract(v1, cirCenter, this.mVertex[(nearestEdge + 1) % 4]); 1528
    vec2.scale(v2, v2, -1);                   1529
    dot = vec2.dot(v1, v2);                  1530
    if (dot < 0) {                           1531
        // Step C2: In Region RG2            1532
        return this.checkCircRectVertex(v1, cirCenter,          1533
                                         otherCir.mRadius, collisionInfo); 1534
    } else {                                     1535
        // Step C3: In Region RG3            1536
        if (bestDistance < otherCir.mRadius) {
            vec2.scale(radiusVec,           1537
                          this.mFaceNormal[nearestEdge], otherCir.mRadius); 1538
            dist = otherCir.mRadius - bestDistance; 1539
            vec2.subtract(ptAtCirc, cirCenter, radiusVec); 1540
            collisionInfo.setInfo(dist,          1541
                                      this.mFaceNormal[nearestEdge], ptAtCirc); 1542
            return true;                      1543
        } else {                           1544
            return false;                 1545
        }
    }
}

```

Calling the Newly Defined Function

The last step is to invoke the newly defined function. Note that the collision function should be called when a circle comes into contact with a rectangle, as well as when a rectangle comes into contact with a circle. For this reason, you must modify both the `RigidRectangle` class in `rigid_rectangle_collision.js` and the `RigidCircle` class in `rigid_circle_collision.js`.

- 1556 1. In the `src/engine/rigid_shapes` folder, edit `rigid_rectangle_`
 1557 `collision.js`, and modify the `collisionTest()` function to call
 1558 the newly defined `collideRectCirc()` when the parameter is a
 1559 circle shape:

```
1560 RigidRectangle.prototype.collisionTest =
1561   function (otherShape, collisionInfo) {
1562     let status = false;
1563     if (otherShape.mType === "RigidCircle") {
1564       status = this.collideRectCirc(otherShape, collisionInfo);
1565     } else {
1566       status = this.collideRectRect(this, otherShape, collisionInfo);
1567     }
1568     return status;
1569 }
```

- 1570 2. In the same folder, edit `rigid_circle_collision.js`, modify
 1571 the `collisionTest()` function to call the newly defined
 1572 `collideRectCirc()` when the parameter is a rectangle shape:

```
1573 RigidCircle.prototype.collisionTest =
1574   function (otherShape, collisionInfo) {
1575     let status = false;
1576     if (otherShape.mType === "RigidCircle") {
1577       status = this.collideCircCirc(this, otherShape, collisionInfo);
1578     } else {
1579       status = otherShape.collideRectCirc(this, collisionInfo);
1580     }
1581     return status;
1582 }
```

1583 Observations

1584 You can now run the project to test your implementation. You can create new rectangles
 1585 and circles, move, and rotate them to observe the corresponding collision information.
 1586 You have finally completed the narrow phase collision detection implementation
 1587 and can begin to examine the motions of these rigid shapes.

Movement

1588

Movement is the description of how object positions change in the simulated world. Mathematically, movement can be formulated in many ways. In Chapter 6, you experienced working with movement where you continuously accumulated a displacement to the position of an object. As illustrated in the following equation and in Figure 9-19, you have been working with describing movement based on constant displacements.

1589

1590

1591

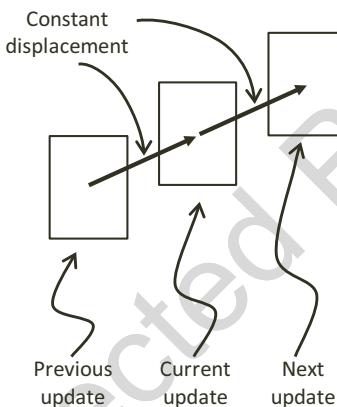
1592

1593

1594

- $p_{\text{new}} = p_{\text{current}} + \text{displacement}$

1595



this figure will be printed in b/w

Figure 9-19. Movement based on constant displacements

Movement that is governed by the constant displacement formulation becomes restrictive when it is necessary to change the amount to be displaced over time. Newtonian mechanics address this restriction by considering time in the movement formulations, as seen in the following equations:

1596

1597

1598

1599

- $v_{\text{new}} = v_{\text{current}} + \int a(t)dt$

1600

- $p_{\text{new}} = p_{\text{current}} + \int v(t)dt$

1601

These two equations represent Newtonian-based movements where $v(t)$ is the velocity that describes the change in position over time and $a(t)$ is the acceleration that describes the change in velocity over time.

1602

1603

1604

Notice that both velocity and acceleration are vector quantities encoding both the magnitude and direction. The magnitude of a velocity vector defines the speed, and the normalized velocity vector identifies the direction that the object is traveling. An acceleration vector lets you know whether an object is speeding up or slowing down as well as the changes in the object's traveling directions. Acceleration is changed by the forces acting upon an object. For example, if you were to throw a ball into the air, the gravitational force would affect the object's acceleration over time, which in turn would change the object's velocity.

Explicit Euler Integration

The Euler method, or Explicit Euler Integration, approximates integrals based on initial values. This is one of the most straightforward approximations for integrals. As illustrated in the following two equations, in the case of the Newtonian movement formulation, the new velocity, v_{new} , of an object can be approximated as the current velocity, v_{current} plus the current acceleration, a_{current} , multiplied by the elapsed time. Similarly, the object's new position, p_{new} , can be approximated by the object's current position, p_{current} , plus the current velocity, v_{current} , multiplied by the elapsed time.

- $v_{\text{new}} = v_{\text{current}} + a_{\text{current}} * dt$

- $p_{\text{new}} = p_{\text{current}} + v_{\text{current}} * dt$

The left diagram of Figure 9-20 illustrates a simple example of approximating movements with Explicit Euler Integration. Notice that the new position, p_{new} , is computed based on the current velocity, v_{current} , while the new velocity, v_{new} , is computed to move the position for the next update cycle.

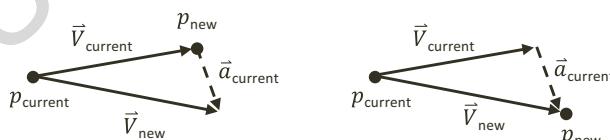


Figure 9-20. Explicit (left) and Symplectic (right) Euler Integration

Symplectic Euler Integration

1627

You will implement the Semi-implicit Euler Integration or Symplectic Euler Integration. 1628
 With Symplectic Euler Integration, instead of current results, intermediate results are 1629
 used in subsequent approximations and thus better simulate the actual movement. The 1630
 following equations show Symplectic Euler Integration. Notice that it is nearly identical 1631
 to the Euler Method except that the new velocity, v_{new} , is being used when calculating 1632
 the new position, p_{new} . This essentially means that the velocity for the next frame is being 1633
 used to calculate the position of this frame. 1634

- $v_{\text{new}} = v_{\text{current}} + a_{\text{current}} * dt$ 1635

- $p_{\text{new}} = p_{\text{current}} + v_{\text{new}} * dt$ 1636

The right diagram of Figure 9-20 illustrates that with the Symplectic Euler Integration, 1637
 the new position p_{new} is computed based on the newly computed velocity, v_{new} . 1638

The Rigid Shape Movements Project

1639

You are now ready to implement Symplectic Euler Integration to approximate 1640
 movements. The fixed time step, dt , formulation conveniently allows the integral to 1641
 be evaluated once per update cycle. This project will guide you through working with 1642
 the RigidShape class to support movement approximation with the Symplectic Euler 1643
 Integration. You can see an example of this project running in Figure 9-21. The source 1644
 code to this project is defined in chapter9/9.5.rigid_shape_movements. 1645

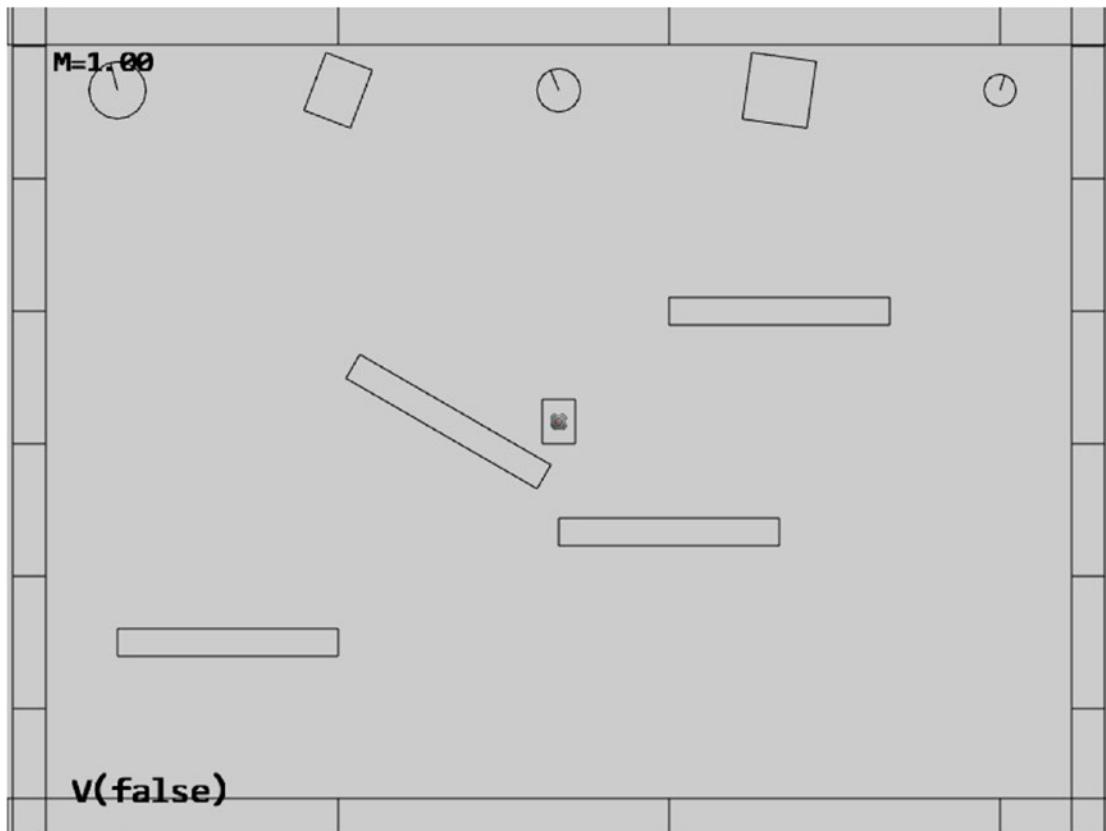


Figure 9-21. Running the Rigid Shape Movements project

1646 The controls of the project are the same as previous with additional commands to
 1647 control the behaviors and the mass of selected object:

- 1648 • **Behavior control:**
 - 1649 • **V key:** Toggles motion of all objects
 - 1650 • **H key:** Injects random velocity to all objects
 - 1651 • **G key:** Randomly creates a new rigid circle or rectangle
- 1652 • **Draw control:**
 - 1653 • **C key:** Toggles the drawing of all CollisionInfo
 - 1654 • **T key:** Toggles textures on all objects
 - 1655 • **R key:** Toggles the drawing of RigidBody
 - 1656 • **B key:** Toggles the drawing of the bound on each RigidBody

- **Object control:**
 - **Left-/right-arrow key:** Sequences through and selects an object. 1657
 - **WASD keys:** Move the selected object. 1658
 - **Z/X key:** Rotates the selected object. 1659
 - **Y/U key:** Increases/decreases RigidShape size of the selected object; this does not change the size of the corresponding Renderable object. 1660
 - **Up-/down-arrow key + M:** Increase/decrease the mass of the selected object. 1661

The goals of the project are as follows:

- To complete the implementation of RigidShape classes to include relevant physical attributes 1666
- To implement movement approximation based on Symplectic Euler Integration 1667

In addition to implementing Symplectic Euler Integration, this project also guides you to define attributes required for collision simulation and response, such as mass, inertia, friction, etc. As will be explained, each of these attributes will play a part in the simulation of object collision responses. This straightforward information is presented here to avoid distracting discussion of the more complex concepts to be covered in the subsequent projects.

In the rest of this section, you will first define relevant physical attributes to complete the RigidShape implementation. After that, you will focus on building Symplectic Euler Integration support for approximating movements.

Completing the RigidShape Implementation

As mentioned, in order to allow focused discussions of the more complex concepts in the later sections, the attributes for supporting collisions and the corresponding supporting functions are introduced in this project. These attributes are defined in the rigid shape classes.

1685 Modifying the RigidShape Class

1686 Edit `rigid_shape.js` in the `src/engine/rigid_shape` folder:

- 1687 1. In the constructor of the `RigidShape` class, define variables
1688 representing acceleration, velocity, angular velocity, mass,
1689 rotational inertia, restitution (bounciness), and friction.
1690 Notice that the inverse of the mass value is actually stored for
1691 computation efficiency (by avoiding an extra division during each
1692 update calculation). Additionally, notice that a mass of zero is
1693 used to represent a stationary object.

```
1694 class RigidShape {
1695     constructor(xf) {
1696         this.mXform = xf;
1697         this.mAcceleration = physics.getSystemAcceleration();
1698         this.mVelocity = vec2.fromValues(0, 0);
1699         this.mType = "";
1700         this.mInvMass = 1;
1701         this.mInertia = 0;
1702         this.mFriction = 0.8;
1703         this.mRestitution = 0.2;
1704         this.mAngularVelocity = 0;
1705         this.mBoundRadius = 0;
1706         this.mDrawBounds = false;
1707     }
```

- 1708 2. Define the `setMass()` function to set the mass of the object.
1709 Once again, for computational efficiency, the inverse of the
1710 mass is stored. Setting the mass of an object to zero or negative
1711 is a signal that the object is stationary with zero acceleration
1712 and will not participate in any movement computation.
1713 Notice that when the mass of an object is changed, you would

need to call `updateInertia()` to update its rotational inertia,
`mInertial`. Rotational inertia is geometric shape specific, and the
implementation of `updateInertia()` is subclass specific.

```
setMass(m) { 1717
    if (m > 0) { 1718
        this.mInvMass = 1 / m;
        this.mAcceleration = physics.getSystemAcceleration();
    } else { 1721
        this.mInvMass = 0;
        this.mAcceleration = [0, 0]; // to ensure object does not move
    }
    this.updateInertia(); 1725
}
} 1726
```

3. Define getter and setter functions for all of the other
corresponding variables. These functions are straightforward and
are not listed here. 1727
4. For the convenience of debugging, define a function
`getCurrentState()` to retrieve variable values as text and a
function `userSetsState()` to allow interactive manipulations of
the variables: 1730

```
getCurrentState() { 1734
    let m = this.mInvMass;
    if (m !== 0)
        m = 1 / m;

    return "M=" + m.toFixed(kPrintPrecision) +
        "(I=" + this.mInertia.toFixed(kPrintPrecision) + ")" +
        " F=" + this.mFriction.toFixed(kPrintPrecision) +
        " R=" + this.mRestitution.toFixed(kPrintPrecision);
}
} 1742

userSetsState() {
    // keyboard control
    let delta = 0;
}
} 1745
```

CHAPTER 9 SIMULATING THE WORLD WITH RIGIDSHAPES

```
1746     if (input.isKeyPressed(input.keys.Up)) {
1747         delta = kRigidShapeUIDelta;
1748     }
1749     if (input.isKeyPressed(input.keys.Down)) {
1750         delta = -kRigidShapeUIDelta;
1751     }
1752     if (delta !== 0) {
1753         if (input.isKeyPressed(input.keys.M)) {
1754             let m = 0;
1755             if (this.mInvMass > 0)
1756                 m = 1 / this.mInvMass;
1757             this.setMass(m + delta * 10);
1758         }
1759         if (input.isKeyPressed(input.keys.F)) {
1760             this.mFriction += delta;
1761             if (this.mFriction < 0)
1762                 this.mFriction = 0;
1763             if (this.mFriction > 1)
1764                 this.mFriction = 1;
1765         }
1766         if (input.isKeyPressed(input.keys.R)) {
1767             this.mRestitution += delta;
1768             if (this.mRestitution < 0)
1769                 this.mRestitution = 0;
1770             if (this.mRestitution > 1)
1771                 this.mRestitution = 1;
1772         }
1773     }
1774 }
```

1775 **Modifying the RigidCircle Class**

1776 As mentioned, the rotational inertia, `mInertial`, is specific to geometric shape and must
1777 be modified by the corresponding classes:

1. Edit `rigid_circle_main.js` in the `src/engine/rigid_shapes` folder to modify the `RigidCircle` class to define the `updateInertia()` function. This function calculates the rotational inertia of a circle when its mass has changed.

```

updateInertia() {
    if (this.mInvMass === 0) {
        this.mInertia = 0;
    } else {
        // this.mInvMass is inverted!!
        // Inertia=mass * radius^2
        this.mInertia = (1 / this.mInvMass) *
            (this.mRadius * this.mRadius) / 12;
    }
};
```

2. Update the `RigidCircle` constructor and `incShapeSize()` function to call the `updateInertia()` function:

```

constructor(xf, radius) {
    super(xf);
    ... identical to previous code ...
    this.updateInertia();
}

incShapeSizeBy(dt) {
    ... identical to previous code ...
    this.updateInertia();
}
```

Modifying the RigidRectangle Class

Modifications similar to the `RigidCircle` class must be defined for the `RigidRectangle` class:

1. Edit `rigid_rectangle_main.js` in the `src/engine/rigid_shapes` folder to define the `updateInertia()` function:

CHAPTER 9 SIMULATING THE WORLD WITH RIGIDSHAPES

```
1808 updateInertia() {  
1809     // Expect this.mInvMass to be already inverted!  
1810     if (this.mInvMass === 0)  
1811         this.mInertia = 0;  
1812     else {  
1813         // inertia=mass*width^2+height^2  
1814         this.mInertia = (1 / this.mInvMass) *  
1815             (this.mWidth * this.mWidth +  
1816              this.mHeight * this.mHeight) / 12;  
1817         this.mInertia = 1 / this.mInertia;  
1818     }  
1819 }  
  
1820     2. Similar to the RigidCircle class, update the constructor and  
1821     incShapeSize() function to call the updateInertia() function:
```

```
1822 constructor(xf, width, height) {  
1823     super(xf);  
1824     ... identical to previous code ...  
1825     this.updateInertia();  
1826 }  
  
1827 incShapeSizeBy(dt) {  
1828     ... identical to previous code ...  
1829     this.updateInertia();  
1830 }
```

1831 Defining System Acceleration and Motion Control

1832 With the RigidBody implementation completed, you are now ready to define the
1833 support for movement approximation.

1834 Define a system-wide acceleration and motion control by adding appropriate
1835 variables and access functions to physics.js in the src/engine/components folder.
1836 Remember to export the newly defined functionality.

```
1837 let mSystemAcceleration = [0, -20];    // system-wide default acceleration  
1838 let mHasMotion = true;
```

```

// getters and setters 1839
function getSystemAcceleration() { 1840
    return vec2.clone(mSystemAcceleration); 1841
}
1842
function setSystemAcceleration(x, y) { 1843
    mSystemAcceleration[0] = x; 1844
    mSystemAcceleration[1] = y; 1845
}
1846
function getHasMotion() { return mHasMotion; } 1847
function toggleHasMotion() { mHasMotion = !mHasMotion; } 1848
... identical to previous code ... 1849
1850
export { 1851
    // Physics system attributes
    getSystemAcceleration, setSystemAcceleration, 1852
    getHasMotion, toggleHasMotion, 1853
    ... identical to previous code ...
}
1854
1855

```

Accessing the Fixed Time Interval

In your game engine, the fixed time step, dt, is simply the time interval in between the calls to the `loopOnce()` function in the game loop component. Now, edit `loop.js` in the `src/engine/core` folder to define and export the update time interval.

```

const kUPS = 60; // Updates per second 1860
const kMPF = 1000 / kUPS; // Milliseconds per update. 1861
const kSPU = 1/kUPS; // seconds per update 1862
... identical to previous code ... 1863
1864
function getUpdateIntervalInSeconds() { return kSPU; }
... identical to previous code ... 1865
1866
export {getUpdateIntervalInSeconds}

```

1867 **Implementing Symplectic Euler Integration in the RigidShape 1868 class**

1869 You can now implement the Symplectic Euler Integration movement approximation in
1870 the rigid shape classes. Since this movement behavior is common to all types of rigid
1871 shapes, the implementation should be located in the base class, `RigidShape`.

- 1872 1. In the `src/engine/rigid_shapes` folder, edit `rigid_shape.js`
1873 to define the `travel()` function to implement Symplectic Euler
1874 Integration for movement. Notice how the implementation closely
1875 follows the listed equations where the updated velocity is used for
1876 computing the new position. Additionally, notice the similarity
1877 between linear and angular motion where the location (either a
1878 position or an angle) is updated by a displacement that is derived
1879 from the velocity and time step. Rotation will be examined in
1880 detail in the last section of this chapter.

```
1881 travel() {
1882     let dt = loop.getUpdateIntervalInSeconds();
1883     // update velocity by acceleration
1884     vec2.scaleAndAdd(this.mVelocity,
1885                         this.mVelocity, this.mAcceleration, dt);
1886     // p = p + v*dt with new velocity
1887     let p = this.mXform.getPosition();
1888     vec2.scaleAndAdd(p, p, this.mVelocity, dt);
1889     this.mXform.incRotationByRad(this.mAngularVelocity * dt);
1890 }
```

- 1891 2. Modify the `update()` function to invoke `travel()` when the object
1892 is not stationary, `mInvMass` of 0, and when motion of the physics
1893 component is switched on:

```
1894 update() {
1895     if (this.mInvMass === 0)
1896         return;
```

```

if (physics.getHasMotion())
    this.travel();
}

```

1897
1898
1899

Modifying MyGame to Test Movements

1900

The modification to the MyGame class involves supporting new user commands for toggling system-wide motion, injecting random velocity, and setting the scene stationary boundary objects to rigid shapes with zero mass. The injecting of random velocity is implemented by the randomizeVelocity() function defined in my_game_bounds.js file.

1901
1902
1903
1904

All updates to the MyGame class are straightforward. To avoid unnecessary distraction, the details are not shown. As always, you can refer to the source code files in the src/my_game folder for implementation details.

1905
1906
1907

Observations

1908

You can now run the project to test your implementation. In order to properly observe and track movements of objects, initially motion is switched off. You can type the V key to enable motion when you are ready. When motion is toggled on, you can observe a natural-looking free-falling movement for all objects. You can type G to create more objects and observe similar free-fall movements of the created objects.

1909
1910
1911
1912
1913

Notice that when the objects fall below the lower platform, they are regenerated in the central region of the scene with a random initial upward velocity. Observe the objects move upward until the y component of the velocity reaches zero, and then they begin to fall downward as a result of gravitational acceleration. Typing the H key injects new random upward velocities to all objects resulting in objects decelerating while moving upward.

1914
1915
1916
1917
1918

Try typing the C key to observe the computed collision information when objects overlap or interpenetrate. Pay attention and note that interpenetration occurs frequently as objects travel through the scene. You are now ready to examine and implement how to resolve object interpenetration in the next section.

1919
1920
1921
1922

Interpenetration of Colliding Objects

1923

The fixed update time step introduced in the previous project means that the actual location of an object in a continuous motion is approximated by a discrete set of positions. As illustrated in Figure 9-22, the movement of the rectangular object is

1924
1925
1926

1927 approximated by placing the object at the three distinct positions over three update
 1928 cycles. The most notable ramification of this approximation is in the challenges when
 1929 determining collisions between objects.

this figure will be printed in b/w

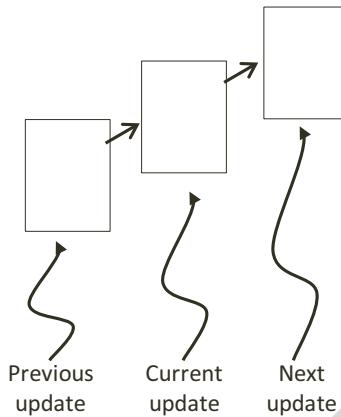


Figure 9-22. A rigid square in continuous motion

1930 You can see one such challenge in Figure 9-22. Imagine a thin wall existed in the
 1931 space between the current and the next update. You would expect the object to collide
 1932 and stop by the wall in the next update. However, if the wall was sufficiently thin, the
 1933 object would appear to pass right through the wall as it jumped from one position to
 1934 the next. This is a common problem faced in many game engines. A general solution for
 1935 these types of problems can be algorithmically complex and computationally intensive.
 1936 It is typically the job of the game designer to mitigate and avoid this problem with well-
 1937 designed (e.g., appropriate size) and well-behaved (e.g., appropriate traveling speed)
 1938 game objects.

1939 Figure 9-23 shows another, and more significant, collision-related challenge
 1940 resulting from fixed update time steps. In this case, before the time step, the objects
 1941 are not touching. After the time step, the results of the movement approximation place
 1942 the two objects where they partly overlap. In the real world, if the two objects are rigid
 1943 shapes or solids, then the overlap, or interpenetration, would never occur. For this
 1944 reason, this situation must be properly resolved in a rigid shape physics simulation. This
 1945 is where details of a collision must be computed such that interpenetrating situations
 1946 like these can be properly resolved.

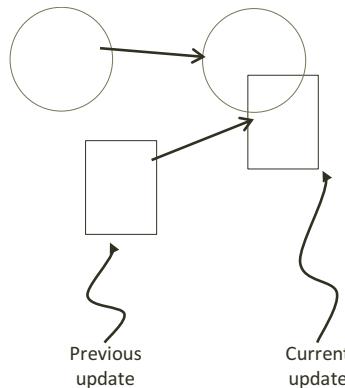


Figure 9-23. The interpenetration of colliding objects

Collision Position Correction

In the context of game engines, collision resolution refers to the process that determines object responses after a collision, including strategies to resolve the potential interpenetration situations that may have occurred. Notice that in the real world, interpenetration of rigid objects can never occur since collisions are strictly governed by the laws of physics. As such, resolutions of interpenetrations are relevant only in a simulated virtual world where movements are approximated and impossible situations may occur. These situations must be resolved algorithmically where both the computational cost and resulting visual appearance should be acceptable.

In general, there are three common methods for responding to interpenetrating collisions. The first is to simply displace the objects from one another by the depth of penetration. This is known as the Projection Method since you simply move positions of objects such that they no longer overlap. While this is simple to calculate and implement, it lacks stability when many objects are in proximity and overlap with each other. In this case, the simple resolution of one pair of interpenetrating objects can result in new penetrations with other nearby objects. However, the Projection Method is still often implemented in simple engines or games with simple object interaction rules. For example, in a game of Pong, the ball never comes to rest on the paddles or walls and remains in continuous motion by bouncing off any object it collides with. The Projection Method is perfect for resolving collisions for these types of simple object interactions.

1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966

CHAPTER 9 SIMULATING THE WORLD WITH RIGIDSHAPES

1967 The second method, the Impulse Method, uses object velocities to compute and
1968 apply impulses to cause the objects to move in the opposite directions at the point of
1969 collision. This method tends to slow down colliding objects rapidly and converges to
1970 relatively stable solutions. This is because impulses are computed based on the transfer
1971 of momentum, which in turn has a damping effect on the velocities of the colliding
1972 objects.

1973 The third method, the Penalty Method, models the depth of object interpenetration
1974 as the degree of compression of a spring and approximates an acceleration to apply
1975 forces to separate the objects. This last method is the most complex and challenging to
1976 implement.

1977 For your engine, you will be combining the strengths of the Projection and Impulse
1978 Methods. The Projection Method will be used to separate the interpenetrating objects,
1979 while the Impulse Method will be used to compute impulses to reduce the object
1980 velocities in the direction that caused the interpenetration. As described, the simple
1981 Projection Method can result in an unstable system, such as objects that sink into each
1982 other when stacked. You will overcome this instability by implementing a relaxation loop
1983 where, in a single update cycle, interpenetrated objects are separated incrementally via
1984 repeated applications of the Projection Method.

1985 With a relaxation loop, each application of the Projection Method is referred to as
1986 a relaxation iteration. During each relaxation iteration, the Projection Method reduces
1987 the interpenetration incrementally by a fixed percentage of the total penetration depth.
1988 For example, by default, the engine sets relaxation iterations to 15, and each relaxation
1989 iteration reduces the interpenetration by 80 percent. This means that within one update
1990 function call, after the movement integration approximation, the collision detection and
1991 resolution procedures will be executed 15 times. While costly, the repeated incremental
1992 separation ensures a stable system.

1993 **The Collision Position Correction Project**

1994 This project will guide you through the implementation of the relaxation iterations to
1995 incrementally resolve inter-object interpenetrations. You are going to use the collision
1996 information computed from previous project to correct the position of the colliding
1997 objects. You can see an example of this project running in Figure 9-24. The source code
1998 to this project is defined in chapter9/9.6.collision_position_correction.

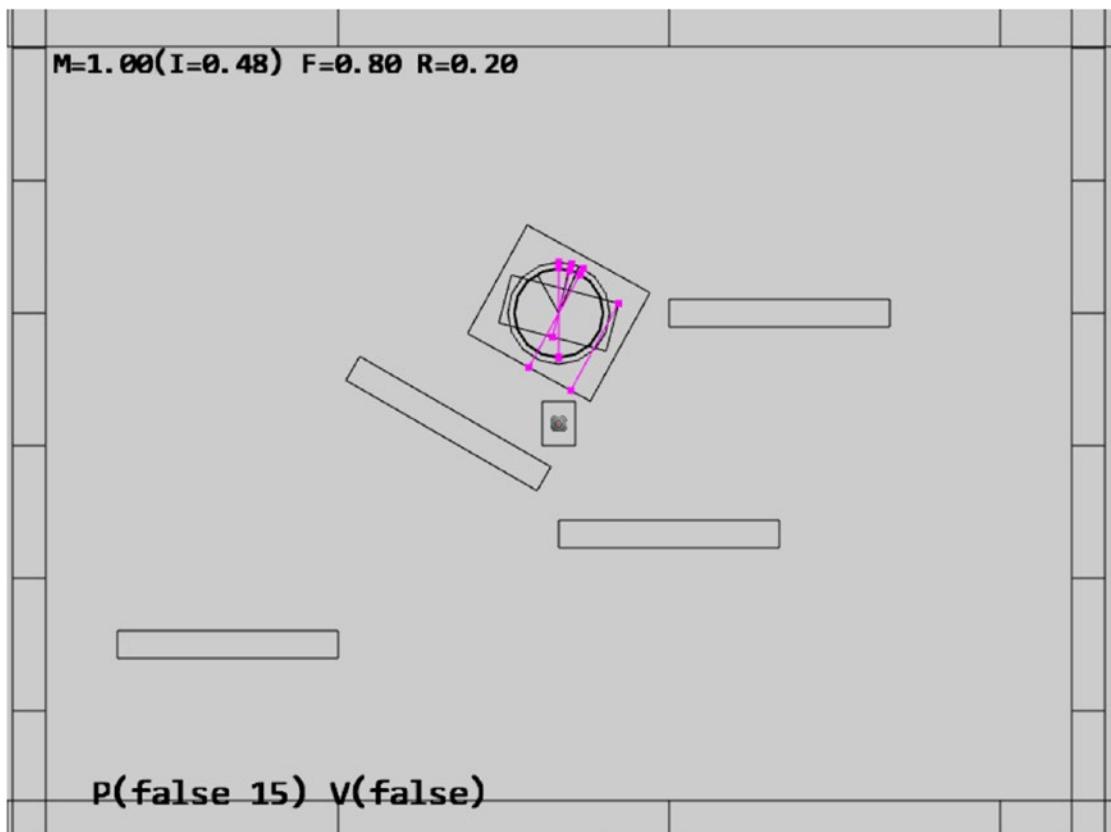


Figure 9-24. Running the Collision Position Correction project

The controls of the project are identical to the previous project with a single addition of the P key command in behavior control:

- **Behavior control:**
 - **P key:** Toggles penetration resolution for all objects
 - **V key:** Toggles motion of all objects
 - **H key:** Injects random velocity to all objects
 - **G key:** Randomly creates a new rigid circle or rectangle
- **Draw control:**
 - **C key:** Toggles the drawing of all CollisionInfo
 - **T key:** Toggles textures on all objects

CHAPTER 9 SIMULATING THE WORLD WITH RIGIDSHAPES

- 2009 • **R key:** Toggles the drawing of RigidShape
- 2010 • **B key:** Toggles the drawing of the bound on each RigidShape
- 2011 • **Object control:**
 - 2012 • **Left-/right-arrow key:** Sequences through and selects an object.
 - 2013 • **WASD keys:** Move the selected object.
 - 2014 • **Z/X key:** Rotates the selected object.
 - 2015 • **Y/U key:** Increases/decreases RigidShape size of the selected object; this does not change the size of the corresponding Renderable object.
 - 2016 • **Up-/down-arrow key + M:** Increase/decrease the mass of the selected object.

2019 The goals of the project are as follows:

- 2021 • To implement positional correction with relaxation iteration
- 2022 • To work with the computed collision information and appreciate its importance
- 2023 • To understand and experience implementing interpenetration resolution

2026 **Updating the Physics Component**

2027 The previous projects have established the required simulation infrastructure including
2028 the completion of the RigidShape implementation. You can now focus on the details
2029 of positional correction logic which is localized and hidden in the core of the physics
2030 component in the `physics.js` file in the `src/engine/components` folder.

- 2031 1. Edit `physics.js` to define variables and the associated getters
2032 and setters for positional correction rate, relaxation loop count,
2033 and, toggling the positional correction computation. Make sure to
2034 export the newly defined functions.

```
2035 let mPosCorrectionRate = 0.8; // % separation to project objects
2036 let mRelaxationCount = 15; // number of relaxation iteration
2037 let mCorrectPosition = true;
```

```

function getPositionalCorrection() { return mCorrectPosition; }          2038
function togglePositionalCorrection() {                                2039
    mCorrectPosition = !mCorrectPosition;                            2040
}
}                                                                    2041

function getRelaxationCount() { return mRelaxationCount; }           2042
function incRelaxationCount(dc) { mRelaxationCount += dc; }          2043
... identical to previous code ...                                2044

export {                                                       2045
    ... identical to previous code ...                          2046

    togglePositionalCorrection,                           2047
    getPositionalCorrection,                         2048
    getRelaxationCount,                            2049
    incRelaxationCount                           2050
}
}                                                                    2051

```

2. Define the positionalCorrection() function to move and
reduce the overlaps between objects by the predefined rate,
mPosCorrectionRate. To properly support object momentum
in the simulation, the amount in which each object moves is
inversely proportional to their masses. That is, upon collision,
an object with a larger mass will be moved by an amount that is
less than the object with a smaller mass. Notice that the direction
of movement is along the collision normal as defined in by the
collisionInfo object. 2052

```

function positionalCorrection(s1, s2, collisionInfo) {           2061
    if (!mCorrectPosition)                                     2062
        return;                                              2063

    let s1InvMass = s1.getInvMass();                           2064
    let s2InvMass = s2.getInvMass();                           2065

    let num = collisionInfo.getDepth() /                      2066
            (s1InvMass + s2InvMass) * mPosCorrectionRate;      2067
    let correctionAmount = [0, 0];                            2068

```

CHAPTER 9 SIMULATING THE WORLD WITH RIGIDSHAPES

```
2069     vec2.scale(correctionAmount, collisionInfo.getNormal(), num);
2070     s1.adjustPositionBy(correctionAmount, -s1InvMass);
2071     s2.adjustPositionBy(correctionAmount, s2InvMass);
2072 }
```

2073 3. Modify the `collideShape()` function to perform positional
2074 correction when a collision is detected. Notice that collision
2075 detection is performed only when at least one of the objects is
2076 with nonzero masses.

```
2077 function collideShape(s1, s2, infoSet = null) {
2078     ... identical to previous code ...
2079     if ((s1 !== s2) &&
2080         ((s1.getInvMass() != 0) || (s2.getInvMass() != 0))) {
2081         if (s1.boundTest(s2)) {
2082             hasCollision = s1.collisionTest(s2, mCInfo);
2083             if (hasCollision) {
2084                 vec2.subtract(mS1toS2, s2.getCenter(), s1.getCenter());
2085                 if (vec2.dot(mS1toS2, mCInfo.getNormal()) < 0)
2086                     mCInfo.changeDir();
2087                 positionalCorrection(s1, s2, mCInfo);
2088
2089             }
2090             return hasCollision;
2091 }
```

2092 4. Integrate a loop in all three utility functions, `processObjToSet()`,
2093 `processSetToSet()`, and `processSet()`, to execute relaxation
2094 iterations when performing the positional corrections:

```
2095 function processObjToSet(obj, set, infoSet = null) {
2096     let j = 0, r = 0;
2097     let hasCollision = false;
2098     let s1 = obj.getRigidBody();
2099     for (r = 0; r < mRelaxationCount; r++) {
2100         for (j = 0; j < set.size(); j++) {
2101             let s2 = set.getObjectAt(j).getRigidBody();
```

```

        hasCollision = collideShape(s1, s2, infoSet) || hasCollision; 2102
    }
}
return hasCollision;
} 2106
function processSetToSet(set1, set2, infoSet = null) { 2107
    let i = 0, j = 0, r = 0; 2108
    let hasCollision = false; 2109
    for (r = 0; r < mRelaxationCount; r++) { 2110
        ... identical to previous code ... 2111
    } 2112
    return hasCollision; 2113
} 2114
// collide all objects in the GameObjectSet with themselves
function processSet(set, infoSet = null) { 2115
    let i = 0, j = 0, r = 0; 2116
    let hasCollision = false; 2117
    for (r = 0; r < mRelaxationCount; r++) { 2118
        ... identical to previous code ... 2119
    } 2120
    return hasCollision; 2121
} 2122
} 2123

```

Testing Positional Correction in MyGame

The MyGame class must be modified to support the new P key command, to toggle off initial motion, positional correction, and to spawn initial objects in the central region of the game scene to guarantee initial collisions. These modifications are straightforward and details are not shown. As always, you can refer to the source code files in the `src/my_game` folder for implementation details.

Observations

You can now run the project to test your implementation. Notice that by default, motion is off, positional correction is off, and showing of collision information is on. For these reasons, you will observe the created rigid shapes clumping in the central region of the game scene with many associated magenta collision information.

2135 Now, type the P key and observe all of the shapes being pushed apart with all
2136 overlaps resolved. You can type the G key to create additional shapes and observe the
2137 shapes continuously push each other aside to ensure no overlaps. A fun experiment to
2138 perform is to toggle off positional correction, followed by typing the G key to create a
2139 large number of overlapping shapes and then to type the P key to observe the shapes
2140 pushing each other apart.

2141 If you switch on motion with the V key, you will first observe all objects free falling as
2142 a result of the gravitational force. These objects will eventually come to a rest on one of
2143 the stationary platforms. Next, you will observe the magenta collision depth increasing
2144 continuously in the vertical direction. This increase in size is a result of the continuously
2145 increasing downward velocity as a result of the downward gravitational acceleration.
2146 Eventually, the downward velocity will grow so large that in one update the object
2147 will move past and appear to fall right through the platform. What you are observing
2148 is precisely the situation discussed in Figure 9-22. The next subsection will discuss
2149 responses to collision and address this ever-increasing velocity.

2150 Lastly, notice that the utility functions defined in the physics component, the
2151 processSet(), processObjToSet(), and processSetToSet(), these are designed to
2152 detect and resolve collisions. While useful, these functions are not designed to report on
2153 if a collision has occurred—a common operation supported by typical physics engines.
2154 To avoid distraction from the rigid shape simulation discussion, functions to support
2155 simple collision detection without responses are not presented. At this point, you have
2156 the necessary knowledge to define such functions, and it is left as an exercise for you to
2157 complete.

2158 Collision Resolution

2159 With a proper positional correction system, you can now begin implementing collision
2160 resolution and support behaviors that resemble real-world situations. In order to focus
2161 on the core functionality of a collision resolution system, including understanding and
2162 implementing the Impulse Method and ensuring system stability, you will begin by
2163 examining collision responses without rotations. After the mechanics behind simple
2164 impulse resolution are fully understood and implemented, the complications associated
2165 with angular impulse resolutions will be examined in the next section.

In the following discussion, the rectangles and circles will not rotate as a response to collisions. However, the concepts and implementation described can be generalized in a straightforward manner to support rotational collision responses. This project is designed to help you understand the basic concepts of impulse-based collision resolutions.

The Impulse Method

You will formulate the solution for the Impulse Method by first reviewing how a circle can bounce off of a wall and other circles in a perfect world. This will subsequently be used to derive an approximation for an appropriate collision response. Note that the following discussion focuses on deriving the formulation for the Impulse Method and does not attempt to present a review on Newtonian mechanics. Here is a brief review of some of the relevant terms:

- **Mass:** Is the amount of matter in an object or how dense an object is.
- **Force:** Is any interaction or energy imparted on an object that will change the motion of that object.
- **Relative velocity:** Is the difference in velocity between two traveling shapes.
- **Coefficient of restitution:** The ratio of relative velocity from after and before a collision. This is a measurement of how much kinetic energy remains after an object bounces off another, or bounciness.
- **Coefficient of friction:** The ratio of the force of friction between two bodies. In your very simplistic implementation, friction is applied directly to slow down linear motion or rotation.
- **Impulse:** Accumulated force over time that can cause a change in the velocity. For example, resulting from a collision.

Note Object rotations are described by their angular velocities and will be examined in the next section. In the rest of this section, the term velocity is used to refer to the movements of objects or their linear velocity.

2194 Components of Velocity in a Collision

2195 Figure 9-25 illustrates a circle A in three different stages. At stage 1, the circle is traveling
 2196 at velocity \vec{V}_1 toward the wall on its right. At stage 2, the circle is colliding with the wall,
 2197 and at stage 3, the circle has been reflected and is traveling away from the wall with
 2198 velocity \vec{V}_2 .

this figure will be printed in b/w

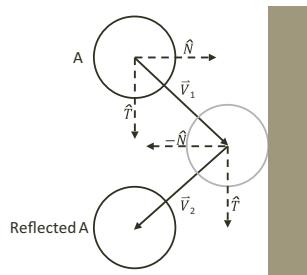


Figure 9-25. Collision between a circle and a wall in a perfect world

2199 Mathematically, this collision and the response can be described by decomposing
 2200 the initial velocity, \vec{V}_1 , into the components that are perpendicular and parallel to the
 2201 colliding wall. In general, the perpendicular direction to a collision is referred to as the
 2202 collision normal, \hat{N} , and the direction that is tangential to the collision position is the
 2203 collision tangent \hat{T} . This decomposition can be seen in the following equation:

$$\bullet \quad \vec{V}_1 = (\vec{V}_1 \cdot \hat{N})\hat{N} + (\vec{V}_1 \cdot \hat{T})\hat{T}$$

2205 In a perfect world with no friction and no loss of kinetic energy, a collision will not
 2206 affect the component along the tangent direction while the normal component will
 2207 simply be reversed. In this way, the reflected vector \vec{V}_2 can be expressed as a linear
 2208 combination of the normal and tangent components of \vec{V}_1 as follows:

$$\bullet \quad \vec{V}_2 = -(\vec{V}_1 \cdot \hat{N})\hat{N} + (\vec{V}_1 \cdot \hat{T})\hat{T}$$

2210 Notice the negative sign in front of the \hat{N} component. You can see in Figure 9-25
 2211 that the \hat{N} component for vector \vec{V}_2 points in the opposite direction to that of \vec{V}_1 as a
 2212 result of the collision. Additionally, notice that in the tangent direction \hat{T} , \vec{V}_2 continues
 2213 to point in the same direction. This is because the tangent component is parallel to
 2214 the of the wall and is unaffected by the collision. This analysis is true in general for any
 2215 collisions in a perfect world with no friction and no loss of kinetic energy.

Relative Velocity of Colliding Shapes

2216

The decomposition of vectors into the normal and tangent directions of the collision can also be applied to the general case of when both of the colliding shapes are in motion. For example, Figure 9-26 illustrates two traveling circle shapes, A and B, coming into a collision.

2217

2218

2219

2220

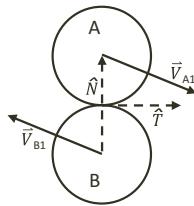


Figure 9-26. Collision between two traveling circles

this figure will be printed in b/w

In the case of Figure 9-26, before the collision, object A is traveling with velocity \vec{V}_{A1} , while object B with velocity \vec{V}_{B1} . The normal direction of the collision, \hat{N} , is defined to be the vector between the two circle centers, and the tangent direction of the collision, \hat{T} , is the vector that is tangential to both of the circles at the point of collision. To resolve this collision, the velocities for objects A and B after the collision, \vec{V}_{A2} and \vec{V}_{B2} , must be computed.

2221

2222

2223

2224

2225

2226

The post-collision velocities are determined based on the relative velocity between the two shapes. The relative velocity between shapes A and B is defined as follows:

2227

2228

- $\vec{V}_{AB1} = \vec{V}_{A1} - \vec{V}_{B1}$

2229

The collision vector decomposition can now be applied to the normal and tangent directions of the relative velocity where the relative velocity after the collision is \vec{V}_{AB2} :

2230

2231

- $\vec{V}_{AB2} \cdot \hat{N} = -e(\vec{V}_{AB1} \cdot \hat{N})$ (1)

2232

- $\vec{V}_{AB2} \cdot \hat{T} = f(\vec{V}_{AB1} \cdot \hat{T})$ (2)

2233

The restitution, e, and friction, f, coefficients model the real-world situation where some kinetic energy is changed to some other forms of energy during the collision. The negative sign of Equation (1) signifies that after the collision, objects will travel in the direction that is opposite to the initial collision normal direction. Equation (2) says that after the collision, friction will scale back the magnitude where objects will continue to

2234

2235

2236

2237

2238

2239 travel in the same tangent direction only at a lower velocity. Notice that all variables on
 2240 the right-hand side of Equations (1) and (2) are defined, as they are known at the time of
 2241 collision. It is important to remember that

2242 • $\vec{V}_{AB2} = \vec{V}_{A2} - \vec{V}_{B2}$

2243 where the goal is to derive a solution for \vec{V}_{A2} and \vec{V}_{B2} , the individual velocities of the
 2244 colliding objects after a collision. You are now ready to model a solution to approximate
 2245 \vec{V}_{A2} and \vec{V}_{B2} .

2246 **Note** The restitution coefficient, e , describes bounciness or the proportion of the
 2247 velocity that is retained after a collision. A restitution value of 1.0 would mean that
 2248 velocities will be the same from before and after a collision. In contrast, friction is
 2249 intuitively associated with the proportion lost or the slow down after a collision. For
 2250 example, a friction coefficient of 1.0 would mean infinite friction where a velocity
 2251 of zero will result from a collision. For consistency of the formulae, the coefficient f
 2252 in Equation (2) is actually 1 minus the intuitive friction coefficient.

2253 **The Impulse**

2254 Accurately describing a collision involves complex considerations including factors
 2255 like energy-changing form, or frictions resulting from different material properties, etc.
 2256 Without considering these advanced issues, a simplistic description of a collision that
 2257 occurs on a shape is a constant mass object changing its velocity from \vec{V}_{A1} to \vec{V}_{A2} after
 2258 contacting with another object. Conveniently, this is the definition of an impulse, as can
 2259 be seen in the following:

2260 • $\vec{J} = m_A \vec{V}_{A2} - m_A \vec{V}_{A1}$

2261 or when solving for \vec{V}_{A2}

2262 • $\vec{V}_{A2} = \vec{V}_{A1} + \frac{\vec{J}}{m_A}$ (3)

Remember that the same impulse also causes the velocity change in object B, only in
the opposite direction:

- $\vec{J} = -(m_B \vec{V}_{B2} - m_B \vec{V}_{B1})$

or when solving for \vec{V}_{B2}

- $\vec{V}_{B2} = \vec{V}_{B1} - \frac{\vec{J}}{m_B}$ (4)

Take a step back from the math and think about what this formula states. It makes
intuitive sense. The equation states that the change in velocity is inversely proportional
to the mass of an object. In other words, the more mass an object has, the less its velocity
will change after a collision. The Impulse Method implements this observation.

Recall that Equations (1) and (2) describe the relative velocity after collision
according to the collision normal and tangent directions independently. The impulse,
being a vector, can also be expressed as a linear combination of components in the
collision normal and tangent directions, j_N and j_T :

- $\vec{J} = j_N \hat{N} + j_T \hat{T}$

Substituting this expression into Equations (3) and (4) results in the following:

- $\vec{V}_{A2} = \vec{V}_{A1} + \frac{j_N}{m_A} \hat{N} + \frac{j_T}{m_A} \hat{T}$ (5)

- $\vec{V}_{B2} = \vec{V}_{B1} - \frac{j_N}{m_B} \hat{N} - \frac{j_T}{m_B} \hat{T}$ (6)

Note that j_N and j_T are the only unknowns in these two equations where the rest of
the terms are either defined by the user or can be computed based on the geometric
shapes. That is, the quantities \vec{V}_{A1} , \vec{V}_{B1} , m_A , and m_B are defined by the user, and \hat{N} and
 \hat{T} can be computed.

Note The \hat{N} and \hat{T} vectors are normalized and perpendicular to each other. For
this reason, the vectors have a value of 1 when dotted with themselves and a value
of 0 when dotted with each other.

Normal Component of the Impulse

The normal component of the impulse, j_N , can be solved by performing a dot product with the \hat{N} vector on both sides of Equations (5) and (6):

$$\vec{V}_{A2} \cdot \hat{N} = \left(\vec{V}_{A1} + \frac{j_N}{m_A} \hat{N} + \frac{j_T}{m_A} \hat{T} \right) \cdot \hat{N} = \vec{V}_{A1} \cdot \hat{N} + \frac{j_N}{m_A}$$

$$\vec{V}_{B2} \cdot \hat{N} = \left(\vec{V}_{B1} - \frac{j_N}{m_B} \hat{N} - \frac{j_T}{m_B} \hat{T} \right) \cdot \hat{N} = \vec{V}_{B1} \cdot \hat{N} - \frac{j_N}{m_B}$$

Subtracting the preceding two equations results in the following:

$$(\vec{V}_{A2} - \vec{V}_{B2}) \cdot \hat{N} = (\vec{V}_{A1} - \vec{V}_{B1}) \cdot \hat{N} + j_N \left(\frac{1}{m_A} + \frac{1}{m_B} \right)$$

Recall that $(\vec{V}_{A2} - \vec{V}_{B2})$ is simply \vec{V}_{AB2} and that $(\vec{V}_{A1} - \vec{V}_{B1})$ is \vec{V}_{AB1} , and this equation simplifies to the following:

$$\vec{V}_{AB2} \cdot \hat{N} = \vec{V}_{AB1} \cdot \hat{N} + j_N \left(\frac{1}{m_A} + \frac{1}{m_B} \right)$$

Substituting Equation (1) for the left-hand side derives the following equation:

$$-e(\vec{V}_{AB1} \cdot \hat{N}) = \vec{V}_{AB1} \cdot \hat{N} + j_N \left(\frac{1}{m_A} + \frac{1}{m_B} \right)$$

Collecting terms and solving for j_N , the impulse in the normal direction, result in the following:

$$j_N = \frac{-(1+e)(\vec{V}_{AB1} \cdot \hat{N})}{\frac{1}{m_A} + \frac{1}{m_B}} \quad (7)$$

Tangent Component of the Impulse

2302

The tangent component of the impulse, j_T , can be solved by performing a dot product with the \hat{T} vector on both sides of Equations (5) and (6):

$$\bullet \quad \vec{V}_{A2} \cdot \hat{T} = \left(\vec{V}_{A1} + \frac{\vec{j}_N}{m_A} \hat{N} + \frac{\vec{j}_T}{m_A} \hat{T} \right) \cdot \hat{T} = \vec{V}_{A1} \cdot \hat{T} + \frac{\vec{j}_T}{m_A} \quad 2305$$

$$\bullet \quad \vec{V}_{B2} \cdot \hat{T} = \left(\vec{V}_{B1} - \frac{\vec{j}_N}{m_B} \hat{N} - \frac{\vec{j}_T}{m_B} \hat{T} \right) \cdot \hat{T} = \vec{V}_{B1} \cdot \hat{T} - \frac{\vec{j}_T}{m_B} \quad 2306$$

Following the similar steps as in the case for the normal component, subtracting the equations, and recognizing $(\vec{V}_{A2} - \vec{V}_{B2})$ is \vec{V}_{AB2} and $(\vec{V}_{A1} - \vec{V}_{B1})$ is \vec{V}_{AB1} derive the following equation:

$$\bullet \quad \vec{V}_{AB2} \cdot \hat{T} = \vec{V}_{AB1} \cdot \hat{T} + j_T \left(\frac{1}{m_A} + \frac{1}{m_B} \right) \quad 2310$$

Now, substituting Equation (2) for the left-hand side leaves the following:

$$\bullet \quad f(\vec{V}_{AB1} \cdot \hat{T}) = \vec{V}_{AB1} \cdot \hat{T} + j_T \left(\frac{1}{m_A} + \frac{1}{m_B} \right) \quad 2312$$

Finally, collecting terms and solving for j_T or the impulse in the tangent direction result in the following:

$$\bullet \quad j_T = \frac{(f-1)(\vec{V}_{AB1} \cdot \hat{T})}{\frac{1}{m_A} + \frac{1}{m_B}} \quad (8) \quad 2315$$

The Collision Resolution Project

2316

This project will guide you through resolving a collision by calculating the impulse and updating the velocities of the colliding objects. You can see an example of this project running in Figure 9-27. The source code for this project is defined in chapter9/9.7.collision_resolution.

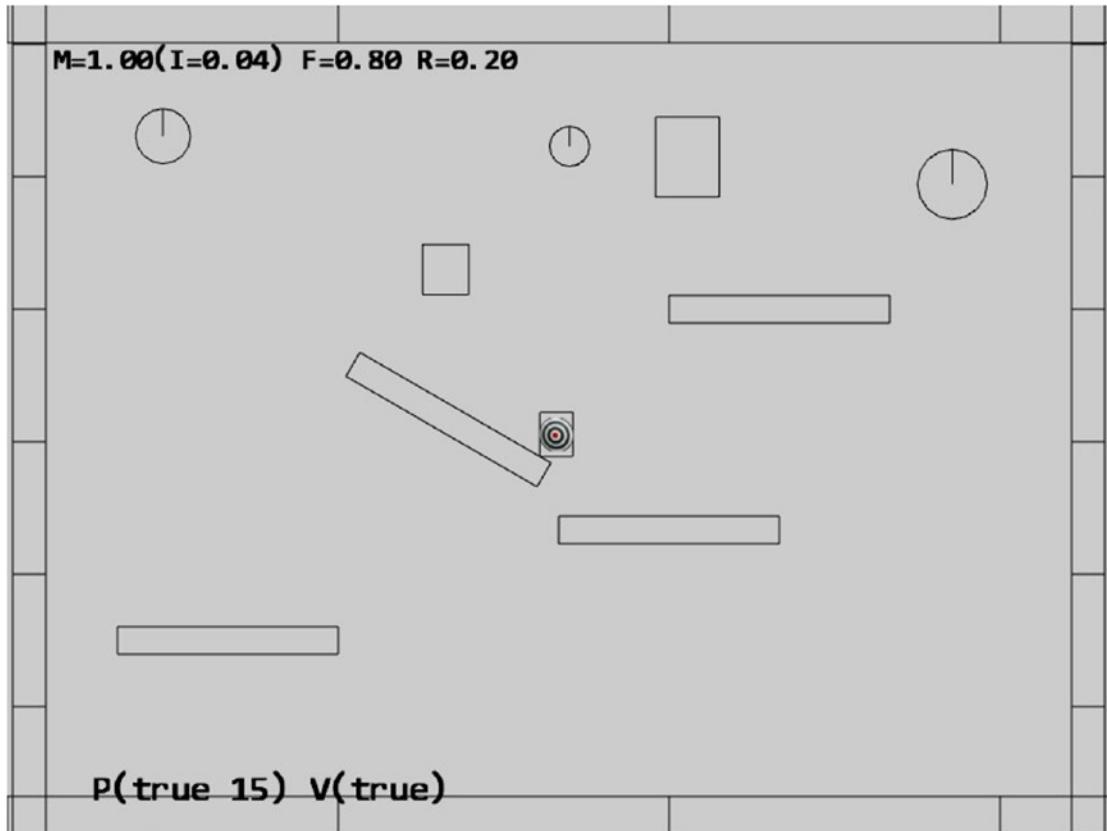


Figure 9-27. Running the Collision Resolution project

2321 The controls of the project are identical to the previous project with additional
 2322 controls for restitution and friction coefficients:

- 2323 • **Behavior control:**
 - 2324 • **P key:** Toggles penetration resolution for all objects
 - 2325 • **V key:** Toggles motion of all objects
 - 2326 • **H key:** Injects random velocity to all objects
 - 2327 • **G key:** Randomly creates a new rigid circle or rectangle
- 2328 • **Draw control:**
 - 2329 • **C key:** Toggles the drawing of all CollisionInfo
 - 2330 • **T key:** Toggles textures on all objects

- **R key:** Toggles the drawing of RigidBody 2331
- **B key:** Toggles the drawing of the bound on each RigidBody 2332
- **Object control:** 2333
 - **Left-/right-arrow key:** Sequences through and selects an object. 2334
 - **WASD keys:** Move the selected object. 2335
 - **Z/X key:** Rotates the selected object. 2336
 - **Y/U key:** Increases/decreases RigidBody size of the selected object; this does not change the size of the corresponding Renderable object. 2337
2338
2339
 - **Up-/down-arrow key + M/N/F:** Increase/decrease the mass/restitution/friction of the selected object. 2340
2341

The goals of the project are as follows: 2342

- To understand the details of the Impulse Method 2343
- To implement the Impulse Method in resolving collisions 2344

Updating the Physics Component

To properly support collision resolution, you only need to focus on the physics component and modify the `physics.js` file in the `src/engine/components` folder: 2346
2347

1. Edit `physics.js` and define the `resolveCollision()` function to 2348
resolve the collision between RigidBody objects, `a` and `b`, with 2349
collision information recorded in the `collisionInfo` object: 2350

```
function resolveCollision(b, a, collisionInfo) { 2351
  let n = collisionInfo.getNormal(); 2352

  // Step A: Compute relative velocity 2353
  let va = a.getVelocity(); 2354
  let vb = b.getVelocity(); 2355
  let relativeVelocity = [0, 0]; 2356
  vec2.subtract(relativeVelocity, va, vb); 2357
```

CHAPTER 9 SIMULATING THE WORLD WITH RIGIDSHAPES

```
2358 // Step B: Determine relative velocity in normal direction
2359 let rVelocityInNormal = vec2.dot(relativeVelocity, n);
2360 // if objects moving apart ignore
2361 if (rVelocityInNormal > 0) {
2362     return;
2363 }
2364 // Step C: Compute collision tangent direction
2365 let tangent = [0, 0];
2366 vec2.scale(tangent, n, rVelocityInNormal);
2367 vec2.subtract(tangent, tangent, relativeVelocity);
2368 vec2.normalize(tangent, tangent);
2369 // Relative velocity in tangent direction
2370 let rVelocityInTangent = vec2.dot(relativeVelocity, tangent);
2371 // Step D: Determine the effective coefficients
2372 let newRestituation = (a.getRestitution() + b.getRestitution()) * 0.5;
2373 let newFriction = 1 - ((a.getFriction() + b.getFriction()) * 0.5);
2374 // Step E: Impulse in the normal and tangent directions
2375 let jN = -(1 + newRestituation) * rVelocityInNormal;
2376 jN = jN / (a.getInvMass() + b.getInvMass());
2377 let jT = (newFriction - 1) * rVelocityInTangent;
2378 jT = jT / (a.getInvMass() + b.getInvMass());
2379 // Step F: Update velocity in both normal and tangent directions
2380 vec2.scaleAndAdd(va, va, n, (jN * a.getInvMass()));
2381 vec2.scaleAndAdd(va, va, tangent, (jT * a.getInvMass()));
2382 vec2.scaleAndAdd(vb, vb, n, -(jN * b.getInvMass()));
2383 vec2.scaleAndAdd(vb, vb, tangent, -(jT * b.getInvMass()));
2384 }
```

The listed code follows the solution derivation closely:

- a. **Steps A and B:** Compute the relative velocity and its normal component. When this normal component is positive, it signifies that two objects are moving away from each other and thus collision resolution is not necessary. 2385
2386
2387
2388
2389
 - b. **Step C:** Computes the collision tangent direction and the tangent component of the relative velocity. 2390
2391
 - c. **Step D:** Uses the averages of the coefficients for impulse derivation. Notice the subtraction by one when computing the newFriction for maintaining consistency with Equation (2). 2392
2393
2394
 - d. **Step E:** Follows the listed Equations (7) and (8) to compute the normal and tangent components of the impulse. 2395
2396
 - e. **Step F:** Solves for the resulting velocities by following Equations (5) and (6). 2397
2398
2. Edit collideShape() to invoke the resolveCollision() function when a collision is detected and position corrected: 2399
2400

```
function collideShape(s1, s2, infoSet = null) {
    let hasCollision = false;
    if ((s1 !== s2) &&
        ((s1.getInvMass() !== 0) || (s2.getInvMass() !== 0))) {
        if (s1.boundTest(s2)) {
            hasCollision = s1.collisionTest(s2, mCInfo);
            if (hasCollision) {
                ... identical to previous code ...

                positionalCorrection(s1, s2, mCInfo);
                resolveCollision(s1, s2, mCInfo);
            }
        }
    }
    ... identical to previous code ...
};
```

2413 Updating MyGame for Testing Collision Resolution

2414 The modifications to the `MyGame` class are trivial, mainly to toggle both motion and
 2415 positional correction to be active by default. Additionally, initial random rotations of the
 2416 created `RigidShape` objects are disabled because at this point, collision response does
 2417 not support rotation. As always, you can refer to the source code files in the `src/my_game`
 2418 folder for implementation details.

2419 Observations

2420 You should test your implementation in three ways. First, ensure that moving shapes
 2421 collide and behave naturally. Second, try changing the physical properties of the objects.
 2422 Third, observe the collision resolution between shapes that are in motion and shapes
 2423 that are stationary with infinite mass (the surrounding walls and stationary platforms).
 2424 Remember that only linear velocities are considered and rotations will not result from
 2425 collisions.

2426 Now, run the project and notice that the shapes fall gradually to the platforms and
 2427 floor with their motions coming to a halt after slight rebounds. This is a clear indication
 2428 that the base case for Euler Integration, collision detection, positional correction, and
 2429 resolution all are operating as expected. Press the H key to excite all shapes and the C
 2430 key to display the collision information. Notice the wandering shapes and the walls/
 2431 platforms interact properly with soft bounces and no apparent interpenetrations.

2432 Use the left/right arrow to select an object and adjust its restitution/friction
 2433 coefficients with the N/F and up-/down-arrow keys. For example, adjust the restitution
 2434 to 1 and friction to 0. Now inject velocity with the H key. Notice how the object seems
 2435 extra bouncy and, with a friction coefficient of 0, seems to skid along platforms/floors.
 2436 You can try different coefficient settings and observe corresponding bounciness and
 2437 slipperiness.

2438 The stability of the system can be tested by increasing the number of shapes in the
 2439 scene with the G key. The relaxation loop count of 15 continuously and incrementally
 2440 pushes interpenetrating shapes apart during each iteration. For example, you can toggle
 2441 off movement and positional corrections with the V and P keys and create multiple, for
 2442 example, 10 to 20, overlapping shapes. Now toggle on motion and positional corrections
 2443 and observe a properly functioning system.

2444 In the next project, you will improve the resolution solution to consider angular
 2445 velocity changes as a result of collisions.

Angular Components of Collision Responses

2446

Now that you have a concrete understanding and have successfully implemented the Impulse Method for collision responses with linear velocities, it is time to integrate the support for the more general case of rotations. Before discussing the details, it is helpful to relate the correspondences of Newtonian linear mechanics to that of rotational mechanics. That is, linear displacement corresponds to rotation, velocity to angular velocity, force to torque, and mass to rotational inertia or angular mass. Rotational inertia determines the torque required for a desired angular acceleration about a rotational axis.

2447

2448

2449

2450

2451

2452

2453

2454

The following discussion focuses on integrating rotation in the Impulse Method formulation and does not attempt to present a review on Newtonian mechanics for rotation. Conveniently, integrating proper rotation into the Impulse Method does not involve the derivation of any new algorithm. All that is required is the formulation of impulse responses with proper consideration of rotational attributes.

2455

2456

2457

2458

2459

Collisions with Rotation Consideration

2460

The key to integrating rotation into the Impulse Method formulation is recognizing the fact that the linear velocity you have been working with, for example, velocity \vec{V}_{A1} of object A, is actually the velocity of the shape at its center location. In the absence of rotation, this velocity is constant throughout the object and can be applied to any position. However, as illustrated in Figure 9-28, when the movement of an object includes angular velocity, $\vec{\omega}_{A1}$, its linear velocity at a position P, \vec{V}_{AP1} , is actually a function of the relative position between the point and the center of rotation of the shape or the positional vector \vec{R}_{AP} .

2461

2462

2463

2464

2465

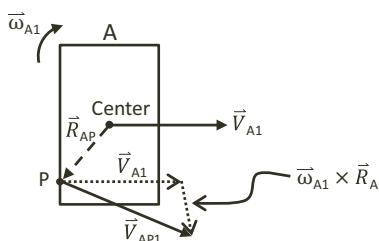
2466

2467

2468

- $\vec{V}_{AP1} = \vec{V}_{A1} + (\vec{\omega}_{A1} \times \vec{R}_{AP})$

2469



this figure will be printed in b/w

Figure 9-28. Linear velocity at a position in the presence of rotation

2470 **Note** Angular velocity is a vector that is perpendicular to the linear velocity.
2471 In this case, as linear velocity is defined on the X/Y plane, $\vec{\omega}$ is a vector in the z
2472 direction. Recall from discussions in the “Introduction” section of this chapter,
2473 the very first assumption made was that rigid shape objects are continuous
2474 geometries with uniformly distributed mass where the center of mass is located at
2475 the center of the geometric shape. This center of mass is the location of the axis of
2476 rotation. For simplicity, in your implementation, $\vec{\omega}$ will be stored as a simple scalar
2477 representing the z-component magnitude of the vector.

Figure 9-29 illustrates an object B with linear and angular velocities of \bar{V}_{B1} and $\bar{\omega}_{B1}$ colliding with object A at position P. By now, you know that the linear velocities at point P before the collision for the two objects are as follows:

- $\vec{V}_{AP1} = \vec{V}_{AI} + (\vec{\omega}_{AI} \times \vec{R}_{AP})$

$$\bullet \quad \vec{V}_{BP1} = \vec{V}_{B1} + (\vec{\omega}_{B1} \times \vec{R}_{BP}) \quad (10)$$

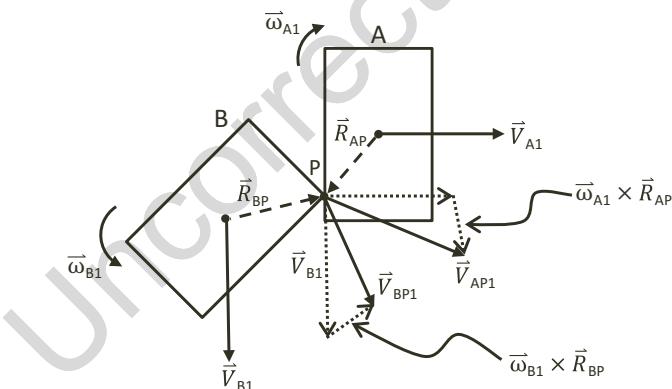


Figure 9-29. Colliding shapes with angular velocities

2483 After the collision, the linear velocity at the collision position can be expressed as
2484 follows:

$$\bullet \quad \vec{V}_{AP2} = \vec{V}_{A2} + (\vec{\omega}_{A2} \times \vec{R}_{AP}) \quad (11)$$

$$\bullet \quad \vec{V}_{BP2} = \vec{V}_{B2} + (\vec{\omega}_{B2} \times \vec{R}_{BP}) \quad (12)$$

where \vec{V}_{A2} and $\vec{\omega}_{A2}$, and \vec{V}_{B2} and $\vec{\omega}_{B2}$ are the linear and angular velocities for objects A and B after the collision, and the derivation of a solution for these quantities is precisely the goal of this section.

Relative Velocity with Rotation

Recall from the previous section that the definitions of relative velocity from before and after a collision between objects A and B are defined as follows:

- $\vec{V}_{AB1} = \vec{V}_{A1} - \vec{V}_{B1}$
- $\vec{V}_{AB2} = \vec{V}_{A2} - \vec{V}_{B2}$

These velocities are analyzed based on components in the collision normal and tangent directions in Equations (1) and (2) and are relisted for convenience in the following:

- $\vec{V}_{AB2} \cdot \hat{N} = -e(\vec{V}_{AB1} \cdot \hat{N})$ (1)
- $\vec{V}_{AB2} \cdot \hat{T} = f(\vec{V}_{AB1} \cdot \hat{T})$ (2)

These equations are derived without considering rotation and the formulation assumes that the velocity is constant over the entire shape. In order to support rotation, these equations must be generalized and solved at the point of collision, P.

- $\vec{V}_{ABP2} \cdot \hat{N} = -e(\vec{V}_{ABP1} \cdot \hat{N})$ (13)
- $\vec{V}_{ABP2} \cdot \hat{T} = f(\vec{V}_{ABP1} \cdot \hat{T})$ (14)

In this case, \vec{V}_{ABP1} and \vec{V}_{ABP2} are relative velocities at collision position P from before and after the collision. It is still true that these vectors are defined by the difference in velocities for objects A and B from before, \vec{V}_{A1} and \vec{V}_{B1} , and after, \vec{V}_{A2} and \vec{V}_{B2} , the collision at the collision position P on each object.

- $\vec{V}_{ABP1} = \vec{V}_{A1} - \vec{V}_{B1}$ (15)
- $\vec{V}_{ABP2} = \vec{V}_{A2} - \vec{V}_{B2}$ (16)

2511 You are now ready to generalize the Impulse Method to support rotation and to derive

2512 a solution to approximate the linear and angular velocities: \vec{V}_{A2} , \vec{V}_{B2} , $\vec{\omega}_{A2}$, and $\vec{\omega}_{B2}$.

2513 Impulse Method with Rotation

2514 Continue with the Impulse Method discussion from the previous section, that after the
 2515 collision between objects A and B, the Impulse Method describes the changes in their
 2516 linear velocities by an impulse, \vec{J} , scaled by the inverse of their corresponding masses,
 2517 m_A and m_B . This change in linear velocities is described in Equations (3) and (4), restated
 2518 as follows:

$$2519 \bullet \quad \vec{V}_{A2} = \vec{V}_{A1} + \frac{\vec{J}}{m_A} \quad (3)$$

$$2520 \bullet \quad \vec{V}_{B2} = \vec{V}_{B1} - \frac{\vec{J}}{m_B} \quad (4)$$

2521 In general, rotations are intrinsic results of collisions and the same impulse must
 2522 properly describe the change in angular velocity from before and after a collision.

2523 Remember that inertial, or rotational inertial, is the rotational mass. In a manner similar
 2524 to linear velocity and mass, it is also the case that the change in angular velocity in a
 2525 collision is inversely related to the rotational inertia. As illustrated in Figure 9-29, for
 2526 objects A and B with rotational inertia of I_A and I_B , after a collision, the angular velocities,
 2527 $\vec{\omega}_{A2}$ and $\vec{\omega}_{B2}$, can be described as follows, where \vec{R}_{AP} and \vec{R}_{BP} are the positional vectors
 2528 of each object:

$$2529 \bullet \quad \vec{\omega}_{A2} = \vec{\omega}_{A1} + \left(\vec{R}_{AP} \times \frac{\vec{J}}{I_A} \right) \quad (17)$$

$$2530 \bullet \quad \vec{\omega}_{B2} = \vec{\omega}_{B1} - \left(\vec{R}_{BP} \times \frac{\vec{J}}{I_B} \right) \quad (18)$$

2531 Recall from the previous section that it is convenient to express the impulse as a
 2532 linear combination of components in the collision normal and tangent directions, \hat{N}
 2533 and \hat{T} , or as shown:

$$2534 \bullet \quad \vec{J} = j_N \hat{N} + j_T \hat{T}$$

2535 Substituting this expression into Equation (17) results in the following:

$$2536 \bullet \quad \vec{\omega}_{A2} = \vec{\omega}_{A1} + \left(\vec{R}_{AP} \times \frac{j_N \hat{N} + j_T \hat{T}}{I_A} \right) = \vec{\omega}_{A1} + \frac{j_N}{I_A} (\vec{R}_{AP} \times \hat{N}) + \frac{j_T}{I_A} (\vec{R}_{AP} \times \hat{T})$$

In this way, Equations (17) and (18) can be expanded to describe the change in angular velocities caused by the normal and tangent components of the impulse as follows:

$$\bullet \quad \vec{\omega}_{A2} = \vec{\omega}_{A1} + \frac{j_N}{I_A} (\vec{R}_{AP} \times \hat{N}) + \frac{j_T}{I_A} (\vec{R}_{AP} \times \hat{T}) \quad (19) \quad 2539$$

$$\bullet \quad \vec{\omega}_{B2} = \vec{\omega}_{B1} - \frac{j_N}{I_B} (\vec{R}_{BP} \times \hat{N}) - \frac{j_T}{I_B} (\vec{R}_{BP} \times \hat{T}) \quad (20) \quad 2540$$

The corresponding equations describing linear velocity changes, Equations (5) and (6), are relisted in the following:

$$\bullet \quad \vec{V}_{A2} = \vec{V}_{A1} + \frac{j_N}{m_A} \hat{N} + \frac{j_T}{m_A} \hat{T} \quad (5) \quad 2543$$

$$\bullet \quad \vec{V}_{B2} = \vec{V}_{B1} - \frac{j_N}{m_B} \hat{N} - \frac{j_T}{m_B} \hat{T} \quad (6) \quad 2544$$

You can now substitute Equations (5) and (19) into Equation (11) and Equations (6) and (20) into Equation (12):

$$\bullet \quad \vec{V}_{AP2} = \left(\vec{V}_{A1} + \frac{j_N}{m_A} \hat{N} + \frac{j_T}{m_A} \hat{T} \right) + \left(\vec{\omega}_{A1} + \frac{j_N}{I_A} (\vec{R}_{AP} \times \hat{N}) + \frac{j_T}{I_A} (\vec{R}_{AP} \times \hat{T}) \right) \times \vec{R}_{AP} \quad (21) \quad 2547$$

$$\bullet \quad \vec{V}_{BP2} = \left(\vec{V}_{B1} - \frac{j_N}{m_B} \hat{N} - \frac{j_T}{m_B} \hat{T} \right) + \left(\vec{\omega}_{B1} - \frac{j_N}{I_B} (\vec{R}_{BP} \times \hat{N}) - \frac{j_T}{I_B} (\vec{R}_{BP} \times \hat{T}) \right) \times \vec{R}_{BP} \quad (22) \quad 2548$$

It is important to reiterate that the changes to both linear and angular velocities are described by the same impulse, \vec{J} . In other words, the normal and tangent impulse components j_N and j_T in Equations (21) and (22) are the same quantities, and these two are the only unknowns in these equations where the rest of the terms are values either defined by the user or can be computed based on the geometric shapes. That is, the quantities \vec{V}_{A1} , \vec{V}_{B1} , m_A , m_B , $\vec{\omega}_{A1}$, $\vec{\omega}_{B1}$, I_A , and I_B , are defined by the user and \hat{N} , \hat{T} , \vec{R}_{AP} , and \vec{R}_{BP} can be computed. You are now ready to derive the solutions for j_N and j_T .

Note In the following derivation, it is important to remember the definition of triple scalar product identity; this identity states that given vectors, \vec{D} , \vec{E} , and, \vec{F} , the following is always true:

$$(\vec{D} \times \vec{E}) \cdot \vec{F} \equiv \vec{D} \cdot (\vec{E} \times \vec{F}) \quad 2559$$

Normal Components of the Impulse

The normal component of the impulse, j_N , can be approximated by assuming that the contribution from the angular velocity tangent component is minimal and can be ignored and isolating the normal components from Equations (21) and (22). For clarity, you will work with one equation at a time and begin with Equation (21) for object A.

Now, ignore the tangent component of the angular velocity and perform a dot product with the \hat{N} vector on both sides of Equation (21) to isolate the normal components:

$$\bullet \quad \vec{V}_{AP2} \cdot \hat{N} = \left\langle \left(\vec{V}_{A1} + \frac{j_N}{m_A} \hat{N} + \frac{j_T}{m_A} \hat{T} \right) + \left(\vec{\omega}_{A1} + \frac{j_N}{I_A} (\vec{R}_{AP} \times \hat{N}) \right) \times \vec{R}_{AP} \right\rangle \cdot \hat{N}$$

Carry out the dot products on the right-hand side, recognizing \hat{N} is a unit vector and is perpendicular to \hat{T} , and let $\vec{D} = \vec{R}_{AP} \times \hat{N}$; then, this equation can be rewritten as follows:

$$\bullet \quad \vec{V}_{AP2} \cdot \hat{N} = (\vec{V}_{A1} \cdot \hat{N}) + \frac{j_N}{m_A} + (\vec{\omega}_{A1} \times \vec{R}_{AP}) \cdot \hat{N} + \frac{j_N}{I_A} (\vec{D} \times \vec{R}_{AP}) \cdot \hat{N} \quad (23)$$

The vector operations of the rightmost term in Equation (23) can be simplified by applying the triple scalar product identity and remembering that $\vec{D} = \vec{R}_{AP} \times \hat{N}$:

$$\bullet \quad (\vec{D} \times \vec{R}_{AP}) \cdot \hat{N} = \vec{D} \cdot (\vec{R}_{AP} \times \hat{N}) = (\vec{R}_{AP} \times \hat{N}) \cdot (\vec{R}_{AP} \times \hat{N}) = \|\vec{R}_{AP} \times \hat{N}\|^2$$

With this manipulation and collection of the terms with a dot product, Equation (23) becomes the following:

$$\bullet \quad \vec{V}_{AP2} \cdot \hat{N} = (\vec{V}_{A1} + (\vec{\omega}_{A1} \times \vec{R}_{AP})) \cdot \hat{N} + \frac{j_N}{m_A} + \frac{j_N}{I_A} \|\vec{R}_{AP} \times \hat{N}\|^2$$

From Equation (9), on the right-hand side, the term with the dot product is simply \vec{V}_{AP1} :

$$\bullet \quad \vec{V}_{AP2} \cdot \hat{N} = \vec{V}_{AP1} \cdot \hat{N} + \frac{j_N}{m_A} + \frac{j_N}{I_A} \|\vec{R}_{AP} \times \hat{N}\|^2 \quad (24)$$

Equation (22) can be processed through an identical algebraic manipulation steps by ignoring the tangent component of the angular velocity and performing a dot product with the \hat{N} vector on both sides of the equation; the following can be derived:

$$\bullet \quad \vec{V}_{BP2} \cdot \hat{N} = \vec{V}_{BP1} \cdot \hat{N} - \frac{j_N}{m_B} - \frac{j_N}{I_B} \|\vec{R}_{BP} \times \hat{N}\|^2 \quad (25)$$

Subtracting Equation (25) from (24) results in the following:

$$\bullet \quad (\vec{V}_{AP2} - \vec{V}_{BP2}) \cdot \hat{N} = (\vec{V}_{AP1} - \vec{V}_{BP1}) \cdot \hat{N} + j_N \left(\frac{1}{m_A} + \frac{1}{m_B} + \frac{\|\vec{R}_{AP} \times \hat{N}\|^2}{I_A} + \frac{\|\vec{R}_{BP} \times \hat{N}\|^2}{I_B} \right) \quad 2585$$

Substituting Equation (16) followed by (13) on the left-hand side and Equation (15) on the right-hand side, you get the following:

$$\bullet \quad -e(\vec{V}_{ABP1} \cdot \hat{N}) = \vec{V}_{ABP1} \cdot \hat{N} + j_N \left(\frac{1}{m_A} + \frac{1}{m_B} + \frac{\|\vec{R}_{AP} \times \hat{N}\|^2}{I_A} + \frac{\|\vec{R}_{BP} \times \hat{N}\|^2}{I_B} \right) \quad 2588$$

Lastly, collect terms and solve for j_N :

$$\bullet \quad j_N = \frac{-(1+e)(\vec{V}_{ABP1} \cdot \hat{N})}{\frac{1}{m_A} + \frac{1}{m_B} + \frac{\|\vec{R}_{AP} \times \hat{N}\|^2}{I_A} + \frac{\|\vec{R}_{BP} \times \hat{N}\|^2}{I_B}} \quad 2590$$
(26)

Tangent Component of the Impulse

The tangent component of the impulse, j_T , can be approximated by assuming that the contribution from the angular velocity normal component is minimal and can be ignored and isolating the tangent components from Equations (21) and (22) by performing a dot product with the \hat{T} vector to both sides of the equations:

$$\bullet \quad \vec{V}_{AP2} \cdot \hat{T} = \left\{ \left(\vec{V}_{A1} + \frac{j_N}{m_A} \hat{N} + \frac{j_T}{m_A} \hat{T} \right) + \left(\vec{\omega}_{A1} + \frac{j_T}{I_A} (\vec{R}_{AP} \times \hat{T}) \right) \times \vec{R}_{AP} \right\} \cdot \hat{T} \quad 2596$$

$$\bullet \quad \vec{V}_{BP2} \cdot \hat{T} = \left\{ \left(\vec{V}_{B1} - \frac{j_N}{m_B} \hat{N} - \frac{j_T}{m_B} \hat{T} \right) + \left(\vec{\omega}_{B1} - \frac{j_T}{I_B} (\vec{R}_{BP} \times \hat{T}) \right) \times \vec{R}_{BP} \right\} \cdot \hat{T} \quad 2597$$

Now follow the exact algebraic manipulation steps as when working with the normal component, the impulse in the tangent direction, j_T , can be derived and expressed as follows:

$$\bullet \quad j_T = \frac{(f-1)(\vec{V}_{AB1} \cdot \hat{T})}{\frac{1}{m_A} + \frac{1}{m_B} + \frac{\|\vec{R}_{AP} \times \hat{T}\|^2}{I_A} + \frac{\|\vec{R}_{BP} \times \hat{T}\|^2}{I_B}} \quad 2601$$
(27)

The Collision Angular Resolution Project

This project will guide you through the implementation of general collision impulse response that supports rotation. You can see an example of this project running in Figure 9-30. The source code to this project is defined in chapter9/9.8.collision_angular_resolution.

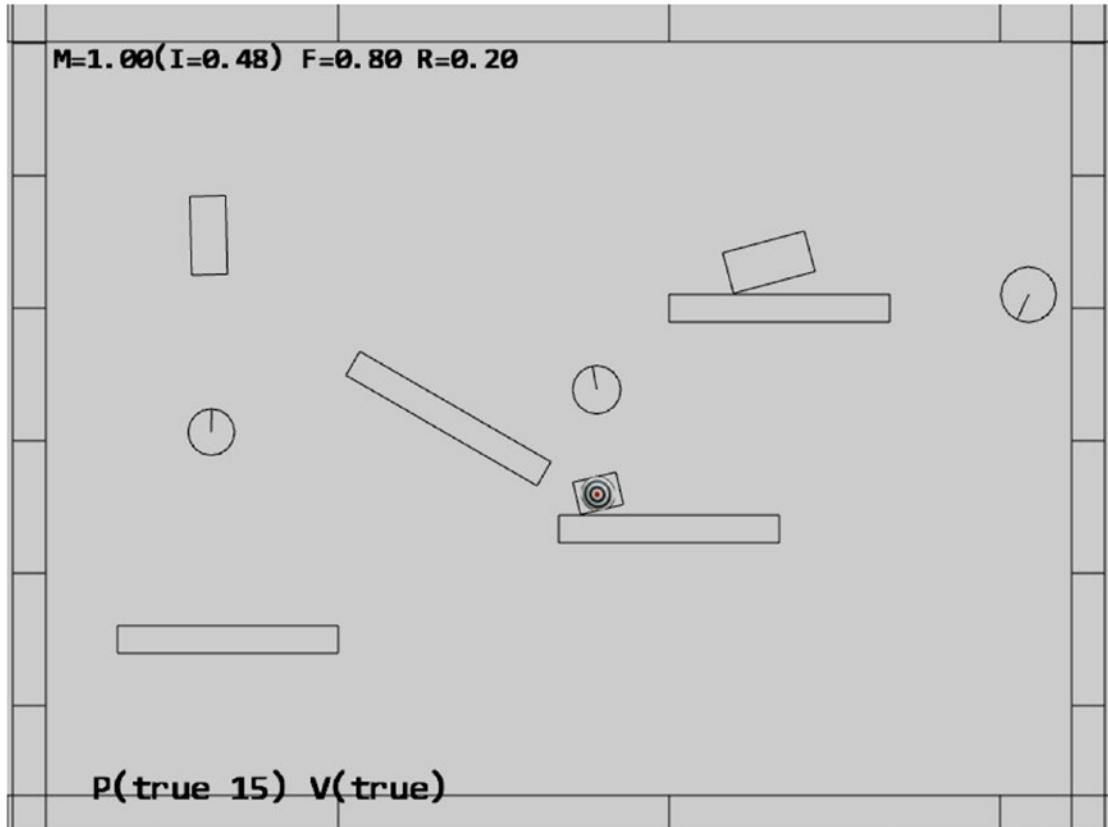


Figure 9-30. Running the Collision Angular Resolution project

The controls of the project are identical to the previous project:

- **Behavior control:**
 - **P key:** Toggles penetration resolution for all objects
 - **V key:** Toggles motion of all objects

- **H key:** Injects random velocity to all objects 2611
- **G key:** Randomly creates a new rigid circle or rectangle 2612
- **Draw control:** 2613
 - **C key:** Toggles the drawing of all CollisionInfo 2614
 - **T key:** Toggles textures on all objects 2615
 - **R key:** Toggles the drawing of RigidBody 2616
 - **B key:** Toggles the drawing of the bound on each RigidBody 2617
- **Object control:** 2618
 - **Left-/right-arrow key:** Sequences through and selects an object. 2619
 - **WASD keys:** Move the selected object. 2620
 - **Z/X key:** Rotates the selected object. 2621
 - **Y/U key:** Increases/decreases RigidBody size of the selected object; this does not change the size of the corresponding Renderable object. 2622
2623
2624
 - **Up-/down-arrow key + M/N/F:** Increase/decrease the mass/restitution/friction of the selected object. 2625
2626

The goals of the project are as follows:

- To understand the details of angular impulse 2628
- To integrate rotation into your collision resolution 2629
- To complete the physics component 2630

Note The cross product between a linear velocity on the x-y plane, $\vec{V} = (x, y, 0)$, and an angular velocity along the z axis, $\vec{\omega} = (0, 0, z)$, $\vec{V} \times \vec{\omega} = (-yz, xz, 0)$, is a vector on the x-y plane. 2631
2632
2633

2634 Updating the Physics Component

2635 To properly integrate angular impulse, you only need to replace the `resolveCollision()`
 2636 function in the `physics.js` file of the `src/engine/components` folder. While the
 2637 implementation closely follows the algebraic derivation steps, it is rather long
 2638 and involved. To facilitate understanding and for clarity, the following details the
 2639 implementation in steps:

```
2640 function resolveCollision(b, a, collisionInfo) {
 2641   let n = collisionInfo.getNormal();
 2642   // Step A: Compute relative velocity
 2643   ... implementation to follow ...
 2644   // Step B: Determine relative velocity in normal direction
 2645   ... implementation to follow ...
 2646   // Step C: Compute collision tangent direction
 2647   ... implementation to follow ...
 2648   // Step D: Determine the effective coefficients
 2649   ... implementation to follow ...
 2650   // Step E: Impulse in the normal and tangent directions
 2651   ... implementation to follow ...
 2652   // Step F: Update velocity in both normal and tangent directions
 2653   ... implementation to follow ...
 2654 }
```

- 2655 1. **Step A:** Compute relative velocity. As highlighted in Figure 9-29
 2656 and Equations (9) and (10), in the presence of angular velocity,
 2657 it is important to determine the collision position (Step A1) and
 2658 compute linear velocities $\vec{V}_{A\text{PI}}$ and $\vec{V}_{B\text{PI}}$ at the collision position
 2659 (Step A2).

```
2660 // Step A: Compute relative velocity
2661 let va = a.getVelocity();
2662 let vb = b.getVelocity();
```

```

// Step A1: Compute the intersection position p           2663
// the direction of collisionInfo is always from b to a   2664
// but the Mass is inverse, so start scale with a and end scale with b   2665
let invSum = 1 / (b.getInvMass() + a.getInvMass());          2666
let start = [0, 0], end = [0, 0], p = [0, 0];                2667
vec2.scale(start, collisionInfo.getStart(), a.getInvMass() * invSum); 2668
vec2.scale(end, collisionInfo.getEnd(), b.getInvMass() * invSum);    2669
vec2.add(p, start, end);                                     2670

// Step A2: Compute relative velocity with rotation components 2671
//     Vectors from center to P                            2672
//     r is vector from center of object to collision point 2673
let rBP = [0, 0], rAP = [0, 0];                          2674
vec2.subtract(rAP, p, a.getCenter());                    2675
vec2.subtract(rBP, p, b.getCenter());                    2676

// newV = V + mAngularVelocity cross R                 2677
let vAP1 = [-1 * a.getAngularVelocity() * rAP[1],        2678
            a.getAngularVelocity() * rAP[0]];             2679
vec2.add(vAP1, vAP1, va);                           2680

let vBP1 = [-1 * b.getAngularVelocity() * rBP[1],        2681
            b.getAngularVelocity() * rBP[0]];             2682
vec2.add(vBP1, vBP1, vb);                           2683

let relativeVelocity = [0, 0];                         2684
vec2.subtract(relativeVelocity, vAP1, vBP1);          2685

    2. Step B: Determine relative velocity in the normal direction. A      2686
        positive normal direction component signifies that the objects are      2687
        moving apart and the collision is resolved.                         2688

// Step B: Determine relative velocity in normal direction 2689
let rVelocityInNormal = vec2.dot(relativeVelocity, n);    2690

// if objects moving apart ignore
if (rVelocityInNormal > 0) {                                2691
    return;                                              2692
}                                                       2693

```

CHAPTER 9 SIMULATING THE WORLD WITH RIGIDSHAPES

2695 3. **Step C:** Compute the collision tangent direction and the tangent
 2696 direction component of the relative velocity.

```
2697 // Step C: Compute collision tangent direction
2698 let tangent = [0, 0];
2699 vec2.scale(tangent, n, rVelocityInNormal);
2700 vec2.subtract(tangent, tangent, relativeVelocity);
2701 vec2.normalize(tangent, tangent);
2702 // Relative velocity in tangent direction
2703 let rVelocityInTangent = vec2.dot(relativeVelocity, tangent);
```

2704 4. **Step D:** Determine the effective coefficients by using the average
 2705 of the colliding objects. As in the previous project, for consistency,
 2706 friction coefficient is one minus the values from the RigidShape
 2707 objects.

```
2708 // Step D: Determine the effective coefficients
2709 let newRestituation = (a.getRestitution() + b.getRestitution()) * 0.5;
2710 let newFriction = 1 - ((a.getFriction() + b.getFriction()) * 0.5);
```

2711 5. **Step E:** Impulse in the normal and tangent directions, these are
 2712 computed by following Equations (26) and (27) exactly.

```
2713 // Step E: Impulse in the normal and tangent directions
2714 // R cross N
2715 let rBPCrossN = rBP[0] * n[1] - rBP[1] * n[0]; // rBP cross n
2716 let rAPcrossN = rAP[0] * n[1] - rAP[1] * n[0]; // rAP cross n
2717 // Calc impulse scalar, formula of jN
2718 // can be found in http://www.myphysicslab.com/collision.html
2719 let jN = -(1 + newRestituation) * rVelocityInNormal;
2720 jN = jN / (b.getInvMass() + a.getInvMass() +
2721           rBPCrossN * rBPCrossN * b.getInertia() +
2722           rAPcrossN * rAPcrossN * a.getInertia());
2723 let rBPCrossT = rBP[0] * tangent[1] - rBP[1] * tangent[0];
2724 let rAPcrossT = rAP[0] * tangent[1] - rAP[1] * tangent[0];
2725 let jT = (newFriction - 1) * rVelocityInTangent;
```

```
jT = jT / (b.getInvMass() + a.getInvMass() +
    rBPCrossT * rBPCrossT * b.getInertia() +
    rAPCrossT * rAPCrossT * a.getInertia());
```

6. **Step F:** Update linear and angular velocities. These updates follow Equations (5), (6), (19), and (20) exactly.

```
// Update linear and angular velocities
vec2.scaleAndAdd(va, va, n, (jN * a.getInvMass()));
vec2.scaleAndAdd(va, va, tangent, (jT * a.getInvMass()));
setAngularVelocityDelta((rAPCrossN * jN * a.getInertia() +
    rAPCrossT * jT * a.getInertia()));

vec2.scaleAndAdd(vb, vb, n, -(jN * b.getInvMass()));
vec2.scaleAndAdd(vb, vb, tangent, -(jT * b.getInvMass()));
b.setAngularVelocityDelta(-(rBPCrossN * jN * b.getInertia() +
    rBPCrossT * jT * b.getInertia()));
```

Observations

Run the project to test your implementation. The shapes that you insert into the scene now rotate, collide, and respond in fashions that are similar to the real world. A circle shape rolls around when other shapes collide with them, while a rectangle shape should rotate naturally upon collision. The interpenetration between shapes should not be visible under normal circumstances. However, two situations can still cause observable interpenetrations: first, a small relaxation iteration, or second, your CPU is struggling with the number of shapes. In the first case, you can try increasing the relaxation iteration to prevent any interpenetration.

With the rotational support, you can now examine the effects of mass differences in collisions. With their abilities to roll, collisions between circles are the most straightforward to observe. Wait for all objects to be stationary and use the arrow key to select one of the created circles; type the M key with up arrow to increase its mass to a large value, for example, 20. Now select another object and use the WASD key to move and drop the selected object on the high-mass circle. Notice that the high-mass circle does not have much in response to the collision. For example, chances are a collision does not even cause the high-mass circle to roll. Now, type the H key to inject random velocities to all objects and observe the collisions. Notice that the collisions with the

2758 high-mass circle are almost like collisions with stationary walls/platforms. The inverted
2759 mass and rotational inertia modeled by the Impulse Method is capable of successfully
2760 capturing the collision effects of objects with different masses.

2761 Now your 2D physics engine implementation is completed. You can continue testing
2762 by creating additional shapes to observe when your CPU begins to struggle with keeping
2763 up real-time performance.

2764 Summary

2765 This chapter has guided you through understanding the foundation behind a working
2766 physics engine. The complicated physical interactions of objects in the real world
2767 are greatly simplified by focusing only on rigid body interactions or rigid shape
2768 simulations. The simulation process assumes that objects are continuous geometries
2769 with uniformly distributed mass where their shapes do not change during collisions. The
2770 computationally costly simulation is performed only on a selected subset of objects that
2771 are approximated by simple circles and rectangles.

2772 A step-by-step derivation of the relevant formulae for the simulations is followed
2773 by a detailed guide to the building of a functioning system. You have learned to extract
2774 collision information between shapes, formulate and compute shape collisions
2775 including the Separating Axis Theorem, approximate Newtonian motion integrals with
2776 the Symplectic Euler Integration, resolve interpenetrations of colliding objects based on
2777 numerically stable gradual relaxations, and derive and implement collision resolution
2778 based on the Impulse Method.

2779 Now that you have completed your physics engine, you can carefully examine
2780 the system and identify potentials for optimization and further abstractions. Many
2781 improvements to the physics engine are still possible. This is especially true from
2782 the perspective of supporting game developers with the newly defined and powerful
2783 functionality. For example, most physics engines also support straightforward collision
2784 detections without any responses. This is an important missing functionality from your
2785 physics component. While your engine is capable of simulating collisions results as is,
2786 the engine does not support responding to the simple, and computationally much lower
2787 cost, question of if objects have collided. As mentioned, this can be an excellent exercise.

2788 Though simple and missing some convenient interface functions, your physics
2789 component is functionally complete and capable of simulating rigid shape interactions
2790 with visually pleasant and realistic results. Your system supports intuitive parameters

including object mass, acceleration, velocity, restitution, and friction that can be related to behaviors of objects in the real world. Though computationally demanding, your system is capable of supporting a nontrivial number of rigid shape interactions. This is especially the case if the game genre only required one or a small set, for example, the hero and friendly characters, interacting with the rest of the objects, for example, the props, platforms, and enemies.

2791
2792
2793
2794
2795
2796

Game Design Considerations

2797

The puzzle level in the examples to this point has focused entirely on creating an understandable and consistent logical challenge; we've avoided burdening the exercise with any kind of visual design, narrative, or fictional setting (design elements traditionally associated with enhancing player presence) to ensure we're thinking only about the rules of play without introducing distractions. However, as you create core game mechanics, it's important to understand how certain elements of gameplay can contribute directly to presence; the logical rules and requirements of core game mechanics often have a limited effect on presence until they're paired with an interaction model, sound and visual design, and a setting. As discussed in Chapter 8, lighting is an example of a presence-enhancing visual design element that can also be used directly as a core game mechanic, and introducing physics to game world objects is similarly a presence-enhancing technique that's perhaps even more often directly connected to gameplay.

2798
2799
2800
2801
2802
2803
2804
2805
2806
2807
2808
2809
2810

Our experience in the real world is governed by physics, so it stands to reason that introducing similar behaviors in a game might be expected to enhance presence. An example of object physics enhancing presence but not necessarily contributing to design could be destructible environments that have no direct impact on gameplay: in a first-person shooter, for example, if the player shoots at crates and other game objects that respond by realistically exploding on impact, or if they throw a ball in the game world that bounces in a reasonable approximation of how a ball would bounce in the physical world, these are examples of physics being used purely to enhance presence but not necessarily contributing to gameplay. If a player is engaging with a game like *Angry Birds*, however, and launches one of the birds from their slingshot into the game space and they need to time the shot based on the physics-modeled parabolic arc the bird follows upon launch (as shown in Figure 9-31), this is an example of physics being used as both a core element of gameplay and also enhancing presence. In fact, any game that

2811
2812
2813
2814
2815
2816
2817
2818
2819
2820
2821
2822
2823

CHAPTER 9 SIMULATING THE WORLD WITH RIGIDSHAPES

2824 involves jumping a character or other game objects in an environment with simulated
2825 gravity is an example of physics contributing to both presence and the core mechanic,
2826 so many platformer games utilize physics as both a core mechanic and a presence-
2827 enhancing design element.

this figure will be printed in b/w

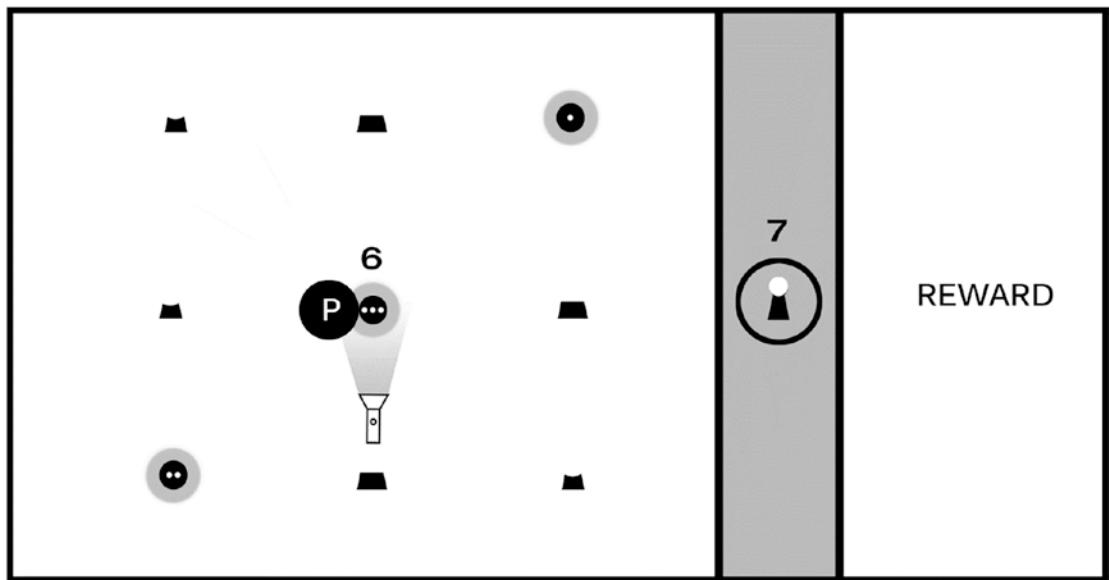


Figure 9-31. Rovio's Angry Birds requires players to launch projectiles from a slingshot in a virtual world that models gravity, mass, momentum, and object collision detection. The game physics are a fundamental component of the game mechanic and enhance the sense of presence by assigning physical world traits to virtual objects

2828 The projects in Chapter 9 introduce you to the powerful ability of physics to bring
2829 players into the game world. Instead of simply moving the hero character like a screen
2830 cursor, the player can now experience simulated inertia, momentum, and gravity
2831 requiring the same kind of predictive assessments around aiming, timing, and forward
2832 trajectory that would exist when manipulating objects in the physical world, and
2833 game objects are now capable of colliding in a manner familiar to our physical world
2834 experience. Even though specific values might take a detour from the real world in a
2835 simulated game space (e.g., lower or higher gravity, more or less inertia, and the like),
2836 as long as the relationships are consistent and reasonably analogous to our physical
2837 experience, presence will typically increase when these effects are added to game
2838 objects. Imagine, for example, a game level where the hero character was required to

push all the robots into a specific area within a specified time limit while avoiding being hit by projectiles. Imagine the same level without physics and it would of course be a very different experience.

We left the level design in Chapter 8 with an interesting two-stage mechanic focused almost exclusively on abstract logical rules and hadn't yet incorporated elements that would add presence to the experience and bring players into the game world. Recall the current state of the level in Figure 9-32.



this figure will be printed in b/w

Figure 9-32. The level as it currently stands includes a two-step puzzle first requiring players to move a flashlight and reveal hidden symbols; the player must then activate the shapes in the correct sequence to unlock the barrier and claim the reward

There is, of course, some sense of presence conveyed by the current level design: the barrier preventing players from accessing the reward is “impenetrable” and represented by a virtual wall, and the flashlight object is “shining” a virtual light beam that reveals hidden clues in the manner perhaps that a UV light in the real world might reveal special ink. Presence is frankly weak at this stage of development, however, as we have yet to place the game experience in a setting and the intentionally generic shapes don't provide much to help a player build their own internal narrative. Our current prototype uses a flashlight-like game object to reveal hidden symbols, but it's now possible to

2854 decouple the game mechanic's logical rules from the current implementation and
 2855 describe the core game mechanic as "the player must explore the environment to find
 2856 tools required to assemble a sequence in the correct order."

2857 For the next iteration of our game, let's revisit the interaction model and evolve
 2858 it from purely a logic puzzle to something a bit more active that makes use of object
 2859 physics. Figure 9-33 changes the game screen to include a jumping component.

this figure will be printed in b/w

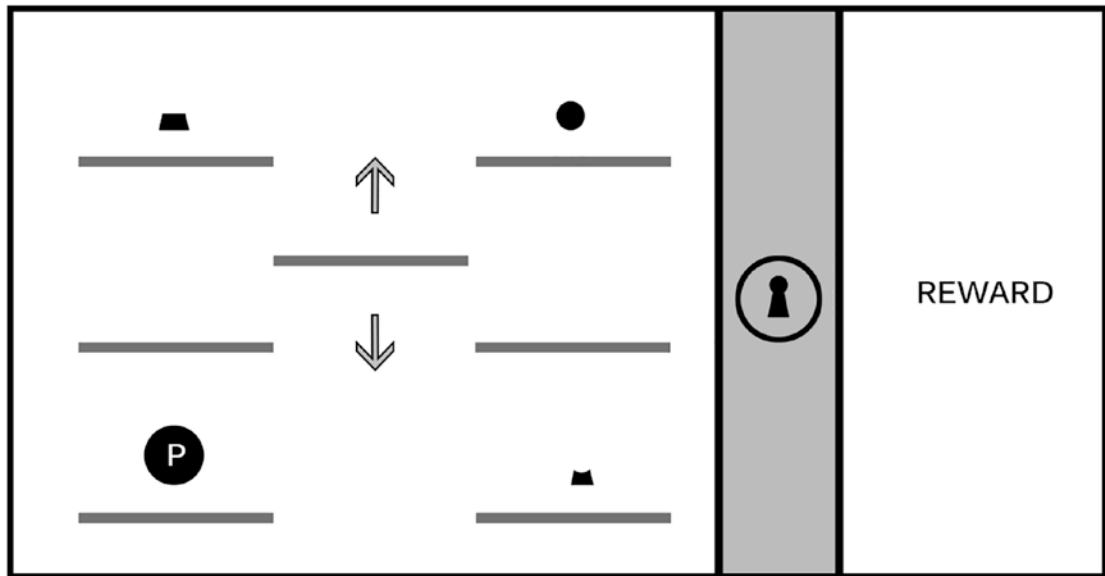
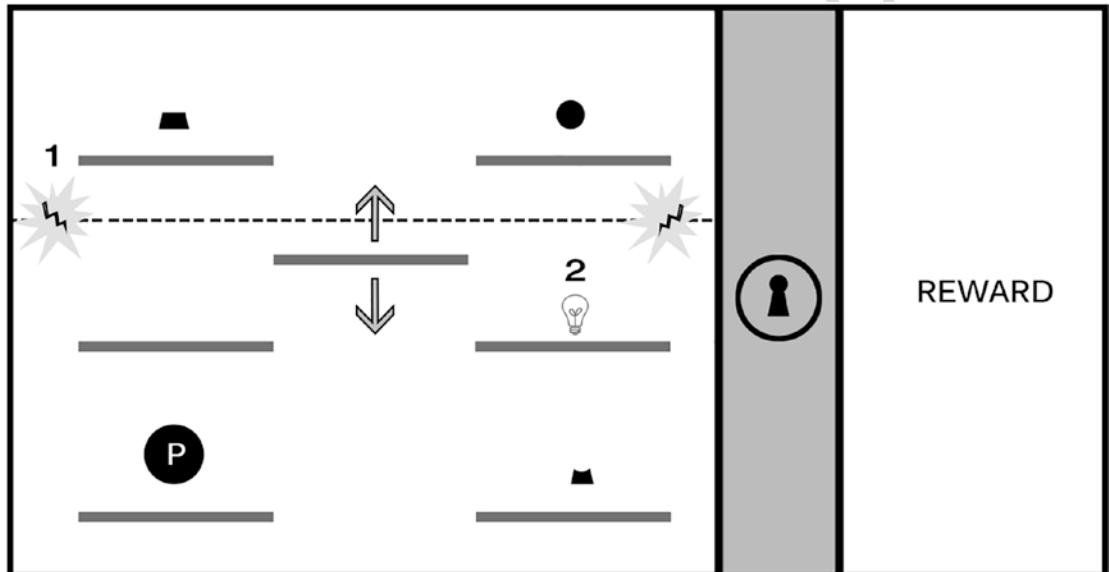


Figure 9-33. The game screen now shows just one instance of each part of the lock (top, middle, bottom), and the hero character moves in the manner of a traditional jumping 2D platformer. The six platforms on the left and right are stationary, and the middle platform moves up and down, allowing the player to ascend to higher levels. (This image assumes the player is able to "jump" the hero character between platforms on the same level but cannot reach higher levels without using the moving platform.)

2860 We're now evolving gameplay to include a dexterity challenge—in this case, timing
 2861 the jumps—yet it retains the same logical rules from the earlier iteration: the shapes
 2862 must be activated in the correct order to unlock the barrier blocking the reward. Imagine
 2863 the player experiences this screen for the first time; they'll begin exploring the screen
 2864 to learn the rules of engagement for the level, including the interaction model (the keys
 2865 and/or mouse buttons used to move and jump the hero character), whether missing

a jump results in a penalty (e.g., the loss of a “life” if the hero character misses a jump and falls off the game screen), and what it means to “activate” a shape and begin the sequence to unlock the barrier.

The game now has the beginning of an interesting (although still basic) platformer puzzle, but we’ve also now simplified the solution compared to our earlier iteration and the platformer jumping component isn’t especially challenging as shown in Figure 9-33. Recall how adding the flashlight in Chapter 8 increased the logical challenge of the original mechanic by adding a second kind of challenge requiring players to identify and use an object in the environment as a tool; we can add a similar second challenge to the platformer component, as shown in Figure 9-34.



this figure will be printed in b/w

Figure 9-34. The introduction of a force field blocking access to the upper platforms (#1) can significantly increase the challenge of the platformer component. In this design, the player must activate the switch (represented with a lightbulb in #2) to disable the force field and reach the first and third shapes

The introduction of a force field opens a variety of interesting possibilities to increase the challenge. The player must time the jump from the moving platform to the switch before hitting the force field, and the shapes must be activated in order (requiring the player to first activate top right, then the bottom right, and then the top left). Imagine a time limit is placed on the deactivation when the switch is flipped and that the puzzle will reset if all shapes aren’t activated before the force field is reengaged.

CHAPTER 9 SIMULATING THE WORLD WITH RIGIDSHAPES

2882 We've now taken an elemental mechanic based on a logical sequence and adapted it
2883 to support an action platformer experience. At this stage of development, the mechanic
2884 is becoming more interesting and beginning to feel more like a playable level, but it's
2885 still lacking setting and context; this is a good opportunity to explore the kind of story
2886 we might want to tell with this game. Are we interested in a sci-fi adventure, perhaps
2887 a survival horror experience, or maybe a series of puzzle levels with no connected
2888 narrative? The setting will not only help inform the visual identity of the game but
2889 can also guide decisions on the kinds of challenges we create for players (e.g., are
2890 "enemies" in the game working against the player, will the gameplay continue focusing
2891 on solving logic puzzles, or perhaps both?). A good exercise to practice connecting a
2892 game mechanic to a setting is to pick a place (e.g., the interior of a space ship) and begin
2893 exploring gameplay in that fictional space and defining the elements of the challenge in
2894 a way that make sense for the setting. For a game on a spaceship, perhaps, something
2895 has gone wrong and the player must make their way from one end of the ship to the
2896 other while neutralizing security lasers through the clever use of environment objects.
2897 Experiment with applying the spaceship setting to the current game mechanic and
2898 adjusting the elements in the level to fit that theme: lasers are just one option, but can
2899 you think of other uses of our game mechanic that don't involve an unlocking sequence?
2900 Try applying the game mechanic to a range of different environments to begin building
2901 your comfort for applying abstract gameplay to specific settings.

2902 Remember also that including object physics in level designs isn't always necessary
2903 to create a great game; sometimes you may want to subvert or completely ignore the laws
2904 of physics in the game worlds you create. The final quality of your game experience is the
2905 result of how effectively you harmonize and balance the nine elements of game design;
2906 it's not about the mandatory implementation of any one design option. Your game might
2907 be completely abstract and involve shapes and forms shifting in space in a way that has
2908 no bearing on the physical world, but your use of color, audio, and narrative might still
2909 combine to create an experience with a strong presence for players. However, if you find
2910 yourself with a game environment that seeks to convey a sense of physicality by making
2911 use of objects that people will associate with things found in the physical world, it's
2912 worth exploring how object physics might enhance the experience.

Author Queries

Chapter No.: 9 0005244102

Queries	Details Required	Author's Response
AU1	Please check if edit to sentence starting “Believable physical behaviors...” is okay.	
AU2	Please check if edit to sentence starting “Based on this set...” is okay.	
AU3	Please check if “Collision Info project” should be changed to “CollisionInfo project” for consistency.	
AU4	Please check if edit to sentence starting “For these reasons...” is okay.	
AU5	Please check if edit to sentence starting “In typical game engines...” is okay.	
AU6	Please check if edit to sentence starting “Notice that the new...” is okay.	
AU7	Please check phrase “tangent component is parallel to the of the wall” for completeness.	
AU8	Please check if edit to sentence starting “Substituting Equation (1)...” is okay.	
AU9	Please check if edit to sentence starting “Collecting terms and...” is okay.	
AU10	Please check if edit to sentence starting “Following the similar...” is okay.	
AU11	Please check if edit to sentence starting “However, two situations...” is okay.	

CHAPTER 10

1

Creating Effects with Particle Systems

2

3

After completing this chapter, you will be able to

- Understand the fundamentals of a particle, a particle emitter, and a particle system
- Appreciate that many interesting physical effects can be modeled based on a collection of dedicated particles
- Approximate the basic behavior of a particle such that the rendition of a collection of these particles resemble a simple explosion-like effect
- Implement a straightforward particle system that is integrated with the RigidShape system of the physics component

4

5

6

7

8

9

10

11

12

13

Introduction

14

So far in your game engine, it is assumed that the game world can be described by a collection of geometries where all objects are Renderable instances with texture, or animated sprite, and potentially illuminated by light sources. This game engine is powerful and capable of describing a significant portion of objects in the real world. However, it is also true that it can be challenging for your game engine to describe many everyday encounters, for example, sparks, fire, explosions, dirt, dust, etc. Many of these observations are transient effects resulting from matters changing physical states or a collection of very small-size entities reacting to physical disturbances. Collectively, these observations are often referred to as special effects and in general do not lend themselves well to being represented by fixed-shape geometries with textures.

15

16

17

18 AU1

20

21

22

23

24

639

25 Particle systems describe special effects by emitting a collection of particles with
26 properties that may include position, size, color, lifetime, and strategically selected
27 texture maps. These particles are defined with specific behaviors where once emitted,
28 their properties are updated to simulate a physical effect. For example, a fire particle may
29 be emitted to move in an upward direction with reddish color. As time progresses, the
30 particle may decrease in size, slow the upward motion, change its color toward yellow, and
31 eventually disappear after certain number of updates. With strategically designed update
32 functions, the rendition of a collection of such particles can resemble a fire burning.

33 In this chapter, you will study, design, and create a simple and flexible particle
34 system that includes the basic functionality required to achieve common effects, such as
35 explosions and magical spell effects. Additionally, you will implement a particle shader
36 to properly integrate your particles within your scenes. The particles will collide and
37 interact accordingly with the RigidShape objects. You will also discover the need for and
38 define particle emitters to generate particles over a period of time such as a campfire or
39 torch.

40 The main goal of this chapter is to understand the fundamentals of a particle
41 system: attributes and behaviors of simple particles, details of a particle emitter, and the
42 integration with the rest of the game engine. This chapter does not lead you to create
43 any specific types of special effects. This is analogous to learning an illumination model
44 in Chapter 8 without the details of creating any lighting effects. The manipulation of
45 light source parameters and material properties to create engaging lighting conditions
46 and the modeling of particle behaviors that resemble specific physical effects are the
47 responsibilities of the game developers. The basic responsibility of the game engine is
48 to define sufficient fundamental functionality to ensure that the game developers can
49 accomplish their job.

50 Particles and Particle Systems

51 A particle is a textured position without dimensions. This description may seem
52 contradictory because you have learned that a texture is an image and images are
53 always defined by a width and height and will definitely occupy an area. The important
54 clarification is that the game engine logic processes a particle as a position with no area,
55 while the drawing system displays the particle as a texture with proper dimensions.
56 In this way, even though an actual displayed area is shown, the width and height
57 dimensions of the texture are ignored by the underlying logic.

In addition to a position, a particle also has properties such as size (for scaling the texture), color (for tinting the texture), and life span. Similar to a typical game object, each particle is defined with behaviors that modify its properties during each update. It is the responsibility of this update function to ensure that the rendition of a collection of particles resembles a familiar physical effect. A particle system is the entity that controls the spawning, updating, and removal of each individual particle. In your game engine, particle systems will be defined as a separate component, just like the physics component.

In the following project, you will first learn about the support for drawing a particle object. After that, you will examine the details of how to create an actual particle object and define its behaviors. A particle is a new type of object for your game engine and requires the support of the entire drawing system, including custom GLSL shaders, default sharable shader instance, and a new Renderable pair.

The Particles Project

This project demonstrates how to implement a particle system to simulate explosion or spell-like effects. You can see an example of this project running in Figure 10-1. The source code of this project is located in the chapter10/10.1.particles folder.

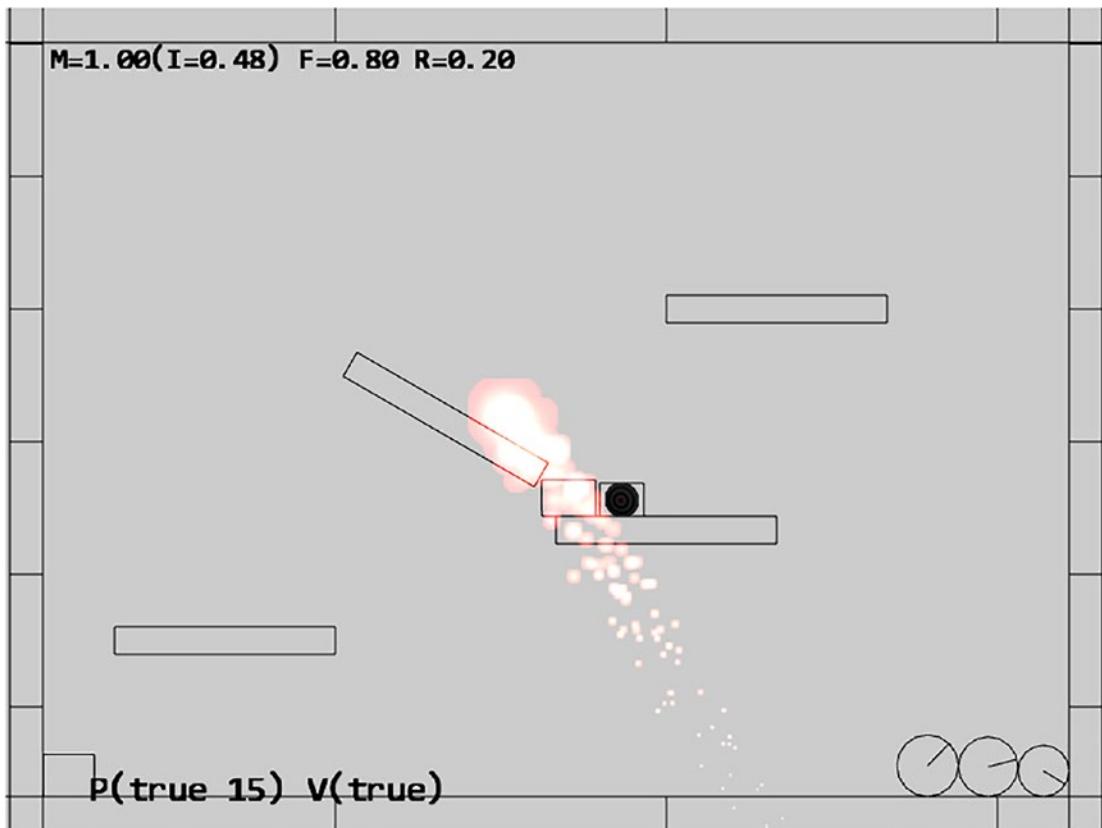


Figure 10-1. Running the Particles project

This project is a continuation from the previous chapter and supports all of the rigid shape and collision controls. For brevity, the details of those controls will not be restated in this chapter. The particle system-specific controls of the project are as follows:

- **Q key:** To spawn particles at the current mouse position
- **E key:** To toggle the drawing of particle bounds

The goals of the project are as follows:

- To understand the details of how to draw a particle and define its behavior
- To implement a simple particle system

You can find the following external resources in the assets folder: the fonts folder that contains the default system fonts, the particles folder that contains particle.png, the default particle texture, and the same four texture images from previous projects.

- minion_sprite.png defines the sprite elements for the hero and the minions. 88
- platform.png defines the platforms, floor, and ceiling tiles. 89
- wall.png defines the walls. 90
- target.png identifies the currently selected object. 91

Supporting Drawing of a Particle

Particles are textured positions with no area. However, as discussed in the introduction, your engine will draw each particle as a textured rectangle. For this reason, you can simply reuse the existing texture vertex shader texture_vs.glsl.

Creating GLSL Particle Fragment Shader

When it comes to the actual computation of each pixel color, a new GLSL fragment shader, particle_fs.glsl, must be created to ignore the global ambient terms. Physical effects such as fires and explosions do not participate in illumination computations.

1. Under the src/glsl_shaders folder, create a new file and name it particle_fs.glsl. 101
2. Similar to the texture fragment shader defined in texture_fs.glsl, you need to declare uPixelColor and vTexCoord to receive these values from the game engine and define the uSampler to sample the texture: 104

```
precision mediump float; 106
// sets the precision for floating point computation 107

// The object that fetches data from texture. 108
// Must be set outside the shader. 109
uniform sampler2D uSampler; 110
```

```

111 // Color of pixel
112 uniform vec4 uPixelColor;

113 // "varying" signifies that the texture coordinate will be
114 // interpolated and thus varies.
115 varying vec2 vTexCoord;

116     3. Now implement the main function to accumulate colors without
117         considering global ambient effect. This serves as one approach
118         for computing the colors of the particles. This function can be
119         modified to support different kinds of particle effects.

120 void main(void) {
121     // texel color look up based on interpolated UV value in vTexCoord
122     vec4 c = texture2D(uSampler, vec2(vTexCoord.s, vTexCoord.t));

123     vec3 r = vec3(c) * c.a * vec3(uPixelColor);
124     vec4 result = vec4(r, uPixelColor.a);

125     gl_FragColor = result;
126 }

```

127 Defining a Default ParticleShader Instance

128 You can now define a default particle shader instance to be shared. Recall from working
 129 with other types of shaders in the previous chapters that shaders are created once and
 130 shared engine wide in the `shader_resoruces.js` file in the `src/engine/core` folder.

131 1. Begin by editing the `shader_resources.js` file in the `src/`
 132 `engine/core` folder to define the constant, variable, and accessing
 133 function for the default particle shader:

```

134 // Particle Shader
135 let kParticleFS = "src/glsl_shaders/particle_fs.glsl";
136 let mParticleShader = null;
137 function getParticleShader() { return mParticleShader }

```

2. In the `init()` function, make sure to load the newly defined
`particle_fs` GLSL fragment shader:

```
function init() { 140
    let loadPromise = new Promise( 141
        async function(resolve) { 142
            await Promise.all([
                ... identical to previous code ... 144
                text.load(kShadowReceiverFS), 145
                text.load(kParticleFS) 146
            ]);
            resolve();
        }).then( 147
            function resolve() { createShaders(); } 148
        );
        map.pushPromise(loadPromise); 149
    } 150
} 151
```

3. With the new GLSL fragment shader, `particle_fs`, properly loaded, you can instantiate a new particle shader when the `createShaders()` function is called:

```
function createShaders() { 157
    ... identical to previous code ... 158
    mShadowReceiverShader = new SpriteShader(kTextureVS, 159
                                              kShadowReceiverFS); 160
    mParticleShader = new TextureShader(kTextureVS, kParticleFS); 161
} 162
```

163 4. In the `cleanUp()` function, remember to perform the proper
 164 cleanup and unload operations:

```
165   function cleanUp() {  

166     ... identical to previous code ...  

167     mShadowCasterShader.cleanUp();  

168     mParticleShader.cleanUp();  

169     ... identical to previous code ...  

170     text.unload(kShadowReceiverFS);  

171     text.unload(kParticleFS);  

172 }
```

173 5. Lastly, do not forget to export the newly defined function:

```
174   export {init, cleanUp,  

175     getConstColorShader, getTextureShader,  

176     getSpriteShader, getLineShader,  

177     getLightShader, getIllumShader,  

178     getShadowReceiverShader, getShadowCasterShader,  

179     getParticleShader}
```

180 Creating the ParticleRenderable Object

181 With the default particle shader class defined to interface to the GLSL `particle_fs`
 182 shader, you can now create a new Renderable object type to support the drawing of
 183 particles. Fortunately, the detailed behaviors of a particle, or a textured position, are
 184 identical to that of a `TextureRenderable` with the exception of the different shader. As
 185 such, the definition of the `ParticleRenderable` object is trivial.

186 In the `src/engine/renderables` folder, create the `particle_renderable.js`
 187 file; import from `defaultShaders` for accessing the particle shader and from
 188 `TextureRenderable` for the base class. Define the `ParticleRenderable` to be a subclass
 189 of `TextureRenderable`, and set the proper default shader in the constructor. Remember
 190 to export the class.

```

import * as defaultShaders from "../core/shader_resources.js";          191
import TextureRenderable from "./texture_renderable.js";                  192

class ParticleRenderable extends TextureRenderable {                      193
    constructor(myTexture) {                                              194
        super(myTexture);
        this._setShader(defaultShaders.getParticleShader());                195
    }
}

export default ParticleRenderable;                                         199

```

Loading the Default Particle Texture

For convenience when drawing, the game engine will preload the default particle texture, `particle.png`, located in the `assets/particles` folder. This operation can be integrated as part of the `defaultResources` initialization process.

1. Edit `default_resources.js` in the `src/engine/resources` folder, add an import from `texture.js` to access the texture loading functionality, and define a constant string for the location of the particle texture map and an accessor for this string:

```

import * as font from "./font.js";                                         208
import * as texture from "../resources/texture.js";                      209
import * as map from "../core/resource_map.js";                            210

// Default particle texture
let kDefaultPSTexture = "assets/particles/particle.png";               211

function getDefaultPSTexture() { return kDefaultPSTexture; }                212

```

2. In the `init()` function, call the `texture.load()` function to load the default particle texture map:

```

function init() {
    let loadPromise = new Promise(
        async function (resolve) {
            await Promise.all([

```

```
220         font.load(kDefaultFont),  
221         texture.load(kDefaultPSTexture)  
222     ]);  
223     resolve();  
224 })  
225 ... identical to previous code ...  
226 }
```

- 227 3. In the cleanUp() function, make sure to unload the default
228 texture:

```
229 function cleanUp() {  
230     font.unload(kDefaultFont);  
231     texture.unload(kDefaultPSTexture);  
232 }
```

- 233 4. Finally, remember to export the accessor:

```
234 export {  
235     ... identical to previous code ...  
236     getDefaultFontName, getDefaultPSTexture,  
237     ... identical to previous code ...  
238 }
```

239 With this integration, the default particle texture file will be loaded into the
240 resource_map during system initialization. This default texture map can be readily
241 accessed with the returned value from the getDefaultPSTexture() function.

242 Defining the Engine Particle Component

243 With the drawing infrastructure defined, you can now define the engine component to
244 manage the behavior of the particle system. For now, the only functionality required is to
245 include a default system acceleration for all particles.

In the `src/engine/components` folder, create the `particle_system.js` file, and define the variable, getter, and setter functions for the default particle system acceleration. Remember to export the newly defined functionality.

```
let mSystemAcceleration = [30, -50.0];  
function getSystemAcceleration() {  
    return vec2.clone(mSystemAcceleration); }  
function setSystemAcceleration(x, y) {  
    mSystemAcceleration[0] = x;  
    mSystemAcceleration[1] = y;  
}  
export {getSystemAcceleration, setSystemAcceleration}
```

Before continuing, make sure to update the engine access file, `index.js`, to allow game developer access to the newly defined functionality.

Defining the Particle and Particle Game Classes

You are now ready to define the actual particle, its default behaviors, and the class for a collection of particles.

Creating a Particle

Particles are lightweight game objects with simple properties wrapping around `ParticleRenderable` for drawing. To properly support motion, particles also implement movement approximation with the Symplectic Euler Integration.

1. Begin by creating the `particles` subfolder in the `src/engine` folder. This folder will contain particle-specific implementation files.
2. In the `src/engine/particles` folder, create `particle.js`, and define the constructor to include variables for position, velocity, acceleration, drag, and drawing parameters for debugging:

```
import * as loop from "../core/loop.js";  
import * as particleSystem from "../components/particle_system.js";  
import ParticleRenderable from "../renderables/particle_renderable.js";  
import * as debugDraw from "../core/debug_draw.js";
```

CHAPTER 10 CREATING EFFECTS WITH PARTICLE SYSTEMS

```
275 let kSizeFactor = 0.2;  
276  
277 class Particle {  
278     constructor(texture, x, y, life) {  
279         this.mRenderComponent = new ParticleRenderable(texture);  
280         this.setPosition(x, y);  
281  
282         // position control  
283         this.mVelocity = vec2.fromValues(0, 0);  
284         this.mAcceleration = particleSystem.getSystemAcceleration();  
285         this.mDrag = 0.95;  
286  
287         // Color control  
288         this.mDeltaColor = [0, 0, 0, 0];  
289  
290         // Size control  
291         this.mSizeDelta = 0;  
292  
293         // Life control  
294         this.mCyclesToLive = life;  
295     }  
296  
297     ... implementation to follow ...  
298 }  
299  
300 export default Particle;
```

3. Define the `draw()` function to draw the particle as a `TextureRenderable` and a `drawMarker()` debug function to draw an X marker at the position of the particle:

```
301 draw(aCamera) {  
302     this.mRenderComponent.draw(aCamera);  
303 }  
304  
305 drawMarker(aCamera) {  
306     let size = this.getSize();  
307     debugDraw.drawCrossMarker(aCamera, this.getPosition(),  
308                               size[0] * kSizeFactor, [0, 1, 0, 1]);  
309 }  
310
```

4. You can now implement the `update()` function to compute the position of the particle based on Symplectic Euler Integration, where the scaling with the `mDrag` variable simulates drags on the particles. Notice that this function also performs incremental changes to the other parameters including color and size. The `mCyclesToLive` variable informs the particle system when it is appropriate to remove this particle.

```

update() {
    this.mCyclesToLive--;

    let dt = loop.getUpdateIntervalInSeconds();

    // Symplectic Euler
    //   v += a * dt
    //   x += v * dt
    let p = this.getPosition();
    vec2.scaleAndAdd(this.mVelocity,
                      this.mVelocity, this.mAcceleration, dt);
    vec2.scale(this.mVelocity, this.mVelocity, this.mDrag);
    vec2.scaleAndAdd(p, p, this.mVelocity, dt);

    // update color
    let c = this.mRenderComponent.getColor();
    vec4.add(c, c, this.mDeltaColor);

    // update size
    let xf = this.mRenderComponent.getXform();
    let s = xf.getWidth() * this.mSizeDelta;
    xf.setSize(s, s);
}

```

5. Define simple get and set accessors. These functions are straightforward and are not listed here.

333 **Creating the ParticleSet**

334 To work with a collection of particles, you can now create the `ParticleSet` to support
 335 convenient looping over sets of `Particle`. For lightweight purposes, the `Particle`
 336 class does not subclass from the more complex `GameObject`; however, as JavaScript is
 337 an untyped language, it is still possible for `ParticleSet` to subclass from and refine
 338 `GameObjectSet` to take advantage of the existing set-specific functionality.

- 339 1. In the `src/engine/particles` folder, create `particle_set.js`,
 340 and define the `ParticleSet` to be a subclass of `GameObjectSet`:

```
341 import * as glSys from "../core/gl.js";
342 import GameObjectSet from "../game_objects/game_object_set.js";
343 class ParticleSet extends GameObjectSet {
344   constructor() {
345     super();
346   }
347   ... implementation to follow ...
348 }
349 export default ParticleSet;
```

- 350 2. Override the `draw()` function of `GameObjectSet` to ensure particles
 351 are drawn with additive blending:

352 **Note** Recall from Chapter 5 that the default `gl.blendFunc()` setting
 353 implements transparency by blending the alpha channel values. This is referred to
 354 as alpha blending. In this case, the `gl.blendFunc()` setting simply accumulates
 355 colors without considering the alpha channel. This is referred to as additive
 356 blending. Additive blending often results in oversaturation of pixel colors, that is,
 357 RGB components with values of greater than the maximum displayable value of
 358 1.0. The oversaturation of pixel color is often desirable when simulating intense
 359 brightness of fire and explosions.

```

draw(aCamera) {
    let gl = glSys.get();
    gl.blendFunc(gl.ONE, gl.ONE); // for additive blending!
    super.draw(aCamera);
    gl.blendFunc(gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA);
        // restore alpha blending
}
drawMarkers(aCamera) {
    let i;
    for (i = 0; i < this.mSet.length; i++) {
        this.mSet[i].drawMarker(aCamera);
    }
}
3. Override the update() function to ensure expired particles are
removed:

```

```

update() {
    super.update();
    // Cleanup Particles
    let i, obj;
    for (i = 0; i < this.size(); i++) {
        obj = this.getObjectAt(i);
        if (obj.hasExpired()) {
            this.removeFromSet(obj);
        }
    }
}

```

Lastly, remember to update the engine access file, `index.js`, to forward the newly defined functionality to the client.

Testing the Particle System

The test should verify two main goals. First, the implemented particle system is capable of generating visually pleasant effects. Second, the particles are handled correctly by being properly created, destroyed, and behaving as expected. The test case is based

CHAPTER 10 CREATING EFFECTS WITH PARTICLE SYSTEMS

```
392 mainly on the previous project with a new _createParticle() function that is called  
393 when the Q key is pressed. The _createParticle() function implemented in the  
394 my_game_main.js file creates particles with pseudo-random behaviors as listed in the  
395 following:  
  
396 function _createParticle(atX, atY) {  
397     let life = 30 + Math.random() * 200;  
398     let p = new engine.Particle(  
399         engine.defaultResources.getDefaultPSTexture(),  
400         atX, atY, life);  
401     p.setColor([1, 0, 0, 1]);  
  
402     // size of the particle  
403     let r = 5.5 + Math.random() * 0.5;  
404     p.setSize(r, r);  
  
405     // final color  
406     let fr = 3.5 + Math.random();  
407     let fg = 0.4 + 0.1 * Math.random();  
408     let fb = 0.3 + 0.1 * Math.random();  
409     p.setFinalColor([fr, fg, fb, 0.6]);  
  
410     // velocity on the particle  
411     let fx = 10 - 20 * Math.random();  
412     let fy = 10 * Math.random();  
413     p.setVelocity(fx, fy);  
  
414     // size delta  
415     p.setSizeDelta(0.98);  
  
416     return p;  
417 }
```

418 There are two important observations to be made on the _createParticle()
419 function. First, the random() function is used many times to configure each created
420 Particle. Particle systems utilize large numbers of similar particles with slight
421 differences to build and convey the desired visual effect. It is important to avoid any
422 patterns by using randomness. Second, there are many seemingly arbitrary numbers
423 used in the configuration, such as setting the life of the particle to be between 30

and 230 or setting the final red component to a number between 3.5 and 4.5. This is 424
unfortunately the nature of working with particle systems. There is often quite a bit of 425
ad hoc experimentation. Commercial game engines typically alleviate this difficulty 426
by releasing a collection of preset values for their particle systems. In this way, game 427
designers can fine-tune specific desired effects by adjusting the provided presets. 428

Observations

Run the project and press the Q key to observe the generated particles. It appears as 430
though there is combustion occurring underneath the mouse pointer. Hold the Q key 431
and move the mouse pointer around slowly to observe the combustion as though there 432
is an engine generating flames beneath the mouse. Type the E key to toggle the drawing 433
of individual particle positions. Now you can observe a green X marking the position of 434
each of the generated particles. 435

If you move the mouse pointer rapidly, you can observe individual pink circles with 436
green X centers changing color while dropping toward the floor. Although all particles 437
are created by the `_createParticle()` function and share the similar behaviors of 438
falling toward the floor while changing color, every particle appears slightly different and 439
does not exhibit any behavior patterns. You can now clearly observe the importance of 440
integrating randomness in the created particles. 441

There are limitless variations to how you can modify the `_createParticle()` 442
function. For example, you can change the explosion-like effect to steam or smoke 443
simply by changing the initial and final color to different shades of gray and 444
transparencies. Additionally, you can modify the default particle texture by inverting the 445
color to create black smoke effects. You could also modify the size change delta to be 446
greater than 1 to increase the size of the particles over time. There are literally no limits 447
to how particles can be created. The particle system you have implemented allows the 448
game developer to create particles with customized behaviors that are most suitable to 449
the game that they are building. 450

Lastly, notice that the generated particles do not interact with the `RigidShape` 451
objects and appears as though the particles are drawn over the rest of the objects in the 452
game scene. This issue will be examined and resolved in the next project. 453

454 Particle Collisions

455 An approach to integrate particles into a game scene is for the particles to follow the
456 implied rules of the scene and interact with the non-particle objects accordingly. The
457 ability to detect collisions is the foundation for interactions between objects. For this
458 reason, it is sometimes important to support particle collisions with the other, non-
459 particle game objects.

460 Since particles are defined only by their positions with no dimensions, the actual
461 collision computations can be relatively straightforward. However, there are typically
462 a large number of particles; as such, the number of collisions to be performed can also
463 be numerous. As a compromise and optimization in computational costs, particles
464 collisions can be based on RigidShape instead of the actual Renderable objects. This
465 is similar to the case of the physics component where the actual simulation is based
466 on simple rigid shapes in approximating the potentially geometrically complicated
467 Renderable objects.

468 The Particle Collisions Project

469 This project demonstrates how to implement a particle collision system that is capable
470 of resolving collisions between particles and the existing RigidShape objects. You can
471 see an example of this project running in Figure 10-2. The source code of this project is
472 located in the chapter10/10.2.particle_collisions folder.

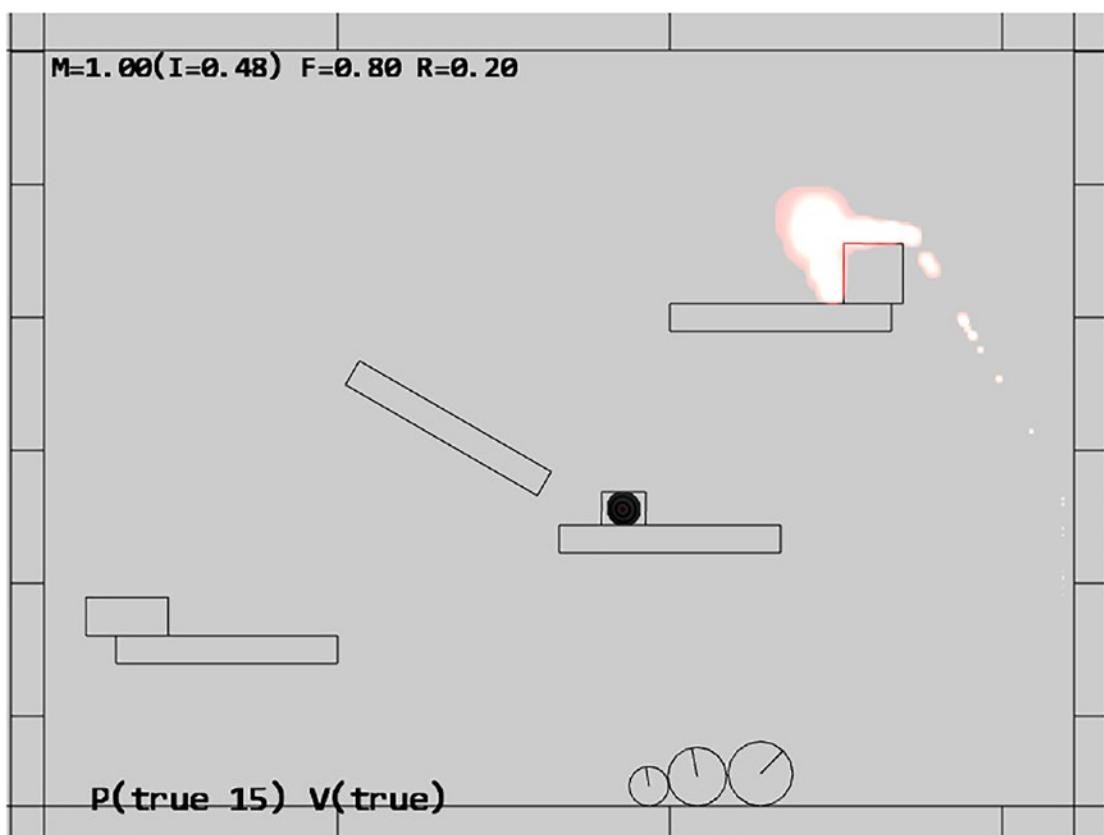


Figure 10-2. Running the Particle Collisions project

The controls of the project are identical to the previous project and support all of the rigid shape and collision controls. The controls that are specific to the particle system are as follows:

- **Q key:** To spawn particles at the current mouse position
- **E key:** To toggle the drawing of particle bounds
- **I key:** To toggle Particle/RigidShape collisions

The goals of the project are as follows:

- To understand and resolve collisions between individual positions and RigidShape objects
- To build a particle engine component that supports interaction with RigidShape

473
474
475

476
477
478

479
480
481
482
483

484 **Modifying the Particle System**

485 With a well-designed infrastructure, implementation of new functionality can be
 486 localized. In the case of particle collisions, all modifications are within the `particle_`
 487 `system.js` file in the `src/engine/components` folder.

- 488 1. Edit `particle_system.js` to define and initialize temporary
 489 local variables for resolving collisions with `RigidShape` objects.
 490 The `mCircleCollider` object will be used to represent individual
 491 particles in collisions.

```
492 import Transform from "../utils/transform.js";
493 import RigidCircle from "../rigid_shapes/rigid_circle.js";
494 import CollisionInfo from "../rigid_shapes/collision_info.js";
495 let mXform = null; // for collision with rigid shapes
496 let mCircleCollider = null;
497 let mCollisionInfo = null;
498 let mFrom1to2 = [0, 0];
499 function init() {
500     mXform = new Transform();
501     mCircleCollider = new RigidCircle(mXform, 1.0);
502     mCollisionInfo = new CollisionInfo();
503 }
```

- 504 2. Define the `resolveCirclePos()` function to resolve the collision
 505 between a `RigidCircle` and a position by pushing the position
 506 outside of the circle shape:

```
507 function resolveCirclePos(circShape, particle) {
508     let collision = false;
509     let pos = particle.getPosition();
510     let cPos = circShape.getCenter();
511     vec2.subtract(mFrom1to2, pos, cPos);
512     let dist = vec2.length(mFrom1to2);
513     if (dist < circShape.getRadius()) {
514         vec2.scale(mFrom1to2, mFrom1to2, 1/dist);
515         vec2.scaleAndAdd(pos, cPos, mFrom1to2, circShape.getRadius());
```

```

    collision = true;
516
}
517
return collision;
518
}
519

3. Define the resolveRectPos() function to resolve the collision
   between a RigidRectangle and a position by wrapping the
   mCircleCollider local variable around the position and invoking
   the RigidCircle to RigidRectangle collision function. When
   interpenetration is detected, the position is pushed outside of the
   rectangle shape according to the computed mCollisionInfo.
520
521
522
523
524
525

function resolveRectPos(rectShape, particle) {
526
    let collision = false;
527
    let s = particle.getSize();
528
    let p = particle.getPosition();
529
    mXform.setSize(s[0], s[1]); // referred by mCircleCollision
530
    mXform.setPosition(p[0], p[1]);
531
    if (mCircleCollider.boundTest(rectShape)) {
532
        if (rectShape.collisionTest(mCircleCollider, mCollisionInfo)) {
533
            // make sure info is always from rect towards particle
534
            vec2.subtract(mFrom1to2,
535
                          mCircleCollider.getCenter(), rectShape.getCenter());
536
            if (vec2.dot(mFrom1to2, mCollisionInfo.getNormal()) < 0)
537
                mCircleCollider.adjustPositionBy(
538
                  mCollisionInfo.getNormal(), -mCollisionInfo.getDepth());
539
            else
540
                mCircleCollider.adjustPositionBy(
541
                  mCollisionInfo.getNormal(), mCollisionInfo.getDepth());
542
            p = mXform.getPosition();
543
            particle.setPosition(p[0], p[1]);
544
            collision = true;
545
        }
546
    }
547
    return collision;
548
}
549

```

4. Implement `resolveRigidShapeCollision()` and `resolveRigidShapeSetCollision()` to allow convenient invocation by client game developers. These functions resolve collisions between a single or a set of `RigidShape` objects and a `ParticleSet` object.

obj: a `GameObject` (with potential `mRigidBody`)
Set: set of particles (`ParticleSet`)

```
function resolveRigidShapeCollision(obj, pSet) {  
    let i, j;  
    let collision = false;  
  
    let rigidShape = obj.getRigidBody();  
    for (j = 0; j < pSet.size(); j++) {  
        if (rigidShape.getType() == "RigidRectangle")  
            collision = resolveRectPos(rigidShape, pSet.getObjectAt(j));  
        else if (rigidShape.getType() == "RigidCircle")  
            collision = resolveCirclePos(rigidShape, pSet.getObjectAt(j));  
    }  
  
    return collision;  
}  
  
objSet: set of GameObjects (with potential mRigidBody)  
Set: set of particles (ParticleSet)
```

4. Implement `resolveRigidShapeSetCollision()` to allow convenient invocation by client game developers. These functions resolve collisions between a single or a set of `RigidShape` objects and a `ParticleSet` object.

```
function resolveRigidShapeSetCollision(objSet, pSet) {  
    let i, j;  
    let collision = false;  
    if ((objSet.size === 0) || (pSet.size === 0))  
        return false;  
    for (i=0; i<objSet.size(); i++) {  
        let rigidShape = objSet.getObjectAt(i).getRigidBody();  
        for (j = 0; j < pSet.size(); j++) {  
            if (rigidShape.getType() == "RigidRectangle")  
                collision = resolveRectPos(rigidShape,  
                                              pSet.getObjectAt(j)) || collision;  
            else if (rigidShape.getType() == "RigidCircle")  
                collision = resolveCirclePos(rigidShape,  
                                              pSet.getObjectAt(j)) || collision;  
        }  
    }  
    return collision;  
}
```

```

        collision = resolveCirclePos(rigidShape,
                                      pSet.getObjectAt(j)) || collision;
      }
    }
  return collision;
}

```

5. Lastly, remember to export the newly defined functions:

```

export {init,
        getSystemAcceleration, setSystemAcceleration,
        resolveRigidShapeCollision, resolveRigidShapeSetCollision}

```

Initializing the Particle System

The temporary variables defined in particle_system.js must be initialized before the game loop begins. Edit loop.js, import from particle_system.js, and call the init() function after asynchronous loading is completed in the start() function.

```

... identical to previous code ...

import * as debugDraw from "./debug_draw.js";
import * as particleSystem from "../components/particle_system.js";

... identical to previous code ...

async function start(scene) {
  ... identical to previous code ...

  // Wait for any async requests before game-load
  await map.waitOnPromises();

  // system init that can only occur after all resources are loaded
  particleSystem.init();

  ... identical to previous code ...
}

```

609 Testing the Particle System

610 The modifications required for the MyGame class are straightforward. A new variable must
 611 be defined to support the toggling of collision resolution, and the update() function
 612 defined in my_game_main.js is modified as follows:

```
613 update() {
  614     ... identical to previous code ...
  615     if (engine.input.isKeyPressed(engine.input.keys.One))
  616         this.mPSCollision = !this.mPSCollision;
  617     if (this.mPSCollision) {
  618         engine.particleSystem.resolveRigidShapeSetCollision(
  619                         this.mAllObjs, this.mParticles);
  620         engine.particleSystem.resolveRigidShapeSetCollision(
  621                         this.mPlatforms, this.mParticles);
  622     }
  623     ... identical to previous code ...
  624 }
```

625 Observations

626 As in previous projects, you can run the project and create particles with the Q and
 627 E keys. However, notice that the generated particles do not overlap with any of the
 628 objects. You can even try moving your mouse pointer to within the bounds of one of the
 629 RigidBody objects and then type the Q key. Notice that in all cases, the particles are
 630 generated outside of the shapes.

631 You can try typing the 1 key to toggle collisions with the rigid shapes. Note that with
 632 collisions enabled, the particles somewhat resemble the amber particles from a fire or an
 633 explosion where they bounce off the surfaces of RigidBody objects in the scene. When
 634 collision is toggled off, as you have observed from the previous project, the particles
 635 appear to be burning or exploding in front of the other objects. In this way, collision is
 636 simply another parameter for controlling the integration of the particle system with the
 637 rest of the game engine.

You may find it troublesome to continue to press the Q key to generate particles. In 638
the next project, you will learn about generation of particles over a fixed period of time. 639

Particle Emitters

640

With your current particle system implementation, you can create particles at a specific 641
point and time. These particles can move and change based on their properties. 642
However, particles can be created only when there is an explicit state change such as 643
a key click. This becomes restricting when it is desirable to persist the generation of 644
particles after the state change, such as an explosion or firework that persists for a short 645
while after the creation of a new RigidShape object. A particle emitter addresses this 646
issue by defining the functionality of generating particles over a time period. 647

The Particle Emitters Project

648

This project demonstrates how to implement a particle emitter for your particle system 649
to support particle emission over time. You can see an example of this project running in 650
Figure 10-3. The source code of this project is located in the chapter10/10.3.particle_ 651
emitters folder. 652

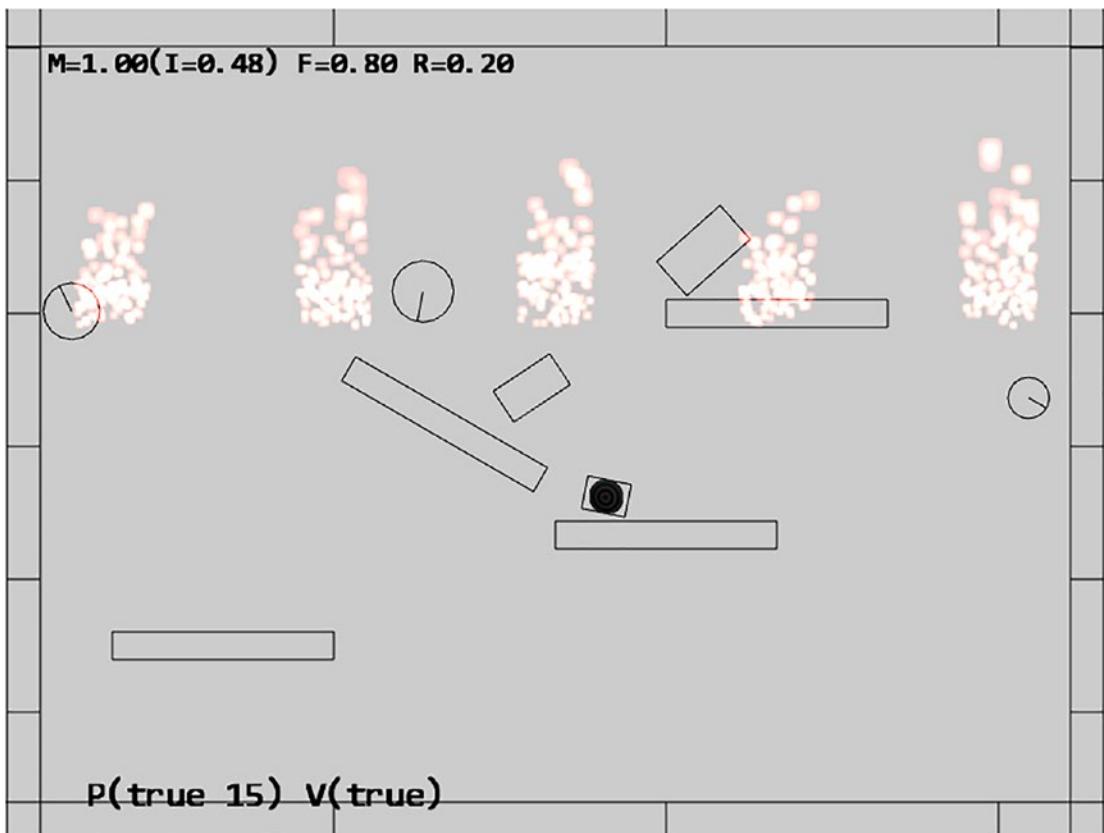


Figure 10-3. Running the Particle Emitters project

653 The controls of the project are identical to the previous project and support all of the
654 rigid shape and collision controls. The particle system-specific controls of the project are
655 as follows:

- 656 • **Q key:** To spawn particles at the current mouse position
657 • **E key:** To toggle the drawing of particle bounds
658 • **1 key:** To toggle Particle/RigidShape collisions

659 The goals of the project are as follows:

- 660 • To understand the need for particle emitters
661 • To experience implementing particle emitters

Defining the ParticleEmitter Class

662

You have observed and experienced the importance of avoiding patterns when working with particles. In this case, as the `ParticleEmitter` object generates new particles over time, once again, it is important to inject *randomness* to avoid any appearance of a pattern.

1. In the `src/engine/particles` folder, create `particle_emitter.js`; define the `ParticleEmitter` class with a constructor that receives the location, number, and how to emit new particles. Note that the `mParticleCreator` variable expects a callback function. When required, this function will be invoked to create a particle. 667
668
669
670
671

```
let kMinToEmit = 5; // Smallest number of particle emitted per cycle 672
class ParticleEmitter { 673
    constructor(px, py, num, createrFunc) { 674
        // Emitter position
        this.mEmitPosition = [px, py]; 675
        // Number of particles left to be emitted
        this.mNumRemains = num; 676
        // Function to create particles (user defined)
        this.mParticleCreator = createrFunc; 677
    } 680
    ... implementation to follow ...
} 683
export default ParticleEmitter; 684
```

2. Define a function to return the current status of the emitter. 685
When there are no more particles to emit, the emitters should be removed. 686
687

```
expired() { return (this.mNumRemains <= 0); } 688
```

689 3. Create a function to actually create or emit particles. Take
690 note of the randomness in the number of particles that are
691 actually emitted and the invocation of the `mParticleCreator()`
692 callback function. With this design, it is unlikely to encounter
693 patterns in the number of particles that are created over time.
694 In addition, the emitter defines only the mechanisms of how,
695 when, and where particles will be emitted and does not define the
696 characteristics of the created particles. The function pointed to by
697 `mParticleCreator` is responsible for defining the actual behavior
698 of each particle.

```
699 emitParticles(pSet) {  
700     let numToEmit = 0;  
701     if (this.mNumRemains < this.kMinToEmit) {  
702         // If only a few are left, emits all of them  
703         numToEmit = this.mNumRemains;  
704     } else {  
705         // Otherwise, emits about 20% of what's left  
706         numToEmit = Math.trunc(Math.random() * 0.2 * this.mNumRemains);  
707     }  
708     // Left for future emitting.  
709     this.mNumRemains -= numToEmit;  
710     let i, p;  
711     for (i = 0; i < numToEmit; i++) {  
712         p = this.mParticleCreator(  
713             this.mEmitPosition[0], this.mEmitPosition[1]);  
714         pSet.addToSet(p);  
715     }  
716 }
```

717 Lastly, remember to update the engine access file, `index.js`, to allow game
718 developer access to the `ParticleEmitter` class.

Modifying the Particle Set

The defined ParticleEmitter class needs to be integrated into ParticleSet to manage the emitted particles:

1. Edit particle_set.js in the src/engine/particles folder, and define a new variable for maintaining emitters:

```
constructor() {  
    super();  
    this.mEmitterSet = [];  
}
```

2. Define a function for instantiating a new emitter. Take note of the func parameter. This is the callback function that is responsible for the actual creation of individual Particle objects.

```
addEmitterAt(x, y, n, func) {  
    let e = new ParticleEmitter(x, y, n, func);  
    this.mEmitterSet.push(e);  
}
```

3. Modify the update function to loop through the emitter set to generate new particles and to remove expired emitters:

```
update() {  
    super.update();  
    // Cleanup Particles  
    let i, obj;  
    for (i = 0; i < this.size(); i++) {  
        obj = this.getObjectAt(i);  
        if (obj.hasExpired()) {  
            this.removeFromSet(obj);  
        }  
    }  
}
```

```

747 // Emit new particles
748 for (i = 0; i < this.mEmitterSet.length; i++) {
749     let e = this.mEmitterSet[i];
750     e.emitParticles(this);
751     if (e.expired()) { // delete the emitter when done
752         this.mEmitterSet.splice(i, 1);
753     }
754 }
755 }
```

756 Testing the Particle Emitter

757 This is a straightforward test of the correct functioning of the ParticleEmitter object.
 758 The MyGame class update() function is modified to create a new ParticleEmitter at
 759 the position of the RigidBody object when the G or H key is pressed. In this way, it will
 760 appear as though an explosion has occurred when a new RigidBody object is created or
 761 when RigidBody objects are assigned new velocities.

762 In both cases, the _createParticle() function discussed in the first project of this
 763 chapter is passed as the argument for the createFunc callback function parameter in
 764 the ParticleEmitter constructor.

765 Observations

766 Run the project and observe the initial firework-like explosions at the locations where
 767 the initial RigidBody objects are created. Type the G key to observe the accompanied
 768 explosion in the general vicinity of the newly created RigidBody object. Alternatively,
 769 you can type the H key to apply velocities to all the shapes and observe explosion-like
 770 effects next to each RigidBody object. For a very rough sense of what this particle
 771 system may look like in a game, you can try enabling texturing (with the T key), disabling
 772 RigidBody drawing (with the R key), and typing the H key to apply velocities. Observe
 773 that it appears as though the Renderable objects are being blasted by the explosions.

774 Notice how each explosion persists for a short while before disappearing gradually.
 775 Compare this effect with the one resulting from a short tapping of the Q key, and observe
 776 that without a dedicated particle emitter, the explosion seems to have fizzled before it
 777 begins.

Similar to particles, emitters can also have drastically different characteristics for simulating different physical effects. For example, the emitter you have implemented is driven by the number of particles to create. This behavior can be easily modified to use time as the driving factor, for example, emitting an approximated number of particles over a given time period. Other potential applications of emitters can include, but are not limited to

- Allowing the position of the emitter to change over time, for example, attaching the emitter to the end of a rocket
- Allowing emitter to affect the properties of the created particles, for example, changing the acceleration or velocity of all created particles to simulate wind effects

Based on the simple and yet flexible particle system you have implemented, you can now experiment with all these ideas in a straightforward manner.

Summary

There are three simple takeaways from this chapter. First, you have learned that particles, positions with an appropriate texture and no dimensions, can be useful in describing interesting physical effects. Second, the capability to collide and interact with other objects assists with the integration and placement of particles in game scenes. Lastly, in order to achieve the appearance of familiar physical effects, the emitting of particles should persist over some period of time.

You have developed a simple and yet flexible particle system to support the consistent management of individual particles and their emitters. Your system is simple because it consists of a single component, defined in `particle_system.js`, with only three simple supporting classes defined in the `src/engine/particles` folder. The system is flexible because of the callback mechanism for the actual creation of particles where the game developers are free to define and generate particles with any arbitrary behaviors.

The particle system you have built serves to demonstrate the fundamentals. To increase the sophistication of particle behaviors, you can subclass from the simple `Particle` class, define additional parameters, and amend the `update()` function accordingly. To support additional physical effects, you can consider modifying or subclassing from the `ParticleEmitter` class and emit particles according to your desired formulations.

810 Game Design Considerations

811 As discussed in Chapter 9, presence in games isn't exclusively achieved by recreating our
812 physical world experience in game environments; while introducing real-world physics
813 is often an effective way to bring players into virtual worlds, there are many other design
814 choices that can be quite effective at pulling players into the game, either in partnership
815 with object physics or on their own. For example, imagine a game with a 2D comic book
816 visual style that displays a "BOOM!" text-based image whenever something explodes;
817 objects don't show the word "BOOM!" when they explode in the physical world, of
818 course, but the stylized and familiar use of "BOOM!" in the context of a comic book
819 visual aesthetic can be quite effective on its own as a way to connect players with what's
820 happening in the game world.



AU2 **Figure 10-4.** Visual techniques like those shown in this graphic are often used in graphic novels to represent various fast-moving or high-impact actions like explosions, punches, crashes, and the like; similar visual techniques have also been used quite effectively in film and video games

821 Particle effects can also be used either in realistic ways that mimic how we'd expect
822 them to behave in the real world or in more creative ways that have no connection to
823 real-world physics. Try using what you've learned from the examples in this chapter and
824 experiment with adding particles to your current game prototype as we left it in Chapter 9:
825 can you think of some uses for particles in the current level that might support and
826 reinforce the presence of existing game elements (e.g., sparks flying if the player character
827 touches the force field)? What about introducing particle effects that might not directly
828 relate to gameplay but enhance and add interest to the game setting?

Author Queries

Chapter No.: 10 0005244103

Queries	Details Required	Author's Response
AU1	Please check if edit to sentence starting “However, it is also true...” is okay.	
AU2	Please provide citation to Figure 10-4.	

Uncorrected Proof

CHAPTER 11

1

Supporting Camera Background

2

3

After completing this chapter, you will be able to

- Implement background tiling with any image in any given camera WC bounds
- Understand parallax and simulate motion parallax with parallax scrolling
- Appreciate the need for layering objects in 2D games and support layered drawing

4

5

6

7

8

9

10

Introduction

11

By this point, your game engine is capable of illuminating 2D images to generate highlights and shadows and simulating basic physical behaviors. To conclude the engine development for this book, this chapter focuses on the general support for creating the game world environment with background tiling and parallax as well as relieving the game programmers from having to manage draw order.

12

13

14

15

16

Background images or objects are included to decorate the game world to further engage the players. This often requires the image being vast in scale with subtle visual complexities. For example, in a side-scrolling game, the background must always be present, and simple motion parallax can create the sense of depth and further capture the players' interests.

17

18

19

20

21

Tiling, in the context of computer graphics and video games, refers to the duplication of an image or pattern along the x and y directions. In video games, images used for tiling are usually strategically constructed to ensure content continuation across

22

23

24

the duplicating boundaries. Figure 11-1 shows an example of a strategically drawn background image tiled three times in the x direction and two times in the y direction. Notice the perfect continuation across the duplicating boundaries. Proper tiling conveys a sense of complexity in a boundless game world by creating only a single image.

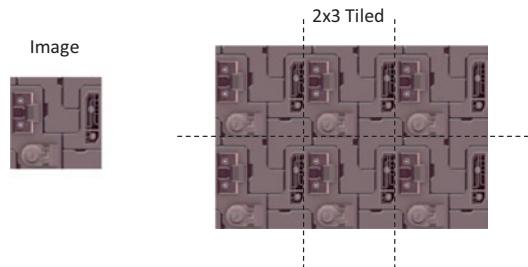


Figure 11-1. Tiling of a strategically drawn background image

Parallax is the apparent displacements of objects when they are viewed from different positions. Figure 11-2 shows an example of the parallax of a shaded circle. When viewed from the middle eye position, the center shaded circle appears to be covering the center rectangular block. However, this same shaded circle appears to be covering the top rectangular block when viewed from the bottom eye position. Motion parallax is the observation that when one is in motion, nearby objects appear to move quicker than those in the distance. This is a fundamental visual cue that informs depth perception. In 2D games, the simulation of motion parallax is a straightforward approach to introduce depth complexity to further captivate the players.

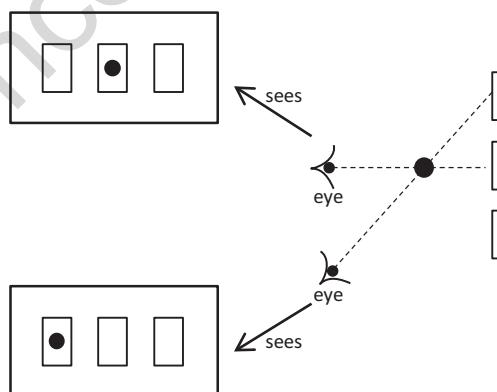


Figure 11-2. Parallax: objects appearing at different positions when observed from different viewpoints

This chapter presents a general algorithm for tiling the camera WC bounds and describes an abstraction for hiding the details of parallax scrolling. With the increase in visual complexity of the background, this chapter discusses the importance of and creates a layer manager to alleviate game programmers from the details of draw ordering.

Tiling of the Background

When tiling the background in a 2D game, it is important to recognize that only the tiles that cover the camera WC bounds need to be drawn. This is illustrated in Figure 11-3. In this example, the background object to be tiled is defined at the WC origin with its own width and height. However, in this case, the camera WC bounds do not intersect with the defined background object. Figure 11-3 shows that the background object needs to be tiled six times to cover the camera WC bounds. Notice that since it is not visible through the camera, the player-defined background object at the origin does not need to be drawn.

38
39
40
41
42
43
44
45
46
47
48
49
50
51

this figure will be printed in b/w

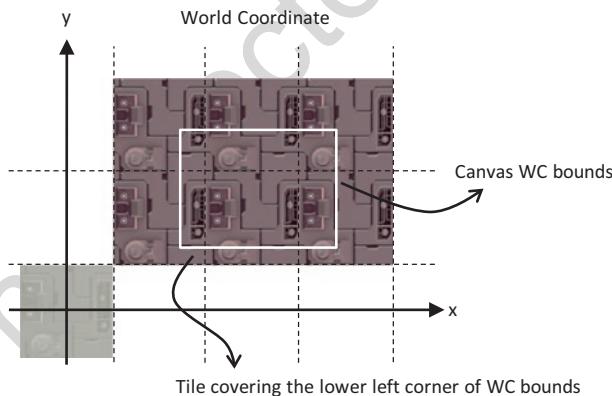


Figure 11-3. Generating tiled background for camera WC bounds

There are many ways to compute the required tiling for a given background object and the camera WC bounds. A simple approach is to determine the tile position that covers the lower-left corner of the WC bound and tile in the positive x and y directions.

52
53
54

The Tiled Objects Project

This project demonstrates how to implement simple background tiling. You can see an example of this project running in Figure 11-4. The source code to this project is defined in the chapter11/11.1.tiled_objects folder.

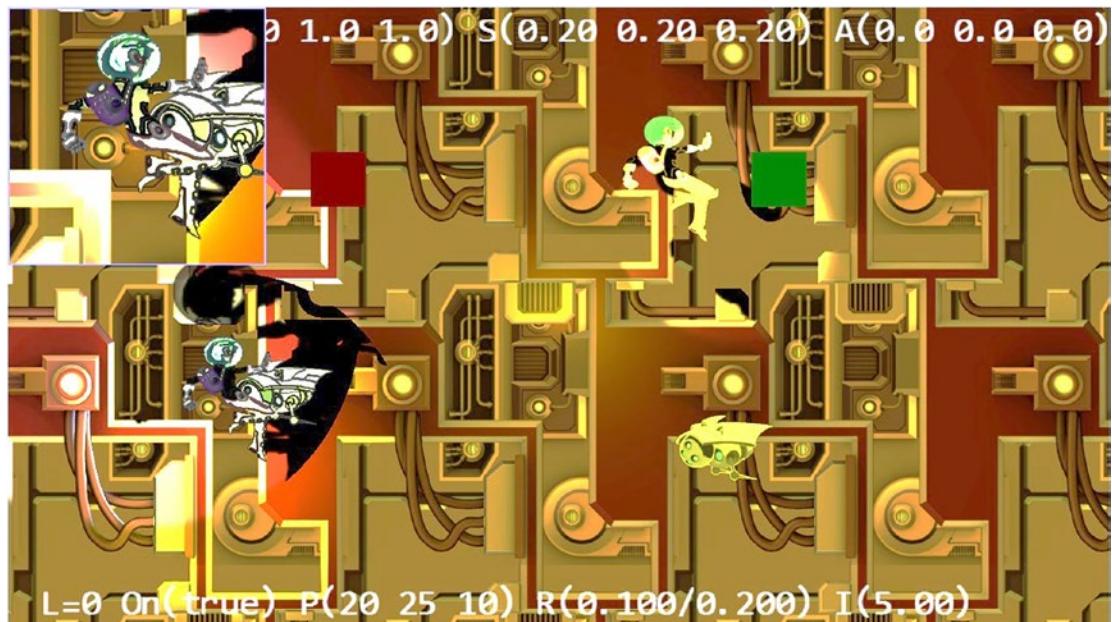


Figure 11-4. Running the Tiled Objects project

The control of the project is as follows:

- **WASD keys:** Move the Dye character (the hero) to pan the WC window bounds

The goals of the project are as follows:

- To experience working with multiple layers of background
- To implement the tiling of background objects for camera WC window bounds

You can find the following external resources in the assets folder. The fonts folder contains the default system fonts and six texture images: `minion_sprite.png`, `minion_sprite_normal.png`, `bg.png`, `bg_normal.png`, `bg_layer.png`, and `bg_layer_normal.png`.

The Hero and Minion objects are represented by sprite elements in the `minion_sprite.png` image, and `bg.png` and `bg_layer.png` are two layers of background images. The corresponding `_normal` files are the normal maps.

Define TiledGameObject

Recall that a `GameObject` abstracts the basic behavior of an object in the game where its appearance is determined by the `Renderable` object that it references. A `TiledGameObject` is a `GameObject` that is capable of tiling the referenced `Renderable` object to cover the WC bounds of a given `Camera` object.

1. Create a new file in the `src/engine/game_objects` folder and name it `tiled_game_object.js`. Add the following code to construct the object. The `mShouldTile` variable provides the option to stop the tiling process.

```
class TiledGameObject extends GameObject {
    constructor(renderableObj) {
        super(renderableObj);
        this.mShouldTile = true; // can switch this off if desired
    }
    ... implementation to follow ...
export default TiledGameObject;
```

2. Define the getter and setter functions for `mShouldTile`:

```
setIsTiled(t) { this.mShouldTile = t; }
shouldTile() { return this.mShouldTile; }
```

3. Define the function to tile and draw the `Renderable` object to cover the WC bounds of the `aCamera` object:

```
_drawTile(aCamera) {
    // Step A: Compute the positions and dimensions of tiling object.
    let xf = this.getXform();
    let w = xf.getWidth();
    let h = xf.getHeight();
    let pos = xf.getPosition();
    let left = pos[0] - (w / 2);
```

CHAPTER 11 SUPPORTING CAMERA BACKGROUND

```
100 let right = left + w;
101 let top = pos[1] + (h / 2);
102 let bottom = top - h;

103 // Step B: Get WC positions and dimensions of the drawing camera.
104 let wcPos = aCamera.getWCCenter();
105 let wcLeft = wcPos[0] - (aCamera.getWCWidth() / 2);
106 let wcRight = wcLeft + aCamera.getWCWidth();
107 let wcBottom = wcPos[1] - (aCamera.getWCHeight() / 2);
108 let wcTop = wcBottom + aCamera.getWCHeight();

109 // Step C: Determine offset to camera window's lower left corner.
110 let dx = 0, dy = 0; // offset to the lower left corner
111 // left/right boundary?
112 if (right < wcLeft) { // left of WC left
113     dx = Math.ceil((wcLeft - right) / w) * w;
114 } else {
115     if (left > wcLeft) { // not touching the left side
116         dx = -Math.ceil((left - wcLeft) / w) * w;
117     }
118 }
119 // top/bottom boundary
120 if (top < wcBottom) { // Lower than the WC bottom
121     dy = Math.ceil((wcBottom - top) / h) * h;
122 } else {
123     if (bottom > wcBottom) { // not touching the bottom
124         dy = -Math.ceil((bottom - wcBottom) / h) * h;
125     }
126 }

127 // Step D: Save the original position of the tiling object.
128 let sX = pos[0];
129 let sY = pos[1];

130 // Step E: Offset tiling object and update related position variables
131 xf.incXPosBy(dx);
132 xf.incYPosBy(dy);
```

```

right = pos[0] + (w / 2);                                133
top = pos[1] + (h / 2);                                  134

// Step F: Determine number of times to tile in x and y directions. 135
let nx = 1, ny = 1; // times to draw in the x and y directions 136
nx = Math.ceil((wcRight - right) / w);                  137
ny = Math.ceil((wcTop - top) / h);                      138

// Step G: Loop through each location to draw a tile           139
let cx = nx;                                              140
let xPos = pos[0];                                         141
while (ny >= 0) {                                         142
    cx = nx;
    pos[0] = xPos;                                         143
    while (cx >= 0) {                                     144
        this.mRenderComponent.draw(aCamera);               145
        xf.incXPosBy(w);                                 146
        --cx;                                            147
    }
    xf.incYPosBy(h);                                     148
    --ny;
}

// Step H: Reset the tiling object to its original position. 153
pos[0] = sX;                                              154
pos[1] = sY;                                              155
}

```

The `_drawTile()` function computes and repositions the Renderable object to cover the lower-left corner of the camera WC bounds and tiles the object in the positive x and y directions. Note the following:

- a. Steps A and B compute the position and dimension of the tiling object and the camera WC bounds.
- b. Step C computes the dx and dy offsets that will translate the Renderable object with bounds that cover the lower-left corner of the aCamera WC bounds. The calls to the

- 165 Math.ceil() function ensure that the computed dx and dy are
166 integer multiples of the Renderable width and height. This is
167 important to ensure there are no overlaps or gaps during tiling.
- 168 c. Step D saves the original position of the Renderable object
169 before offsetting and drawing it. Step E offsets the Renderable
170 object to cover the lower-left corner of the camera WC bounds.
- 171 d. Step F computes the number of repeats required, and step G
172 tiles the Renderable object in the positive x and y directions
173 until the results cover the entire camera WC bounds. The calls
174 to the Math.ceil() function ensure that the computed nx and
175 ny, the number of times to tile in the x and y directions, are
176 integers.
- 177 e. Step H resets the position of the tiled object to the original
178 location.
- 179 4. Override the draw() function to call the _drawTile() function
180 when tiling is enabled:

```
181 draw(aCamera) {  
182     if (this.isVisible() && (this.mDrawRenderable)) {  
183         if (this.shouldTile()) {  
184             // find out where we should be drawing  
185             this._drawTile(aCamera);  
186         } else {  
187             this.mRenderComponent.draw(aCamera);  
188         }  
189     }  
190 }
```

191 Lastly, remember to update the engine access file, index.js, to forward the newly
192 defined functionality to the client.

Modify MyGame to Test Tiled Objects

193

MyGame should test for the correctness of object tiling. To test multiple layers of tiling, two separate instances of `TiledGameObject` and `Camera` are created. The two `TiledGameObject` instances are located at different distances from the cameras (z depth) and are illuminated by different combinations of light sources. The added second camera focused on one of the Hero objects.

194
195
196
197
198

Only the creation of the `TiledGameObject` instance is of interest. This is because once created, a `TiledGameObject` instance can be handled in the same manner as a `GameObject` instance. For this reason, only the `init()` function of the `MyGame` class is examined in detail.

199
200
201
202

```
init() {
    // Step A: set up the cameras
    this.mCamera = new engine.Camera(
        vec2.fromValues(50, 37.5), // position of the camera
        100,                      // width of camera
        [0, 0, 1280, 720]         // viewport (X, Y, width, height)
    );
    this.mCamera.setBackgroundColor([0.8, 0.8, 0.8, 1]);
    // sets the background to gray

    this.mHeroCam = new engine.Camera(
        vec2.fromValues(20, 30.5), // position of the camera
        14,                      // width of camera
        [0, 420, 300, 300],      // viewport (X, Y, width, height)
        2
    );
    this.mHeroCam.setBackgroundColor([0.5, 0.5, 0.9, 1]);

    // Step B: the lights
    this._initializeLights(); // defined in MyGame_Lights.js

    // Step C: the far Background
    let bgR = new engine.IllumRenderable(this.kBg, this.kBgNormal);
    bgR.setElementPixelPositions(0, 1024, 0, 1024);
    bgR.getXform().setSize(30, 30);
    bgR.getXform().setPosition(0, 0);
```

```

226    bgR.getMaterial().setSpecular([0.2, 0.1, 0.1, 1]);
227    bgR.getMaterial().setShininess(50);
228    bgR.getXform().setZPos(-5);
229    bgR.addLight(this.mGlobalLightSet.getLightAt(1));
230                                // only the directional light
231    this.mBg = new engine.TiledGameObject(bgR);

232    // Step D: the closer Background
233    let i;
234    let bgR1 = new engine.IllumRenderable(
235                                this.kBgLayer, this.kBgLayerNormal);
236    bgR1.getXform().setSize(30, 30);
237    bgR1.getXform().setPosition(0, 0);
238    bgR1.getXform().setZPos(-2);
239    for (i = 0; i < 4; i++) {
240        bgR1.addLight(this.mGlobalLightSet.getLightAt(i)); // all lights
241    }
242    bgR1.getMaterial().setSpecular([0.2, 0.2, 0.5, 1]);
243    bgR1.getMaterial().setShininess(10);
244    this.mBgL1 = new engine.TiledGameObject(bgR1);

245    ... identical to previous code ...
246 }
```

247 In the listed code, the two cameras are first created in step A, followed by the
248 creation and initialization of all the light sources in the `_initializeLights()` function.
249 Step C defines `bgR` as a `TiledGameObject` with an `IllumRenderable` that is being
250 illuminated by one light source. Step D defines the second `TiledGameObject` based
251 on another `IllumRenderable` that is being illuminated by four light sources. Since
252 the `mShouldTile` variable of the `TileGameObject` class defaults to true, both of the tile
253 objects will tile the camera that they are drawing to.

254 Observations

255 You can now run the project and move the `Hero` object with the WASD keys. As expected,
256 the two layers of tiled backgrounds are clearly visible. You can switch off the illumination
257 to the farther background by selecting and turning off light source 1 (type the 1 key)

followed by the H key). Move the Hero object to pan the cameras to verify that the tiling and the background movement behaviors are correct in both of the cameras.

An interesting observation is that while the two layers of backgrounds are located at different distances from the camera, when the camera pans, the two background images scroll synchronously. If not for the differences in light source illumination, it would appear as though the background is actually a single image. This example illustrates the importance of simulating motion parallax.

Simulating Motion Parallax with Parallax Scrolling

Parallax scrolling simulates motion parallax by defining and scrolling objects at different speeds to convey the sense that these objects are located at different distances from the camera. Figure 11-5 illustrates this idea with a top view showing the conceptual distances of objects from the camera. Since this is a bird's-eye view, the width of the camera WC bounds is shown as a horizontal line at the bottom. The Hero object is the closest to the camera in front of two layers of backgrounds, Layer1 and Layer2. For typical 2D games, the vast majority of objects in the game will be located at this default distance from the camera. The background objects are located farther from the camera, behind the default distance. The distance perception can be conveyed by strategic drawings on the background objects (e.g., grass fields for Layer1 and distant mountains for Layer2) accompanied with appropriate scroll speeds. Take note that positions P_1 and P_2 on background objects Layer1 and Layer2 are directly behind the Hero object.

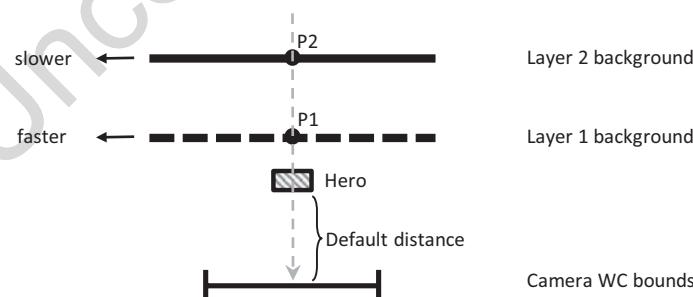


Figure 11-5. Top view of a scene with two background objects at different distances

278 Figure 11-6 shows the results of leftward parallax scrolling with a stationary
 279 camera. With Layer1 scrolling at a faster speed than Layer2, position P_1 has a greater
 280 displacement than P_2 from their original positions. A continuous scrolling will move
 281 Layer1 faster than Layer2 and properly convey the sense that it is closer than Layer2. In
 282 parallax scrolling, objects that are closer to the camera always have a greater scroll speed
 283 than objects that are farther.

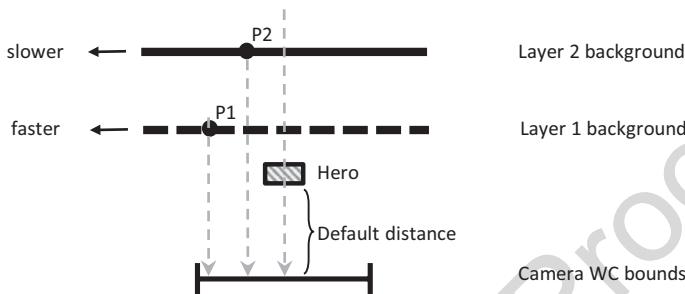


Figure 11-6. Top view of parallax scrolling with stationary camera

284 In the case when the camera is in motion, relative speeds of objects must be
 285 considered when implementing parallax scrolling. Figure 11-7 illustrates, with a top
 286 view, the situation of a moving camera with stationary objects. In this example, the
 287 camera WC bounds have moved rightward by d units. Since the movement is in the
 288 camera, all stationary objects in the camera view will appear to have been displaced
 289 by the inverse of the camera movement. For example, the stationary Hero object is
 290 displaced from the center leftward to the left edge of the new WC bounds. To properly
 291 simulate motion parallax, the two backgrounds, Layer1 and Layer2, must be displaced
 292 by different relative distances. In this case, relative distances must be computed such
 293 that farther objects will appear to move slower. At the end of the camera movement, in
 294 the new WC bounds, the Hero object that is closest to the camera will appear to have
 295 been displaced leftward by d units, the Layer1 object by $0.75d$, and the Layer2 object by
 296 $0.25d$. In this way, the displacements of the objects reflect their relative distances from
 297 the camera. To achieve this, the translation of the Hero object is zero, and the Layer1 and
 298 Layer2 objects must be translated rightward by $0.25d$ and $0.75d$, respectively. Notice
 299 that the backgrounds are translated rightward by amounts that are less than that of the
 300 camera movement, and as a result, the backgrounds are actually moving leftward. For
 301 example, although the Layer1 object is translated rightward by $0.25d$, when viewed from
 302 the camera that has been moved rightward by d , the resulting relative movement is such
 303 that the Layer1 object has been displaced leftward by $0.75d$.

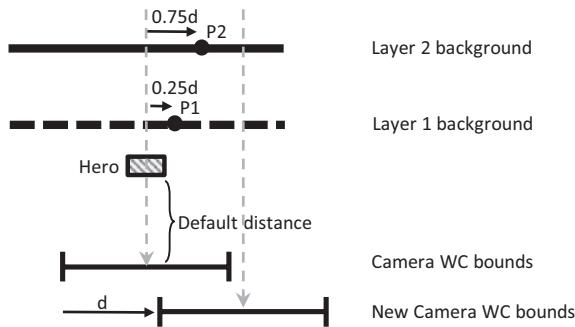


Figure 11-7. Top view of parallax scrolling with the camera in motion

It is important to note that in the described approach to implement parallax scrolling for a moving camera, stationary background objects are displaced. There are two limitations to this approach. First, the object locations are changed for the purpose of conveying visual cues and do not reflect any specific game state logic. This can create challenging conflicts if the game logic requires the precise control of the movements of the background objects. Fortunately, background objects are usually designed to serve the purposes of decorating the environment and engaging the players. Background objects typically do not participate in the actual gameplay logic. The second limitation is that the stationary background objects are actually in motion and will appear so when viewed from cameras other than the one causing the motion parallax. When views from multiple cameras are necessary in the presence of motion parallax, it is important to carefully coordinate them to avoid player confusion.

The Parallax Objects Project

This project demonstrates parallax scrolling. You can see an example of this project running in Figure 11-8. The source code to this project is defined in the chapter11/11.2.parallax_objects folder.



Figure 11-8. Running the Parallax Objects project

320 The controls of the project are as follows:

- 321 • **P key:** Toggles the drawing of a second camera that is not in motion
322 to highlight background object movements in simulating parallax
323 scrolling
- 324 • **WASD keys:** Move the Dye character (the hero) to pan the WC
325 window bounds

326 The goals of the project are as follows:

- 327 • To understand and appreciate motion parallax
- 328 • To simulate motion parallax with parallax scrolling

329 **Define ParallaxGameObject to Implement Parallax Scrolling**

330 Parallax scrolling involves the continuous scrolling of objects, and `TiledGameObject`
331 provides a convenient platform for never-ending scrolling. For this reason,
332 `ParallaxGameObject` is defined as a subclass of `TiledGameObject`.

1. Create `parallax_game_object.js` in the `src/engine/game_objects` folder, and add the following code to construct the object: 333

```

import TiledGameObject from "./tiled_game_object.js"; 335

class ParallaxGameObject extends TiledGameObject { 336
    constructor(renderableObj, scale, aCamera) { 337
        super(renderableObj);
        this.mRefCamera = aCamera; 339
        this.mCameraWCCenterRef = 340
            vec2.clone(this.mRefCamera.getWCCenter()); 341
        this.mParallaxScale = 1; 342
        this.setParallaxScale(scale); 343
    } 344
    ... implementation to follow ... 345
}
export default ParallaxGameObject; 346
  
```

The `ParallaxGameObject` object maintains `mRefCamera`, a reference to `aCamera` and `mCameraWCCenterRef`, the current WC bounds center. These values are used to compute relative movements based on the motion of the referenced camera to support parallax scrolling. The `scale` parameter is a positive value. A `scale` value of 1 represents that the object is located at the default distance, and values of less than 1 convey that the object is in front of the default distance. A `scale` of greater than 1 represents objects that are behind the default distance. The larger the `scale` value, the farther the object is from the camera. 348

2. Define the getter and setter functions for `mParallaxScale`. Notice 356
the clamping of negative values; this variable must be positive. 357

```

getParallaxScale() { return this.mParallaxScale; } 358
setParallaxScale(s) {
    this.mParallaxScale = s;
    if (s <= 0) { 359
        this.mParallaxScale = 1; 360
    }
} 363
  
```

365 3. Override the update() function to implement parallax scrolling:

```
366     update() {
367         // simple default behavior
368         this._refPosUpdate(); // check to see if the camera has moved
369         super.update();
370     }
```

371 The _refPosUpdate() function is the one that computes a relative
 372 displacement based on the reference camera's WC center position.

373 4. Define the _refPosUpdate() function:

```
374     _refPosUpdate() {
375         // now check for reference movement
376         let deltaT = vec2.fromValues(0, 0);
377         vec2.sub(deltaT,
378                 this.mCameraWCCenterRef, this.mRefCamera.getWCCenter());
379         this.setWCTranslationBy(deltaT);
380
381         // update WC center ref position
382         vec2.sub(this.mCameraWCCenterRef, this.mCameraWCCenterRef, deltaT);
383     }
```

383 The deltaT variable records the movement of the camera and
 384 setWCTranslationBy() moves the object to simulate parallax
 385 scrolling.

386 5. Define the function to translate the object to implement parallax
 387 scrolling. The negative delta is designed to move the object in
 388 the same direction as that of the camera. Notice the variable f is 1
 389 minus the inverse of mParallaxScale.

```
390     setWCTranslationBy(delta) {
391         let f = (1 - (1>this.mParallaxScale));
392         this.getXform().incXPosBy(-delta[0] * f);
393         this.getXform().incYPosBy(-delta[1] * f);
394     }
```

When `mParallaxScale` is less than 1, the inverse is greater than 1 and `f` becomes a negative number. In this case, when the camera moves, the object will move in the opposite direction and thus create the sensation that the object is in front of the default distance. 395
396
397

Conversely, when `mParallaxScale` is greater than 1, its inverse will be less than 1 and result in a positive `f` with a value of less than 1. In this case, the object will be moving in the same direction as the camera, only slower. A larger `mParallaxScale` would correspond to `f` value being closer to 1, and the movement of the object will be closer to that of the camera, or the object will appear to be at a further distance from the camera. 398
399
400
401
402

Lastly, remember to update the engine access file, `index.js`, to forward the newly defined functionality to the client. 403
404

Testing ParallaxGameObject in MyGame

The testing of `ParallaxGameObject` involves testing for the correctness of parallax scrolling while the camera is in motion with objects in front of and behind the default distance, all while observing the `ParallaxGameObject` from a separate stationary camera. The source code of the `MyGame` level is largely similar to that from the previous project, and the details are not listed. The relevant part of the `init()` function is listed for the purpose of illustrating how to create the `ParallaxGameObject` instances. 405
406
407
408
409
410
411

```
init() {
    // Step A: set up the cameras
    this.mCamera = new engine.Camera(
        vec2.fromValues(50, 37.5), // position of the camera
        100,                      // width of camera
        [0, 0, 1280, 720]         // viewport (orgX, orgY, width, height)
    );
    this.mCamera.setBackgroundColor([0.8, 0.8, 0.8, 1]);
    // sets the background to gray

    this.mParallaxCam = new engine.Camera(
        vec2.fromValues(40, 30), // position of the camera
        45,                      // width of camera
        [0, 420, 600, 300],      // viewport (orgX, orgY, width, height)
        2
    );
    this.mParallaxCam.setBackgroundColor([0.5, 0.5, 0.9, 1]);
}
```

CHAPTER 11 SUPPORTING CAMERA BACKGROUND

```
428 // Step B: the lights
429 this._initializeLights(); // defined in MyGame_Lights.js
430
431 // Step C: the far Background
432 let bgR = new engine.IllumRenderable(this.kBg, this.kBgNormal);
433 bgR.setElementPixelPositions(0, 1024, 0, 1024);
434 bgR.getXform().setSize(30, 30);
435 bgR.getXform().setPosition(0, 0);
436 bgR.getMaterial().setSpecular([0.2, 0.1, 0.1, 1]);
437 bgR.getMaterial().setShininess(50);
438 bgR.getXform().setZPos(-5);
439
440 // only the directional light
441 bgR.addLight(this.mGlobalLightSet.getLightAt(1));
442 this.mBg = new engine.ParallaxGameObject(bgR, 5, this.mCamera);
443
444 // Step D: the closer Background
445 let i;
446 let bgR1 = new engine.IllumRenderable(
447                                         this.kBgLayer, this.kBgLayerNormal);
448 bgR1.getXform().setSize(25, 25);
449 bgR1.getXform().setPosition(0, -15);
450 bgR1.getXform().setZPos(0);
451 // the directional light
452 bgR1.addLight(this.mGlobalLightSet.getLightAt(1));
453 // the hero spotlight light
454 bgR1.addLight(this.mGlobalLightSet.getLightAt(2));
455 // the hero spotlight light
456 bgR1.addLight(this.mGlobalLightSet.getLightAt(3));
457 bgR1.getMaterial().setSpecular([0.2, 0.2, 0.5, 1]);
458 bgR1.getMaterial().setShininess(10);
459 this.mBgL1 = new engine.ParallaxGameObject(bgR1, 3, this.mCamera);
460
461 // Step E: the front layer
462 let f = new engine.TextureRenderable(this.kBgLayer);
463 f.getXform().setSize(50, 50);
```

```

f.getXform().setPosition(-3, 2);                                460
this.mFront = new engine.ParallaxGameObject(f, 0.9, this.mCamera); 461
... identical to previous code ...                                462
}

```

463

The `mBg` object is created as a `ParallaxGameObject` with a scale of 5, `mBgL1` with a scale of 3, and `mFront` with a scale of 0.9. Recall that `scale` is the second parameter of the `ParallaxGameObject` constructor. This parameter signifies the object distance from the camera, with values greater than 1 being farther from and values less than 1 being closer to the default distance. In this case, `mBg` is the furthest away from the camera while `mBgL1` is closer. Regardless, both are still behind the default distance. The `mFront` object is the closest to the camera and in front of the default distance or in front of the `Hero` object.

Observations

You can now run the project and observe the darker foreground layer partially blocking the `Hero` and `Minion` objects. You can move the `Hero` object to pan the camera and observe the two background layers scrolling at different speeds. The `mBg` object is farther away and thus scrolls slower than the `mBgL1` object. You will also notice the front-layer parallax scrolls at a faster speed than all other objects, and as a result, panning the camera reveals different parts of the stationary `Minion` objects.

Press the `P` key to enable the drawing of the second camera. Notice that when the `Hero` is stationary, the view in this camera is as expected, not moving. Now, if you move the `Hero` object to pan the main camera, note the foreground and background objects in the second camera view are also moving and exhibit motion parallax even though the second camera is not moving! As game designers, it is important to ensure this side effect does not confuse the player.

Layer Management

Although the engine you are developing is for 2D games, you have worked with a few situations where depth ordering and drawing orders are important. For example, the shadow receiver must always be defined behind the shadow casters, and as discussed in the previous example, foreground and background parallax objects must be carefully

490 defined and drawn in the order of their depth ordering. It is convenient for the game
491 engine to provide a utility manager to help game programmers manage and work with
492 the depth layering. A typical 2D game can have the following layers, in the order of the
493 distance from the camera, from nearest to furthest:

- 494 • **Heads-up display (HUD) layer:** Typically, closest to the camera
495 displaying essential user interface information
- 496 • **Foreground or front layer:** The layer in front of the game objects for
497 decorative or partial occlusion of the game objects
- 498 • **Actor layer:** The default distance layer in Figure 11-5, where all game
499 objects reside
- 500 • **Shadow receiver layer:** The layer behind the actor layer to receive
501 potential shadows
- 502 • **Background layer:** The decorative background

503 Each layer will reference all objects defined for that layer, and these objects will be
504 drawn in the order they were inserted into the layer, with the last inserted drawn last and
505 covering objects before it. This section presents the Layer engine component to support
506 the described five layers to relieve game programmers from the details of managing
507 updates and drawings the objects. Note that the number of layers a game engine should
508 support is determined by the kinds of games that the engine is designed to build. The
509 five layers presented are logical and convenient for simple games. You may choose to
510 expand the number of layers in your own game engine.

511 The Layer Manager Project

512 This project demonstrates how to develop a utility component to assist in managing
513 layers for the game programmers. You can see an example of this project running in
514 Figure 11-9. The source code to this project is defined in the chapter11/11.3.layer_
515 manager folder.



Figure 11-9. Running the Layer Manager project

The controls of the project are identical to the previous project:

- **P key:** Toggles the drawing of a second camera that is not in motion to highlight background object movements in simulating parallax scrolling
- **WASD keys:** Move the Dye character (the hero) to pan the WC window bounds

The goals of the project are as follows:

- To appreciate the importance of layering in 2D games
- To develop a layer manager engine component

Layer Management in the Engine

Follow the pattern of defining an engine component, for example, similar to that of physics and particle systems:

1. Create a new file in the `src/engine/components` folder and name it `layer.js`. This file will implement the Layer engine component.

530 2. Define enumerators for the layers:

```
531 const eBackground = 0;  
532 const eShadowReceiver = 1;  
533 const eActors = 2;  
534 const eFront = 3;  
535 const eHUD = 4;
```

536 3. Define appropriate constants and instance variables to keep
537 track of the layers. The `mAllLayers` variable is an array of
538 `GameObjectSet` instances representing each of the five layers.

```
539 let kNumLayers = 5;  
540 let mAllLayers = [];
```

541 4. Define an `init()` function to create the array of `GameObjectSet`
542 instances:

```
543 function init() {  
544     mAllLayers[eBackground] = new GameObjectSet();  
545     mAllLayers[eShadowReceiver] = new GameObjectSet();  
546     mAllLayers[eActors] = new GameObjectSet();  
547     mAllLayers[eFront] = new GameObjectSet();  
548     mAllLayers[eHUD] = new GameObjectSet();  
549 }
```

550 5. Define a `cleanUp()` function to reset the `mAllLayer` array:

```
551 function cleanUp() {  
552     init();  
553 }
```

554 6. Define functions to add to, remove from, and query the layers.
555 Note the `addAsShadowCaster()` function assumes that the shadow
556 receiver objects are already inserted into the `eShadowReceiver`
557 layer and adds the casting object to all receivers in the layer.

```

function addToLayer(layerEnum, obj) { 558
    mAllLayers[layerEnum].addToSet(obj); }
function removeFromLayer(layerEnum, obj) { 559
    mAllLayers[layerEnum].removeFromSet(obj); }
function layerSize(layerEnum) { return mAllLayers[layerEnum].size(); } 560
561
function addAsShadowCaster(obj) { 562
    let i;
    for (i = 0; i < mAllLayers[eShadowReceiver].size(); i++) { 563
        mAllLayers[eShadowReceiver].get0bjectAt(i).addShadowCaster(obj);
    }
}

```

7. Define functions to draw a specific layer or all the layers, from the furthest to the nearest to the camera:

```

function drawLayer(layerEnum, aCamera) { 564
    mAllLayers[layerEnum].draw(aCamera); }
function drawAllLayers(aCamera) { 565
    let i;
    for (i = 0; i < kNumLayers; i++) { 566
        mAllLayers[i].draw(aCamera);
    }
}

```

8. Define a function to move a specific object such that it will be drawn last (on top):

```

function moveToLayerFront(layerEnum, obj) { 567
    mAllLayers[layerEnum].moveToLast(obj);
}

```

9. Define functions to update a specific layer or all the layers:

```

function updateLayer(layerEnum) { mAllLayers[layerEnum].update(); } 568
function updateAllLayers() { 569
    let i;
}

```

CHAPTER 11 SUPPORTING CAMERA BACKGROUND

```
588     for (i = 0; i < kNumLayers; i++) {  
589         mAllLayers[i].update();  
590     }  
591 }
```

592 10. Remember to export all the defined functionality:

```
593 export {  
594     // array indices  
595     eBackground, eShadowReceiver, eActors, eFront, eHUD,  
596     // init and cleanup  
597     init, cleanUp,  
598     // draw/update  
599     drawLayer, drawAllLayers,  
600     updateLayer, updateAllLayers,  
601     // layer-specific support  
602     addToLayer, addAsShadowCaster,  
603     removeFromLayer, moveToLayerFront,  
604     layerSize  
605 }
```

606 Lastly, remember to update the engine access file, `index.js`, to forward the newly
607 defined functionality to the client.

608 **Modify Engine Components and Objects**

609 You must modify the rest of the game engine slightly to integrate the new Layer
610 component.

611 **Enhance the GameObjectSet Functionality**

612 Add the following function to support moving an object to the end of a set array:

```
613 moveToLast(obj) {  
614     this.removeFromSet(obj);  
615     this.addToSet(obj);  
616 }
```

Initialize Layer in index.js

617

In addition to import/export the Layer component, modify the engine init() and cleanUp() functions in index.js to initialize and clean up the component:

618

619

```
... identical to previous code ...
function init(htmlCanvasID) {
    glSys.init(htmlCanvasID);
    vertexBuffer.init();
    input.init(htmlCanvasID);
    audio.init();
    shaderResources.init();
    defaultResources.init();
    layer.init();
}
function cleanUp() {
    layer.cleanUp();
    loop.cleanUp();
    shaderResources.cleanUp();
    defaultResources.cleanUp();
    audio.cleanUp();
    input.cleanUp();
    vertexBuffer.cleanUp();
    glSys.cleanUp();
}
```

620

621

622

623

624

625

626

627

628

629

630

631

632

633

634

635

636

637

638

639

Define the Update Function for Layer Membership

640

Define update functions for objects that may appear as members in the Layer layers:

641

Renderable and ShadowReceiver.

642

Modify MyGame to Work with the Layer Component

643

The MyGame level implements the same functionality as in the previous project. The only difference is the delegation of layer management to the Layer component. The following description focuses only on function calls relevant to layer management.

644

645

646

647 1. Modify the `unload()` function to clean up the Layer:

```
648     unload() {  
649         engine.layer.cleanUp();  
  
650         engine.texture.unload(this.kMinionSprite);  
651         engine.texture.unload(this.kBg);  
652         engine.texture.unload(this.kBgNormal);  
653         engine.texture.unload(this.kBgLayer);  
654         engine.texture.unload(this.kBgLayerNormal);  
655         engine.texture.unload(this.kMinionSpriteNormal);  
656     }
```

657 2. Modify the `init()` function to add the game objects to the
658 corresponding layers in the Layer component:

```
659     init() {  
660         ... identical to previous code ...  
  
661         // add to layer managers ...  
662         engine.layer.addToLayer(engine.layer.eBackground, this.mBg);  
663         engine.layer.addToLayer(engine.layer.eShadowReceiver,  
664                               this.mBgShadow1);  
  
665         engine.layer.addToLayer(engine.layer.eActors, this.mIllumMinion);  
666         engine.layer.addToLayer(engine.layer.eActors, this.mLgtMinion);  
667         engine.layer.addToLayer(engine.layer.eActors, this.mIllumHero);  
668         engine.layer.addToLayer(engine.layer.eActors, this.mLgtHero);  
  
669         engine.layer.addToLayer(engine.layer.eFront, this.mBlock1);  
670         engine.layer.addToLayer(engine.layer.eFront, this.mBlock2);  
671         engine.layer.addToLayer(engine.layer.eFront, this.mFront);  
  
672         engine.layer.addToLayer(engine.layer.eHUD, this.mMsg);  
673         engine.layer.addToLayer(engine.layer.eHUD, this.mMatMsg);  
674     }
```

3. Modify the draw() function to rely on the Layer component for
the actual drawings:

```
draw() {  
    engine.clearCanvas([0.9, 0.9, 0.9, 1.0]); // clear to light gray  
  
    this.mCamera.setViewAndCameraMatrix();  
engine.layer.drawAllLayers(this.mCamera);  
  
    if (this.mShowParallaxCam) {  
        this.mParallaxCam.setViewAndCameraMatrix();  
engine.layer.drawAllLayers(this.mParallaxCam);  
    }  
}
```

4. Modify the update() function to rely on the Layer component for
the actual update of all game objects:

```
update() {  
    this.mCamera.update(); // to ensure proper interpolated movement  
    this.mParallaxCam.update();  
  
engine.layer.updateAllLayers();  
  
    ... identical to previous code ...  
}
```

Observations

You can now run the project and observe the same output and interactions as the
previous project. The important observation for this project is in the implementation.
By inserting game objects to the proper layers of the Layer component during init(),
the draw() and update() functions of a game level can be much cleaner. The simpler
and cleaner update() function is of special importance. Instead of being crowded
with mundane game object update() function calls, this function can now focus on
implementing the game logic and controlling the interactions between game objects.

702 Summary

703 This chapter explained the need for tiling and introduced the `TileGameObject` to
704 implement a simple algorithm that tiles and covers a given camera WC bound. The
705 basics of parallax and approaches to simulate motion parallax with parallax scrolling
706 were introduced. Motion parallax with stationary and moving cameras were examined,
707 and solutions were derived and implemented. You learned that computing movements
708 relative to the camera motions to displace background objects results in visually
709 pleasing motion parallax but may cause player confusion when viewed from different
710 cameras. With shadow computations introduced earlier and now parallax scrolling,
711 game programmers must dedicate code and attention to coordinate the drawing order
712 of different types of objects. To facilitate the programmability of the game engine, the
713 Layer engine component is presented as a utility tool to relieve game programmers from
714 managing the drawing of the layers.

715 The game engine proposed by this book is now complete. It can draw objects with
716 texture maps, sprite animations, and even support illumination from various light
717 sources. The engine defines proper abstractions for simple behaviors, implements
718 mechanisms to approximate and accurately compute collisions, and simulates physical
719 behaviors. Views from multiple cameras can be conveniently displayed over the same
720 game screens with manipulation functionality that is smoothly interpolated. Keyboard/
721 mouse input is supported, and now background objects can scroll without bounds with
722 motion parallax simulated.

723 The important next step, to properly test your engine, is to go through a simple game
724 design process and implement a game based on your newly completed game engine.

725 Game Design Considerations

726 In previous sections, you've explored how developing one simple game mechanic from
727 the ground up can lead in many directions and be applied to a variety of game types.
728 Creative teams in game design studios frequently debate which elements of game design
729 take the lead in the creative process: writers often believe story comes first, while many
730 designers believe that story and everything else must be secondary to gameplay. There's
731 no right or wrong answer, of course; the creative process is a chaotic system and every
732 team and studio is unique. Some creative directors want to tell a particular story and will
733 search for mechanics and genres that are best suited to supporting specific narratives,
734 while others are gameplay purists and completely devoted to a culture of "gameplay"

first, next, and last." The decision often comes down to understanding your audience; 735
if you're creating competitive multiplayer first-person shooter experience, for example, 736
consumers will have specific expectations for many of the core elements of play, and it's 737
usually a smart move to ensure that gameplay drives the design. If you're creating an 738
adventure game designed to tell a story and provide players with new experiences and 739
unexpected twists, however, story and setting might lead the way. 740

Many game designers (including seasoned veterans as well as those new to the 741
discipline) begin new projects by designing experiences that are relatively minor 742
variations on existing well-understood mechanics; while there are sound reasons for this 743
approach (as in the case of AAA studios developing content for particularly demanding 744
audiences or a desire to work with mechanics that have proven to be successful across 745
many titles), it tends to significantly limit exploration into new territory and is one 746
reason why many gamers complain about creative stagnation and a lack of gameplay 747
diversity between games within the same genre. Many professional game designers grew 748
up enjoying certain kinds of games and dreamed about creating new experiences based 749
on the mechanics we know and love, and several decades of that culture has focused 750
much of the industry around a comparatively few numbers of similar mechanics and 751
conventions. That said, a rapidly growing independent and small studio community 752
has boldly begun throwing long-standing genre convention to the wind in recent years 753
and new distribution platforms like mobile app stores and Valve's Steam have opened 754
opportunities for a wide range of new game mechanics and experiences to flourish. 755

If you continue exploring game design, you'll realize there are relatively few 756
completely unique core mechanics but endless opportunities for innovating as you 757
build those elemental interactions into more complex causal chains and add unique 758
flavor and texture through elegant integration with the other elements of game 759
design. Some of the most groundbreaking and successful games were created through 760
exercises very much like the mechanic exploration you've done in these "Game Design 761
Considerations" sections; Valve's Portal, for example, is based on the same kind of 762
"escape the room" sandbox you have been exploring and is designed around a similarly 763
simple base mechanic. What made Portal such a breakthrough hit? While many things 764
need to come together to create a hit game, Portal undoubtedly benefitted from a design 765
team that started building the experience from the most basic mechanic and smartly 766
increased complexity as they became increasingly fluent in its unique structure and 767
characteristics, instead of starting at the 10,000-foot level with a codified genre and a 768
predetermined set of design rules. 769

770 Of course, nobody talks about Portal without also mentioning the rogue artificial
771 intelligence character GLaDOS and her Aperture Laboratories playground: setting,
772 narrative, and audiovisual design are as important to the Portal experience as the portal-
773 launching game mechanic, and it's hard to separate the gameplay from the narrative
774 given how skillfully intertwined they are. The projects in this chapter provide a good
775 opportunity to begin similarly situating the game mechanic from the "Game Design
776 Considerations" sections in a unique setting and context: you've probably noticed many
777 of the projects throughout this book are building toward a sci-fi visual theme, with a
778 spacesuit-wearing hero character, a variety of flying robots, and now in Chapter 11 the
779 introduction of large-scale parallax environments, and while you're not building a game
780 with the same degree of environment and interaction complexity as Portal, that doesn't
781 mean you don't have the same opportunity to develop a highly engaging game setting,
782 context, and cast of characters.

783 The first thing you should notice about the Tiled Objects project is the dramatic
784 impact on environment experience and scale compared to earlier projects. The
785 factors enhancing presence in this project are the three independently moving layers
786 (hero character, moving wall, and stationary wall) and the seamless tiling of the two
787 background layers. Compare the Tiled Objects project to the Shadow Shaders project
788 from Chapter 8, and notice the difference in presence when the environment is broken
789 into multiple layers that appear to move in an analogous (if not physically accurate)
790 way to how you experience movement in the physical world. The sense of presence is
791 further strengthened when you add multiple background layers of parallax movement in
792 the Parallax Objects project; as you move through the physical world, the environment
793 appears to move at different speeds, with closer objects seeming to pass by quickly
794 while objects toward the horizon appear to move slowly. Parallax environment objects
795 simulate this effect, adding considerable depth and interest to game environments. The
796 Layer Manager project pulls things together and begins to show the potential for a game
797 setting to immediately engage the imaginations of players. With just a few techniques,
798 you're able to create the impression of a massive environment that might be the interior
799 of an ancient alien machine, the outside of a large spacecraft, or anything else you might
800 care to create. Try using different kinds of image assets with this technique: exterior
801 landscapes, underwater locations, abstract shapes, and the like would all be interesting
802 to explore. You'll often find inspiration for game settings by experimenting with just a
803 few basic elements, as you did in Chapter 11.

Pairing environment design (both audio and visual) with interaction design (and occasionally the inclusion of haptic feedback-like controller vibrations) is an approach you can use to create and enhance presence, and the relationship that environments and interactions have with the game mechanic contributes the majority of what players experience in games. Environment design and narrative context create the game setting, and as previously mentioned, the most successful and memorable games achieve an excellent harmony between game setting and player experience. At this point, the game mechanic from the “Game Design Considerations” section in Chapter 9 has been intentionally devoid of any game setting context, and you’ve only briefly considered the interaction design, leaving you free to explore any setting that captures your interest.	804
In Chapter 12, you’ll further evolve the sci-fi setting and image assets used in the main chapter projects with the unlocking mechanic from the “Game Design Considerations” section to create a fairly advanced 2D platformer game-level prototype.	816

Building a Sample Game: From Design to Completion

The projects included in the main sections of Chapters 1 to 11 began with simple shapes and slowly introduced characters and environments to illustrate the concepts of each chapter; those projects focused on individual behaviors and techniques (such as collision detection, object physics, lighting, and the like) but lacked the kind of structured challenges necessary to deliver a full gameplay experience. The projects in the “Design Considerations” sections demonstrate how to introduce the types of logical rules and challenges required to turn basic behaviors into well-formed game mechanics. This chapter now changes the focus to emphasize the design process from an early concept through a functional prototype, bringing together and extending the work done in earlier projects by using some characters and environments from prior chapters along with the basic idea for the unlocking platform game from the “Design Considerations” section of Chapter 11. As with earlier chapters, the design framework utilized here begins with a simple and flexible starting template and adds complexity incrementally and intentionally to allow the game to grow in a controlled manner.

The design exercises have until now avoided consideration of most of the nine elements of game design described in the “How Do You Make a Great Video Game?” section of Chapter 1 and instead focused on crafting the basic game mechanic in order to clearly define and refine the core characteristics of the game itself. The design approach used in this book is a ground-up framework that emphasizes first working with an isolated game mechanic prior to the consideration of the game’s genre or setting; when you begin incorporating a setting and building out levels that include additional design elements on top of the core mechanic, the gameplay will grow and evolve in

unique directions as you grow the game world. There are endless potential variations for a game's mechanic and the associated gameplay loops you design. You'll be surprised by how differently the same foundational elements of gameplay develop and evolve based on the kind of creative choices you make.

Part 1: Refining the Concept

By this point, you should have the beginnings of a concept using a 2D jumping and puzzle-solving mechanic that revolves around unlocking a barrier and to reach a reward. Recall Figure 12-1 as the final screen layout and design from Chapter 11.

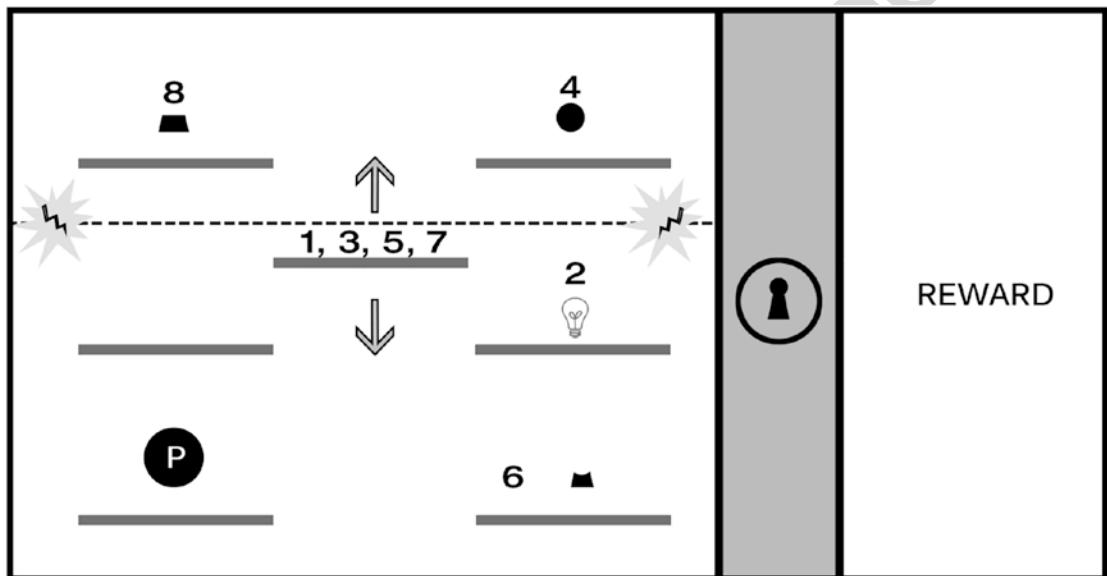


Figure 12-1. The 2D implementation from Chapter 11

This design already has a multistage solution requiring the player to both demonstrate timing-based agility and puzzle-solving logic. In the current design, the player controls the hero character (perhaps by using the A and D keys to move left and right and using the spacebar to jump). The player can jump between horizontal platforms on the same level but can't reach platforms above without using the middle "elevator" platform that rises and falls. A horizontal "energy field" will zap the player if

they touch it, causing the game to reset. The explicit steps to completing the level are as follows:

1. The player must jump the hero character (the circle with the letter *p* in the center of Figure 12-1) on the moving elevator platform (#1 in Figure 12-1) and jump off to the middle platform of the right column before touching the energy field. 43
44
45
46
2. The player activates the off switch for the energy field by colliding the hero character with it (#2, represented by the lightbulb icon in Figure 12-1). 47
48
49
3. When the energy field is switched off, the player rides the elevator platform to the top (#3) and jumps the hero to the top platform in the right column. 50
51
52
4. The player collides the hero with the small circle that represents the top third of the lock icon (#4), activating the corresponding part of the lock icon and making it glow. 53
54
55
5. The player jumps the hero back on the elevator platform (#5) and then jumps the hero to the bottom platform in the right column. 56
57
6. The player collides the hero with the shape corresponding to the middle section of the lock icon (#6), activating the corresponding part of the lock icon and making it glow. Two-thirds of the lock icon now glows, signaling progress. 58
59
60
61
7. The player jumps the hero on the elevator platform once again (#7) and then jumps the hero to the top platform in the left column. 62
63
64
8. The player collides the hero with the shape corresponding to the bottom section of the lock icon (#8), activating the final section of the icon and unlocking the barrier. 65
66
67

Writing out this sequence (or *game flow diagram*) may seem unnecessary given the mock-up screens you've created. It's important, however, for designers to understand everything the player must do in exact order and detail to ensure you're able to tune, balance, and evolve the gameplay without becoming mired in complexity or losing sight of how the player makes their way through the level. It's clear from diagramming the

previous game flow, for example, that the elevator platform is the centerpiece of this level and is required to complete every action; this is great information to have available in a schematic representation and game flow description because it provides an opportunity to intelligently refine the gameplay logic in a way that allows you to visualize the effect of each change on the overall flow of the level.

You could continue building out the mechanic to make the level more interesting and challenging (e.g., you might include a timer on the energy field's off switch requiring players to collide with all the lock parts within a limited amount of time). However, at this stage of concept development, it's often helpful to take a step back from gameplay and begin considering game setting and genre, using those elements to help inform how the game mechanic evolves from here.

Recall from Chapter 11 that the projects ended with a set of concept explorations supporting a sci-fi setting. Figure 12-2 shows a futuristic industrial environment design, a hero character wearing a space suit, and what appear to be flying robots.

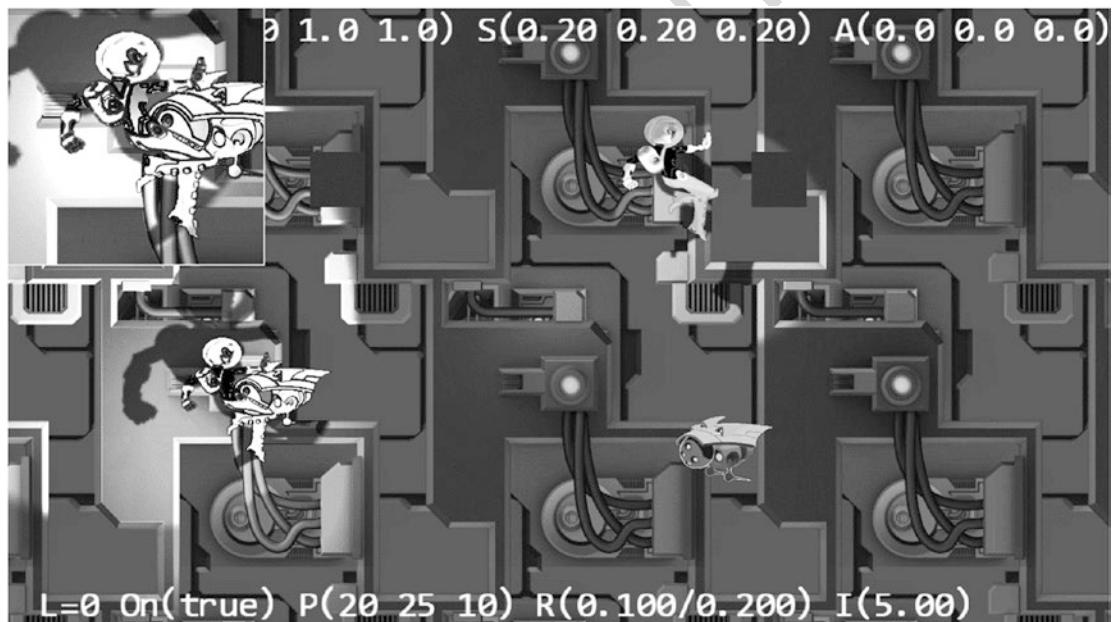


Figure 12-2. Concepts from Chapter 11

Note there isn't anything specific about the game mechanic you've been creating that would necessarily lead you in a sci-fi direction; game mechanics are abstract interactive structures and can typically integrate with any kind of setting or visual style.

In this case, the authors of this book chose a setting that take place on spaceships, so this chapter will use that motif as the setting for the game prototype. As you proceed through the design process, consider exploring alternate settings: how might the game mechanic from Chapter 11 be adapted to a jungle setting, a contemporary urban location, a medieval fantasy world, or an underwater metropolis? 90
91
92
93
94

Part 2: Integrating a Setting

Now is a good time to begin assigning some basic fictional background to evolve and extend the game mechanic in unique ways that enhance the setting you choose (don't worry if this is unclear at the moment; the mechanism will become more apparent as you proceed with the level design). Imagine, for example, that the hero character is a member of the crew on a large spaceship and that she must complete a number of objectives to save the ship from exploding. Again, there is nothing about the current state of the game mechanic driving this narrative; the design task at this stage includes brainstorming some fictional context that propels the player through the game and captures their imagination. Using the few concept art assets already created/provided, the hero could just as easily be participating in a race, looking for something that was lost, exploring an abandoned alien vessel, or any of a million other possibilities. 95
96
97
98
99
100
101
102
103
104
105
106

Contextual Images Bring the Setting to Life

Now that you've described a basic narrative and fictional wrapper that reads something like "Players must complete a series of platformer puzzle levels to save their spaceship before it explodes," swap just a few of the shapes from the mechanic design with some of the included concept elements. Figure 12-3 introduces a humanoid hero character, platforms that feel a bit more like spaceship components, and a barrier wall with a locked door to replace the abstract lock from the mechanic design. 107
108
109
110
111
112
113

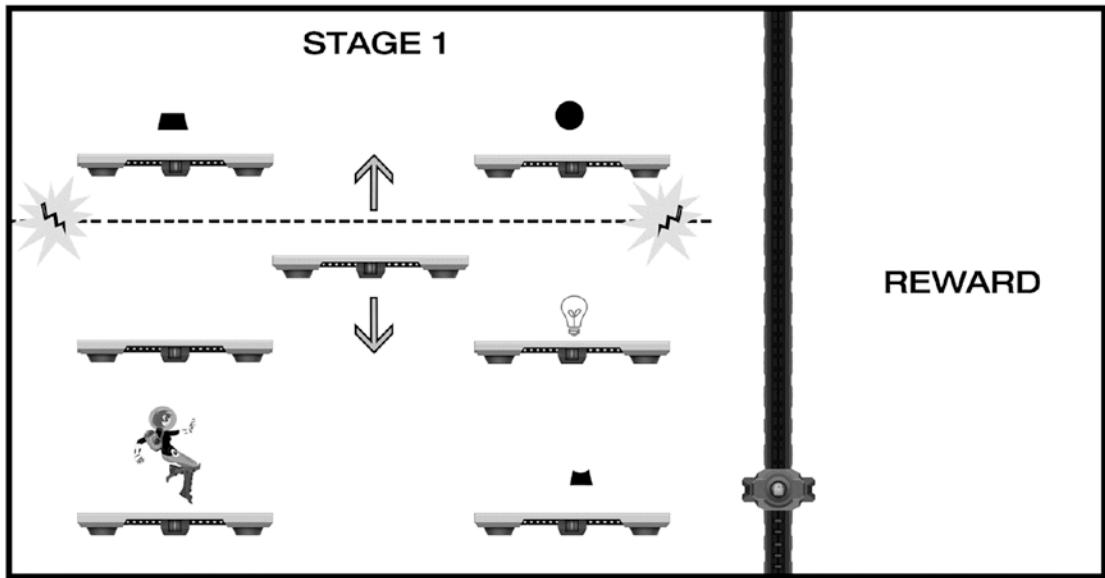


Figure 12-3. The introduction of several visual design elements supporting the game setting and evolving narrative

114 Although you've made only a few minor substitutions and don't yet have the visual
 115 elements anchored in an environment, Figure 12-3 conveys quite a bit more fictional
 116 context and contributes significantly more to presence than the abstract shapes of
 117 Figure 12-1. The hero character now suggests a scale that will naturally be contextualized
 118 as players benchmark relative sizes against the human figure, which brings the relative
 119 size of the entire game environment into focus for players. The implementation of
 120 object physics for the hero character as described in Chapter 10 also becomes an
 121 important component of play: simulated gravity, momentum, and the like connect
 122 players viscerally to the hero character as they move them through the game world. By
 123 implementing the design as described in Figure 12-3, you've already accomplished some
 124 impressive cognitive feats that support presence simply by adding a few visual elements
 125 and some object physics.

126 Defining the Playable Space

127 At this point in the design process, you've sufficiently described the game's core
 128 mechanic and setting to begin expanding the single screen into a full-level concept.
 129 It's not critical at this stage to have a final visual style defined, but including some
 130 concept art will help guide how the level grows. (Figure 12-3 provides a good visual

representation for the amount of gameplay that will take place on a single screen given
the scale of objects.) This is also a good stage to “block in” the elements from Figure 12-3
in a working prototype to begin getting a sense for how movement feels (e.g., the speed
the hero character runs, the height the hero can jump, and so on), the scale of objects
in the environment, the zoom level of the camera, and the like. There’s no need to
include interactions and behaviors such as the lock components or the energy field at
this stage because you haven’t yet designed how the level will play. At this stage, you’re
experimenting with basic hero character movement, object placement, and collision.
The next set of tasks includes laying out the full level and tuning all the interactions.

The current state of the design in Figure 12-3 still needs some work to provide
sufficient challenge. While all the elements of a well-formed level are in place, the
current difficulty is trivial, and most players will likely be able to complete the level
quickly. There is, however, a strong foundation to begin extending the jumping and
sequencing mechanic; to begin, you might extend the horizontal game space to include
a more playable area and provide additional room for the character to maneuver, as
shown in Figure 12-4.

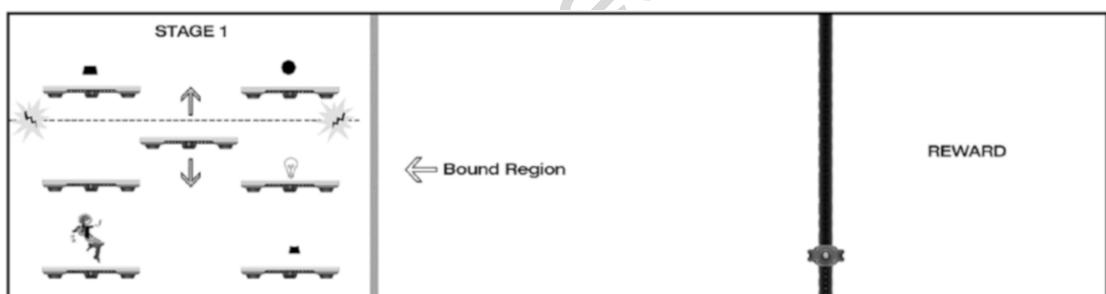


Figure 12-4. The level design grows to include an additional playable area

Recall from the Simple Camera Manipulations project in Chapter 7 that you can
“push” the game screen forward by moving the character close to the edge of the bound
region, which allows you to design a level that extends far beyond the dimensions of a
single static screen. You might choose to keep this level contained to the original game
screen size, of course, and increase the complexity of the timing-based agility and logical
sequence challenges (and indeed it’s a good design exercise to challenge yourself to
work within space constraints), but for the purposes of this design, a horizontal scrolling
presentation adds interest and challenge.

155

Adding Layout to the Playable Space

156 It's now time to begin laying out the level to make good use of the additional horizontal
 157 space. There's no need to change the basic gameplay at this point; you'll simply expand
 158 the current level design to fit the new dimensions of the game screen. Figure 12-5
 159 includes some additional platforms placed with no particular methodology other than
 160 ensuring players can successfully reach each platform.

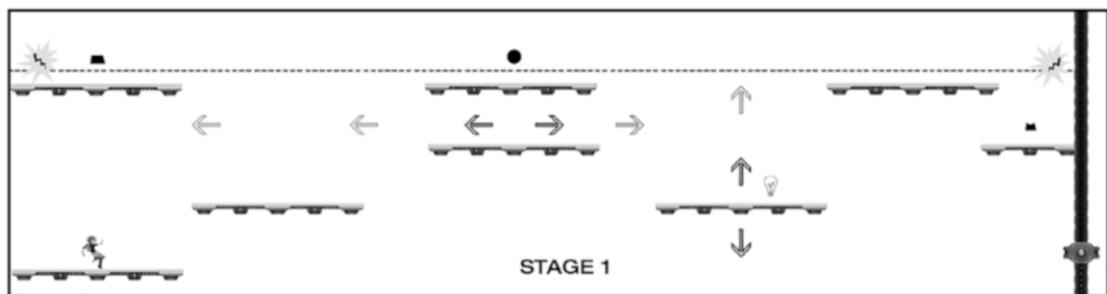


Figure 12-5. Expands the layout to use the additional screen real estate, the diagram represents the entire length of stage 1 with the player able to see approximately 50 percent of the full level at any time. The camera scrolls the screen forward or backward as the player moves the hero character toward the screen bound regions. Note: for the moving platforms shown, darker arrows represent direction, and lighter arrows represent the range of the platform's movement

161 Now that the level has some additional space to work with, there are several factors
 162 to evaluate and tune. For example, the scale of the hero character in Figure 12-5 has
 163 been reduced to increase the number of vertical jumps that can be executed on a single
 164 screen. Note that at this point, you also have the opportunity to include additional
 165 vertical gameplay in the design if desired, implementing the same mechanism used
 166 to move the camera up and down that you used to move it left and right; many 2D
 167 platformer games allow players to move through the game world both horizontally
 168 and vertically. This level prototype will limit movement to the x plane (left and right)
 169 for simplicity although you can easily extend the level design to include vertical play in
 170 future iterations and/or subsequent levels.

171 As you're placing platforms in the level, you will again want to minimize design
 172 complexity while blocking out the game flow. Figure 12-5 adds one additional design
 173 element: a platform that moves left to right. Try to list the detailed sequence required to

activate the three lock sections in Figure 12-5 using the same numbering methodology shown in Figure 12-1. When you’re finished mapping out the sequence, compare it with Figure 12-6.

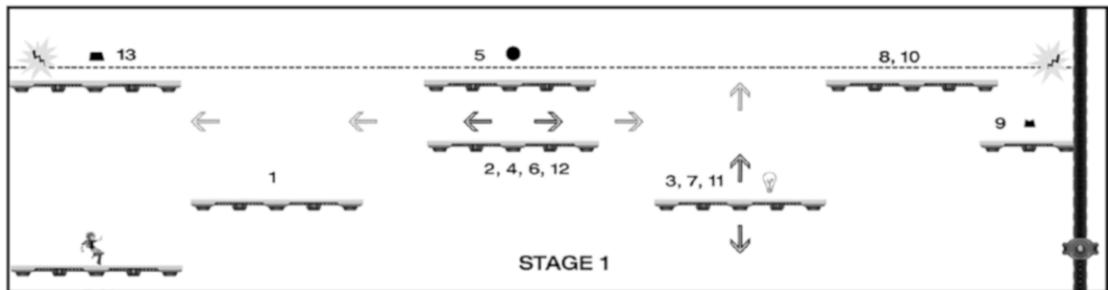
174
175
176

Figure 12-6. The most efficient sequence to unlock the barrier

Did your sequence match Figure 12-6, or did you have extra steps? There are many potential paths players can take to complete this level, and it’s likely that no two players will take the same route (the only requirement from the mechanic design is that the lock sections be activated in order from top to bottom).

177
178
179
180

Tuning the Challenge and Adding Fun

In this stage of the design is when the puzzle-making process really begins to open up; Figure 12-6 shows the potential to create highly engaging gameplay with only the few basic elements you’ve been working with. The authors use the previous template and many similar variations in brainstorming sessions for many kinds of games—introducing one or two novel elements to a well-understood mechanic and exploring the impact new additions have on gameplay—and the results often open exciting new directions. As an example, you might introduce platforms that appear and disappear, platforms that rotate after a switch is activated, a moving energy field, teleporting stations, and so on. The list of ways you can build out this mechanic is of course limitless, but there is enough definition with the current template that adding a single new element is fairly easy to experiment with and test, even on paper.

182
183
184
185
186
187
188
189
190
191
192

There are two factors with the newly expanded level design that increase the challenge. First, the addition of the horizontally moving platform requires players to time the jump to the “elevator” platform more precisely (if they jump while the platform is ascending, there is little time to deactivate the energy field before it zaps them).

193
194
195
196

197 The second factor is less immediately evident but equally challenging: only a portion of
198 the level is visible at any time, so the player is not able to easily create a mental model of
199 the entire level sequence, like they can when the entire layout is visible on a single screen.
200 It's important for designers to understand both explicit challenges (such as requiring
201 players to time jumps between two moving platforms) and less obvious (and often
202 unintentional) challenges such as being able to see only part of the level at any given
203 time. Think back to a game you've played where it felt like the designers expected you
204 remember too many elements; that kind of frustration is often the result of unintentional
205 challenges overburdening what the player can reasonably hold in short-term memory.

206 As a designer, you need to be aware of unintentional challenges and areas of
207 unintentional frustration or difficulty; these are a key reason why it's vital to observe
208 people playing your game as early and often as possible. As a general rule, any time
209 you're 100 percent certain you've designed something that makes perfect sense, at least
210 half the people who play your game will tell you exactly the opposite. Although a detailed
211 discussion of the benefits of user testing is outside the scope of this book, you should
212 plan to observe people playing your game from the earliest proof of concept all the way
213 to final release. There is no substitute for the insights you'll gain from watching different
214 people play what you've designed.

215 The level as described in previous figures currently assumes the hero character
216 can rest only on platforms; although there's no design plan for what happens if the
217 character misses a jump and falls to the bottom of the screen, players might reasonably
218 imagine that it would result in a loss condition and trigger a game reset. If you added a
219 "floor" to the level, player strategy would noticeably change; in addition to removing a
220 significant risk, players would be able to access the elevator platform directly, as shown
221 in Figure 12-7.

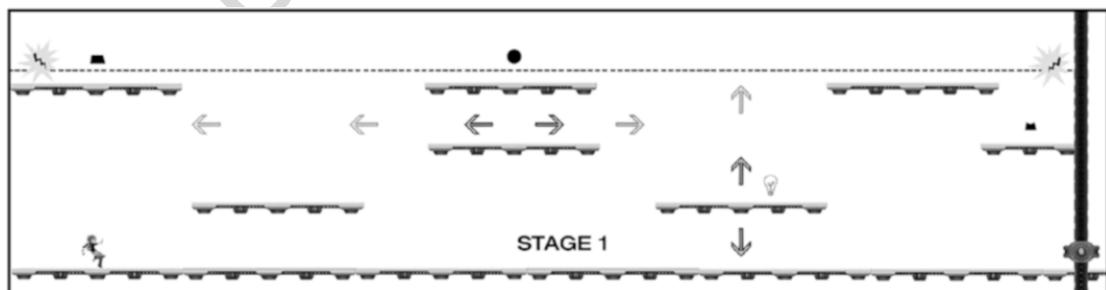
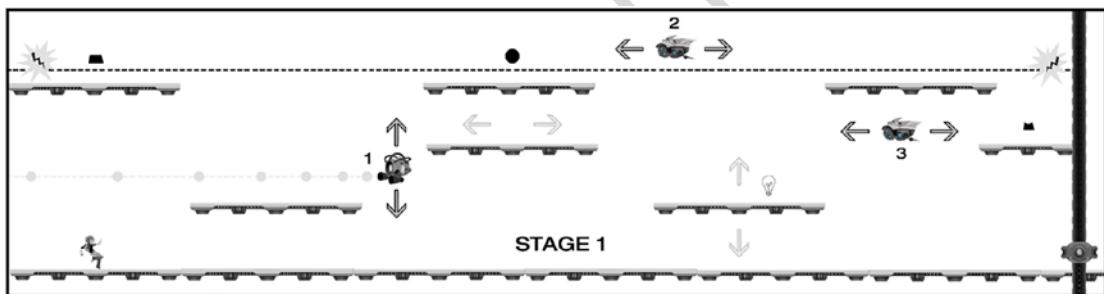


Figure 12-7. The addition of a "floor" to the game world significantly changes the level challenge

Further Tuning: Introducing Enemies

222
223
224
225
226
227
228
229
230
231
232
233
234

You're now experimenting with variations on the level layout to evolve it along with the setting and looking for ways to increase player engagement while also upping the challenge (if desired). Prior to adding a floor, the level had two risks: failing to land on a platform and triggering a loss condition, and colliding with the energy field and triggering a loss condition. The addition of the floor removes the falling risk and potentially decreases the challenge of the level, but you might decide that the floor encourages players to more freely explore, experiment, and pay closer attention to the environment. You're also now becoming increasingly conversant with the gameplay and flow for this mechanic and layout, so let's introduce a new element: attacking enemies (we can't let those robot designs from previous chapters go to waste)! Figure 12-8 introduces two basic enemy robot types: one that fires a projectile and one that simply patrols.



this figure will be printed in b/w

Figure 12-8. Two new object types are introduced to the level: a shooting robot (#1) that moves vertically and fires at a constant rate and a patrolling robot (#2) that moves back and forth in a specific range

You've now reached a turning point in the design of this level, where the setting is starting to exert a significant influence on the evolution of the mechanic and game loop. The core of the mechanic hasn't changed from Chapter 11, and this level is still fundamentally about activating sections of a lock in the proper sequence to remove a barrier, but the moving platforms and attacking enemies are additional obstacles and are strongly influenced by the particular setting you've chosen.

235
236
237
238
239
240
241
242
243
244

Of course, you certainly could have added the attacking enemy behavior while still working with abstract shapes and pure mechanics. It's worth noting, however, that the more complex and multistaged a mechanic becomes, the more the setting will need to conform to the implementation; this is why transitioning from purely abstract mechanic

245 design to laying out a level (or part of a level) in the context of a particular setting when
246 the mechanic is still fairly elemental is helpful. Designers typically want the game
247 mechanic to feel deeply integrated with the game setting, so it's beneficial to allow
248 both to develop in tandem. Finding that sweet spot can be challenging: sometimes the
249 mechanic leads the design, but as the setting evolves, it often will move into the driver's
250 seat. Bring in the setting too soon and you lose focus on refining pure gameplay; bring
251 in the setting too late and gameplay may feel like an afterthought or something that's
252 bolted on.

253 General Considerations

254 Returning to the current design as represented in Figure 12-8, you now have all the
255 elements required to create a truly engaging sequence situated in an emerging setting.
256 You can also fairly easily tune the movement and placement of individual units to make
257 things more or less challenging. Players will need to observe patterns of movement for
258 both platforms and enemies to time their jumps so they can navigate the level without
259 getting zapped or rammed, all while discovering and solving the unlocking puzzle.
260 Note how quickly the level went from trivially easy to complete to potentially quite
261 challenging: working with multiple moving platforms adds an element of complexity,
262 and the need to use timing for jumps and adding attacking enemies—even the simple
263 enemies from Figure 12-8 that are locked into basic movement patterns—opens nearly
264 unlimited possibilities to create devious puzzles in a controlled and intentional way.

265 If you haven't already, now is a good time to prototype your level design (including
266 interactions) in code to validate gameplay. For this early-level prototype, it's only
267 important that major behaviors (running, jumping, projectile firing, moving platforms,
268 object activations, and the like) and steps required to complete the level (puzzle
269 sequences) are properly implemented. Some designers insist at this stage that players
270 who have never encountered the level before should be able to play through the entire
271 experience and fully understand what they need to do with little or no assistance, while
272 others are willing to provide direction and fill in gaps around missing onscreen UI and
273 incomplete puzzle sequences. It's common practice to playtest and validate major
274 sections of gameplay at this stage and provide playtesters with additional guidance to
275 compensate for incomplete UI or unimplemented parts of a sequence. As a general
276 rule, the less you need to rely on over-the-shoulder guidance for players at this stage,
277 the better your insights into the overall design will be. The amount of the early-level

prototype you'll implement at this stage also depends on the size and complexity of your design. Large and highly complex levels may be implemented and tested in several (or many) pieces before the entire level can be played through at once, but even in the case of large and complex levels, the goal is to have the full experience playable as early as possible.

Note If you've been exploring the working prototype included with this book, you'll discover some minor variations between the design concepts in this chapter and the playable level (the energy field was not included in the working prototype, e.g.). Consider exploring alternate design implementations with the included assets; exploration and improvisation are key elements of the creative-level design process. How many extensions of the current mechanic can you create?

Part 3: Integrating Additional Design Elements

The prototype you've been building in this chapter would serve as an effective proof of concept for a full game at its current level of development, but it's still missing many elements typically required for a complete game experience (including visual detail and animations, sounds, scoring systems, win conditions, menus and user interface [UI] elements, and the like). In game parlance, the prototype level is now at the *blockout-plus* stage (*blockout* is a term used to describe a prototype that includes layout and functional gameplay but lacks other design elements; the inclusion of some additional concept art is the "plus" here). It's now a good time to begin exploring audio, scoring systems, menu and onscreen UI, and the like. If this prototype were in production at a game studio, a small group might take the current level to a final production level of polish and completeness while another team worked to design and prototype additional levels. A single level or a part of a level that's taken to final production is referred to as a *vertical slice*, meaning that one small section of the game includes everything that will ship with the final product. Creating a vertical slice is helpful to focus the team on what the final experience will look, feel, and sound like and can be used to validate the creative direction with playtesters.

306 Visual Design

307 Although you've begun integrating some visual design assets that align with the setting
308 and narrative, the game typically will have few (if any) final production assets at this
309 time and any animations will be either rough or not yet implemented (the same is true
310 for game audio). While it's good practice to have gameplay evolve in parallel with the
311 game setting, studios don't want to burn time and resources creating production assets
312 until the team is confident that the level design is locked and they know what objects are
313 needed and where they'll be placed.

314 You should now have a fairly well-described layout and sequence for your level
315 design (if you've been experimenting with a different layout compared to what's shown
316 in the examples, make sure you have a complete game flow described as in Figures 12-1
317 and 12-6.) At this point in the project, you can confidently begin "rezzing in" production
318 assets (*rezzing in* is a term used by game studios to mean increasing the resolution—
319 in this case, the visual polish and overall production quality of the level—over time).
320 Rezzing in is typically a multistage process that begins when the major elements of the
321 level design are locked, and it can continue for most of the active production schedule.
322 There are often hundreds (or thousands) of individual assets, animations, icons, and
323 the like that will typically need to be adjusted multiple times based on the difference
324 between how they appear outside the game build and inside the game build. Elements
325 that appear to harmonize well in isolation and in mockups often appear quite differently
326 after being integrated into the game.

327 The process of rezding-in assets can be tedious and frustrating (there always
328 seems to be an order of magnitude more assets than you think there will be). It can
329 also be challenging to make things look as awesome in the game as they do in an
330 artist's mockups. However, it's typically a satisfying experience when it all start to come
331 together: something magical happens to a level design as it transitions from blockout
332 to polished production level, and there will usually be one build where a few key visual
333 assets have come in that make the team remark "Wow, now this feels like our game!" For
334 AAA 3D games, these "wow" moments frequently happen as high-resolution textures
335 are added to 3D models and as complex animations, lighting, and shadows bring the
336 world to life; for the current prototype level, adding a parallaxing background and some
337 localized lighting effects should really make the spaceship setting pop.

338 The working prototype included with this book represents a build of the final game
339 that would typically be midway between blockout and production polish. The hero
340 character includes several animation states (idle, run, jump), localized lighting on the

hero and robots adds visual interest and drama, the level features a two-layer parallaxing background with normal maps that respond to the lighting, and major game behaviors are in place. You can build upon this prototype and continue to polish the game or modify it how you see fit. 344

Game Audio 345

Many new game designers (and even some veteran designers) make the mistake of treating audio as less important than the visual design, but as every gamer knows, bad audio in some cases can mean the difference between a game you love and a game you stop playing after a short time. As with visual design, audio often contributes directly to the game mechanic (e.g., countdown timers, warning sirens, positional audio that signals enemy location), and background scores enhance drama and emotion in the same way that directors use musical scores to support the action on film). However, audio in mobile games is often considered optional because many players mute the sound on their mobile devices. Well-designed audio, however, can have a dramatic impact on presence even for mobile games. In addition to sounds corresponding to game objects (walking sounds for characters who walk, shooting sounds for enemies who fire, popping sounds for things that pop, and the like), contextual audio attached to in-game actions is an important feedback mechanism for players. Menu selections, activating in-game switches, and the like should all be evaluated for potential audio support. As a general rule, if an in-game object responds to player interaction, it should be evaluated for contextual audio. 361

Audio designers work with level designers to create a comprehensive review of game objects and events that require sounds, and as the visuals rezz in, the associated sounds will typically follow. Game sounds often lag behind visual design because audio designers want to see what they’re creating sounds for; it’s difficult to create a “robot walking” sound, for example, if you can’t see what the robot looks like or how it moves. In much the same way that designers want to tightly integrate the game setting and mechanic, audio engineers want to ensure that the visual and audio design work well together. 369

370 Interaction Model

371 The current prototype uses a common interaction model: A and D keys on the keyboard
372 move the character right and left, and the spacebar is used to jump. Object activations
373 in the world happen simply by colliding the hero character with the object, and the
374 design complexity is fairly low for those interactions. Imagine, however, that as you
375 continue building out the mechanic (perhaps in later levels), you include the ability
376 for the character to launch projectiles and collect game objects to store in inventory.
377 As the range of possible interactions in the game expands, complexity can increase
378 dramatically, and unintentional challenges (as mentioned previously) can begin to
379 accumulate, which can lead to bad player frustration (as opposed to “good” player
380 frustration, which as discussed earlier results from intentionally designed challenges).

381 It’s also important to be aware of the challenges encountered when adapting
382 interaction models between different platforms. Interactions designed initially for
383 mouse and keyboard often face considerable difficulty when moving to a game console
384 or touch-based mobile device. Mice and keyboard interaction schemes allow for extreme
385 precision and speed of movement compared to the imprecise thumb sticks of game
386 controllers, and although touch interactions can be precise, mobile screens tend to be
387 significantly smaller and obscured by fingers covering the play area. The industry took
388 many years and iterations to adapt the first-person-shooter (FPS) genre from using mice
389 and keyboards to game consoles, and FPS conventions for touch devices remain highly
390 variable more than a decade after the first mobile FPS experiences launched (driven in
391 part by the differences in processing capabilities and screen sizes of the many phones
392 and tablets on the market). If you plan to deliver a game across platforms, make sure you
393 consider the unique requirements of each as you’re developing the game.

394 Game Systems and Meta-game

395 The current prototype has few systems to balance and does not yet incorporate a meta-
396 game, but imagine adding elements that require balancing such as variable-length
397 timers for object activations or the energy field. If you’re unsure what this means,
398 consider the following scenario: the hero character has two potential ways to deactivate
399 the energy field, and each option is a trade-off. The first option perhaps deactivates the
400 energy field permanently but spawns more enemy robots and considerably increases
401 the difficulty in reaching the target object, while the second option does not spawn
402 additional robots but only deactivates the energy field for a short time, requiring

players to choose the most efficient path and execute nearly perfect timing. To balance
effectively between the two options, you need to understand the design and degree of
challenge associated with each system (unlimited vs. limited time). Similarly, if you
added hit points to the hero character and made the firing robot create x amount of
damage while the charging minion creates y amount of damage per hit, you'd want to
understand the relative trade-offs between paths to objectives, perhaps making some
paths less dangerous but more complex to navigate, while others might be faster to
navigate but more dangerous.

As with most other aspects of the current design, there are many directions you
could choose to pursue in the development of a meta-game; what might you provide
to players for additional positive reinforcement or overarching context as they played
through a full game created in the style of the prototype level? As one example, imagine
that players must collect a certain number of objects to access the final area and prevent
the ship from exploding. Perhaps each level has one object that required players to
solve a puzzle of some kind before they could access it, and only after collecting the
object would they then be able to solve the door-unlocking component of the level.
Alternatively, perhaps each level has an object players can access to unlock cinematics
and learn more about what happened on the ship for it to reach such a dire state. Or
perhaps players are able to disable enemy robots in some way and collect points, with
a goal to collect as many points as possible by the end of the game. Perhaps you'll
choose to forego traditional win and loss conditions entirely. Games don't always focus
on explicit win and loss conditions as a core component of the meta-game, and for a
growing number of contemporary titles, especially indie games, it's more about the
journey than the competitive experience (or the competitive element becomes optional).
Perhaps you can find a way to incorporate both a competitive aspect (e.g., score the most
points or complete each level in the shortest time) and meta-game elements that focus
more on enhancing play.

A final note on systems and meta-game: player education (frequently achieved by
in-game tutorials) is an important component of these processes. Designers become
intimately acquainted with how the mechanics they design function and how the
controls work, and it's easy (and common) to lose awareness of how the game will
appear to someone who encounters it for the first time. Early and frequent playtests
help provide information about how much explanation players will require in order to
understand what they need to do, but most games require some level of tutorial support
to help teach the rules of the game world. Tutorial design techniques are outside the

438 scope of this book, but it's often most effective to teach players the logical rules and
439 interactions of the game as they play through an introductory level or levels. It's also
440 more effective to show players what you want them to do rather than making them read
441 long blocks of text (research shows that many players never access optional tutorials and
442 will dismiss tutorials with excessive text without reading them; one or two very short
443 sentences per tutorial event are a reasonable target). If you were creating an in-level
444 tutorial system for your prototype, how would you implement it? What do you think
445 players would reasonably discover on their own vs. what you might need to surface for
446 them in a tutorial experience? If you haven't played mobile games recently, it's worth
447 exploring a few titles and paying attention to how they integrate tutorial content with
448 gameplay.

449 User Interface (UI) Design

450 Game UI design is important not just from a functionality perspective (in-game menus,
451 tutorials, and contextually important information such as health, score, and the like)
452 but also as a contributor to the overall setting and visual design of the experience. Game
453 UI is a core component of visual game design that's frequently overlooked by new
454 designers and can mean the difference between a game people love and a game nobody
455 plays. Think back to games you've played that make use of complex inventory systems
456 or that have many levels of menus you must navigate through before you can access
457 common functions or items; can you recall games where you were frequently required
458 to navigate through multiple sublevels to complete often-used tasks? Or perhaps games
459 that required you to remember elaborate button combinations to access common game
460 objects?

461 Elegant and logical UI is critical to player comprehension, but UI that's integrated
462 into the game world also supports the game setting and narrative. Using the current
463 prototype and proposed systems design as reference, how would you visually represent
464 game UI in a way that supported the setting and aesthetic? If you haven't spent time
465 evaluating UI before (and even if you have), revisit several games with sci-fi settings
466 and pay particular attention to how they integrate UI elements visually in the game
467 screen. Figure 12-9 shows the weapon customization UI from Visceral Games' *Dead*
468 *Space 3*: note how the interface design is completely embedded within the game setting,
469 represented as an information screen on the fictional ship.



Figure 12-9. Most UI elements in Visceral Games' *Dead Space 3* are represented completely within the game setting and fiction, with menus appearing as holographic projections invoked by the hero character or on objects in the game world (image copyright Electronic Arts)

Many games choose to house their UI elements in reserved areas of the game screen (typically around the outer edges) that don't directly interact with the game world; however, integrating the visual aesthetic with the game setting is another way to contribute directly to the presence of the game. Imagine the current sci-fi prototype example with a fantasy-themed UI and menu system, using the kind of medieval aesthetic design and calligraphic fonts used for a game like Bioware's *Dragon Age*, for example; the resulting mismatch would be jarring and likely to pull players out of the game setting. User interface design is a complex discipline that can be challenging to master; you'll be well served, however, by spending focused time to ensure intuitive, usable, and aesthetically appropriate UI integration into the game worlds you create.

470
471
472
473
474
475
476
477
478
479

Game Narrative

480
481
482
483

At this stage, you've added just a basic narrative wrapper to the prototype example: a hero character must complete a number of objectives to prevent their spaceship from exploding. At the moment, you haven't explicitly shared this narrative with players at

484 all, and they have no way of knowing the environment is on a spaceship or what the
485 objective might be other than perhaps eventually unlocking the door at the far right
486 of the screen. Designers have a number of options for exposing the game narrative
487 to players; you might create an introductory cinematic or animated sequence that
488 introduces players to the hero character, their ship, and the crisis, perhaps choosing
489 something simple like a pop-up window at the start of the level with brief introduction
490 text that provides players with the required information. Alternatively, you might not
491 provide any information about what's happening when the game starts but instead
492 choose to slowly reveal the dire situation of the ship and the objectives over time as the
493 player proceeds through the game world. You could even choose to keep any narrative
494 elements implied, allowing players to overlay their own interpretation. As with many
495 other aspects of game design, there's no single way to introduce players to a narrative
496 and no universal guidance for how much (or how little) narrative might be required for a
497 satisfying experience.

498 Narrative can also be used by designers to influence the way levels are evolved
499 and built out even if those elements are never exposed to players. In the case of this
500 prototype, it's helpful as the designer to visualize the threat of an exploding ship to
501 propel the hero character through a series of challenges with a sense of urgency; players
502 however might experience a well-constructed side-scrolling action platformer only with
503 a series of devilishly clever levels. You might create additional fiction around robots that
504 have been infected with a virus, causing them to turn against the hero as a reason for
505 their attack behavior (as just one example). By creating a narrative framework for the
506 action to unfold within, you're able to make informed decisions about ways to extend
507 the mechanic that feel nicely integrated into the setting even if you don't share all the
508 background with players.

509 Of course, some game experiences have virtually no explicit narrative elements
510 either exposed to players or not and are simply implementations of novel mechanics.
511 Games like Zynga's *Words with Friends* and Gabriele Cirulli's hyper-casual *2048* are
512 examples of game experiences purely based on a mechanic with no narrative wrapper.

513 If you continue developing this prototype, how much narrative would you choose
514 to include, and how much would you want to expose to players to make the game come
515 alive?

Bonus Content: Adding a Second Stage to the Level

516

If you've completed playing through stage 1 of the included prototype, you'll enter a second room with a large moving unit; this is a sandbox with a set of assets for you to explore. The prototype implementation includes just some basic behaviors to spark your imagination: a large, animated level boss unit hovers in the chamber and produces a new kind of enemy robot that seeks out the hero character, spawning a new unit every few seconds.

517

518

519

520

521

522

523

524

Figure 12-10 shows a layout in the style you've been using to prototype basic mechanics.

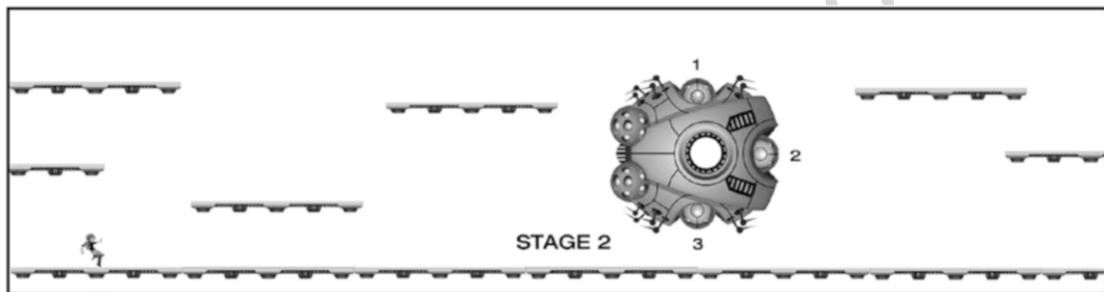


Figure 12-10. A possible second stage the hero character can enter after unlocking the door in stage 1. This concept includes a large “boss” unit with three nodes; one objective for this stage might be to disable each of the nodes to shut the boss down

It's a bit of a shortcut to begin the mechanic exploration with the diagram in Figure 12-10, but because you've already identified the setting and a number of visual elements, it can be helpful to continue developing new stages with some of the visual assets already in place. The diagram includes the same kind of platforms used in stage 1, but what if, for example, this area had no gravity and the hero character was able to fly freely? Compare this area with stage 1, and think about how you might slightly alter the experience to mix things up a bit without fundamentally changing the game; you've ideally become fairly fluent with the sequencing mechanic from stage 1, and the experience in stage 2 can be a greater or lesser evolution of that mechanic.

525

526

527

528

529

530

531

532

533

If you choose to include the hero-seeking flying robot units, the game flow diagram will become more complex than the model used in stage 1 because of the unpredictable movement of the new robot types. You may also want to consider a mechanism for the hero character to eliminate the robot units (perhaps even working the removal of robot

534

535

536

537

538 units into the mechanic for disabling the nodes on the boss). If you find your designs
539 becoming difficult to describe as part of an explicit and repeatable game flow, it may
540 signal that you're working with more complex systems and may need to evaluate them
541 in a playable prototype before you can effectively balance their integration with other
542 components of the level. Of course, you can also reuse conventions and units from
543 stage 1; you might choose to combine patrolling robots with hero-seeking robots and an
544 energy field, for example, creating a challenging web of potential risks for the player to
545 navigate as they work to disable the boss nodes.

546 You might also decide that the main objective for the level is to *enable* the boss nodes
547 in order to unlock the next stage or level of the game. You can extend the narrative in any
548 direction you like, so units can be helpful or harmful, objectives can involve disabling or
549 enabling, the hero character can be running toward something or away from something,
550 or any other possible scenario you can imagine. Remember, narrative development and
551 the level design will play on each other to drive the experience forward, so stay alert for
552 inspiration as you become increasingly fluent with the level designs for this prototype.

553 Summary

554 Game design is unique among the creative arts in the ways it requires players to become
555 active partners in the experience, which can change dramatically depending on who the
556 player is. Although some games share quite a bit in common with cinema (especially
557 as story-driven games become more popular), there's always an unpredictable element
558 when the player controls the on-screen action to a greater or lesser extent. Unlike movies
559 and books, video games are interactive experiences that demand constant two-way
560 engagement with players, and poorly designed mechanics or levels with unclear rules
561 can block players from enjoying the experience you've created.

562 The design methodology presented in this book focuses first on teaching you the
563 letters of the design alphabet (basic interactions), leading into the creation of words
564 (game mechanics and gameplay), followed by sentences (levels); we hope you'll take
565 the next step and begin writing the next great novel (full-game experiences in existing or
566 entirely new genres). The "escape the room" design template featured here can be used
567 to quickly prototype a wide range of mechanics for many kinds of game experiences,
568 from the included 2D side-scroller to isometric games to first-person experiences and
569 more. Remember, game mechanics are fundamentally well-formed abstract puzzles
570 that can be adapted as needed. If you find yourself having difficulty brainstorming new

mechanics in the beginning, borrow some simple existing mechanics from common casual games (“match 3” variants are a great source for inspiration) and start there, adding one or two simple variations as you go. As with any creative discipline, the more you practice the basics, the more fluent you’ll become with the process, and after you’ve gained some experience with simple mechanics and systems, you’ll likely be surprised by the number of interesting variations you can quickly create. Some of those variations might just contribute to the next breakthrough title. 571
572
573
574
575
576
577

This book demonstrates the relationship between the technical and experiential aspects of game design. Designers, developers, artists, and audio engineers must work in close partnership to deliver the best experiences, taking issues such as performance/ responsiveness, user inputs, system stability, and the like into consideration throughout production. The game engine you’ve developed in this book is well matched for the type of game described in this chapter (and many others). You should now be ready to explore your own game designs with a strong technical foundation to build upon and a global understanding of how the nine elements of game design work together to create experiences that players love. 578
579
580
581
582
583
584
585
586

Index

1

A

- Abstraction, JavaScript classes
 - objects project
 - client source code, 50
 - creation, 43
 - game engine, 44–50
 - goals, 43
 - source code organization, 43
 - observations, 51
 - activate() function, 418, 438, 451
 - addAsShadowCaster() function, 692
 - addShadowCaster() function, 508
 - aLight parameter, 506
 - Ambient light, *see* Global ambient project
 - Angular velocity, 590, 618, 620
 - Animated sprite sheet, 186, 233
 - Application programming interfaces (APIs), 3, 19
 - Asynchronous shader loading
 - resources for asynchronous support, 140, 141
 - SimpleShader modification, 141, 143
 - testing, 144
 - wait for asynchronous loading, 143
 - Audio
 - background audio, 171
 - component testing
 - BlueLevel.js, 181, 182
 - MyGame.js, 179, 180
 - design, 16
 - export module to client, 177

mAudioContext, 174

33

resource module, defining, 172–177

34

sound effects, 170

35

support project

36

controls, 171

37

goals, 172

38

running, 171

39

video games

40

aXform object, 341

41

Axis-aligned bounding boxes (AABBs),
294, 549

42
43

B

Background, Camera

44
45

engine development, 671

46

game engine, 698

47

layer manager Project, 689–695, 697

48

motion parallax, 698

49

motion parallax, parallax scrolling,
672, 681–683

50
51

ParallaxGameObject in MyGame,
687, 689

52
53

ParallaxObjects project, 683, 684, 686

54

side-scrolling game, 671

55

test tiled objects, 679, 680

56

TiledGameObject, 675, 677, 678

57

tiled objects project, 674, 675

58

tiling, 671, 673

59

visual complexity, 673

60

Bitmap fonts, 221, 245, 246

61

INDEX

62	BlueLevel scene, 168–170	
63	Bounding box	
64	class definition, 297, 299, 300	
65	and Collisions Project (<i>see</i> Collisions	
66	Project)	
67	definition, 353	
68	in engine, 300, 302	
69	testing with MyGame, 302, 303	
70	Brain GameObject, 291	
71	Broad phase method, 549, 563	
72	C	
73	Camera	
74	Camera Objects Project, 102–104	
75	testing, 108–110	
76	Camera object abstraction, 335, 336, 388	
77	Camera interpolations project	
78	CameraState class, 353, 355	
79	controls, 350	
80	goals, 350	
81	integrate camera manipulation	
82	operations, 355, 357	
83	Lerp Class, 351, 352	
84	LerpVec2 Class, 353	
85	in MyGame, 357	
86	running, 349	
87	Camera manipulation	
88	clamping to camera WC bounds,	
89	340, 341	
90	controls, 338	
91	convenient functions, 337	
92	HTML canvas, 335	
93	in MyGame, 344, 346, 347	
94	operations, camera_manipulation.js	
95	file, 342, 344	
96	running, 338	
97	source code, 339, 340	
	setUpViewProjection() function, 336	98
	source code, 337	99
	WC, 335	100
	zoom functions, 337	101
	Camera Objects Project	102
	Camera class, 103, 105, 106	103
	engine access file, 107	104
	goals, 103	105
	Renderable class, 107	106
	running, 102	107
	CameraShake abstraction, 366–368	108
	Camera Shake/Object Oscillate project	109
	camera modification to support shake	110
	effect, 366–368	111
	controls, 359	112
	goals, 360	113
	running, 359	114
	shake behavior	115
	Oscillate Class, 361, 362	116
	Shake class, 363	117
	ShakeVec2 class, 364, 365	118
	utilities, 360	119
	shaking effect, 365, 366	120
	testing in MyGame, 368, 369	121
	Camera transform	122
	goals, 94	123
	renderable modification, 96	124
	running, 93	125
	scene design, 97	126
	SimpleShader modification, 95, 96	127
	vertex shader modification, 95	128
	Camera utility functions, 423	129
	clampAtBoundary() function, 341, 342	130
	cleanup() function, 162, 163, 177	131
	Coefficient of friction, 607	132
	Coefficient of restitution, 607	133
	Collector.pixelTouches()	134
	function, 318	135

136	collideCircCirc() function, 556	
137	collideRectCirc() function, 580	
138	collideShape() function, 558	
139	Collision Angular Resolution project	
140	controls, 626	
141	goals, 627	
142	physics component, 628, 629, 631	
143	running, 626	
144	Collision detection	
145	between rectangles and circles, 576	
146	broad and narrow phase methods, 549	
147	Rectangle Collisions project, 569–576	
148	Separating Axis Theorem, 563–569	
149	CollisionInfo/Circle Collisions project	
150	CollisionInfo class, 552, 554	
151	controls, 551	
152	goals, 552	
153	MyGame modifications, 561, 562	
154	physics component, 558–560	
155	RigidCircle class, 555–558	
156	RigidRectangle class, 554	
157	running, 551	
158	Collision resolution, 599	
159	impulse method (<i>see</i> Impulse method)	
160	project	
161	controls, 614	
162	goals, 615	
163	MyGame for testing collision	
164	resolution, 618	
165	physics component, 615, 617	
166	running, 614	
167	Collisions Project	
168	controls of, 297	
169	goals of, 297	
170	running of, 296	
171	computeFaceNormals() function, 540	
172	_computeShadowGeometry()	
173	function, 504	
	Coordinate systems and transformations	174
	major axes, 90	175
	NDC (<i>see</i> Normalized Device	176
	Coordinates (NDC) System)	177
	origin, 90	178
	2D Cartesian, 90	179
	WC system (<i>see</i> World Coordinate	180
	(WC) System)	181
	_createParticle() function, 655, 668	182
	Cross product, 27, 284, 285	183
	D	184
	Damped simple harmonic motion, 358, 360	185
	defaultResources module, 403	186
	Design	187
	attacking enemy, 713	188
	basic narrative wrapper, 721, 722	189
	“blockout-plus” stage, 715	190
	bonus content, 723, 724	191
	challenge, 711, 712	192
	contextual images, 707, 708	193
	early-level prototype, 714	194
	“elevator” platform, 704	195
	experience, 724	196
	explicit steps, 705	197
	game audio, 717	198
	game systems and meta game, 718, 720	199
	horizontal “energy field”, 704	200
	interaction model, 718	201
	Playable Space, 708, 709, 711	202
	sci-fi setting, 706	203
	technical and experiential aspects, 725	204
	2D jumping and puzzle-solving	205
	mechanic, 704	206
	UI design, 720, 721	207
	visual design, 716	208
	well-formed mechanics, 703	209

INDEX

210	Development environment	248
211	Google Chrome web browser, 4	249
212	JavaScript syntax checker, 5	250
213	LiveServer installation, 5	251
214	VS code, 5	252
215	Diffuse reflection	
216	color texture, 445	253
217	human vision system, 443, 445	254
218	LightEffect() function, 448	255
219	normal/light vectors, 442, 443	256
220	normal mapping (<i>see</i> Normal mapping)	257
221	normal vectors of surfaces, 441	258
222	RGB channels, 444	259
223	static background, 456	260
224	surface normal vectors, 442	261
225	texture mapping, 444	262
226	texture uv coordinates, 449	263
227	3D objects, 443	264
228	uNormalSampler, 449	265
229		
230	Directional and Spotlights project	
231	Camera Transform class, 484	266
232	controls, 475	267
233	GLSL illumination fragment shader, 476, 478, 480	268
234	GLSL light fragment shader, 480	269
235	goals, 476	270
236	Light class, 481, 482	271
237	material property controls, 476	272
238	running, 475	273
239	ShaderLightAt class, 482, 484	274
240	testing, light types, 485	275
241		
242	DistanceDropOff() function, 463, 493	276
243	Dot product, 284, 612, 613	277
244	draw() function, 82, 96, 110, 122, 453	278
245	Drawing operations, <i>See also</i> Renderable Objects Project	279
246	encapsulation, 63, 64	280
247		281
	HTML5 Canvas	282
	creating and clearing, 21, 22, 24	
	project, 20, 21	
	subregions, 64	
	_drawTile() function, 677	
	E	
	Elementary drawing	254
	observations, 41	255
	one square project	256
	GLSL shaders, 33–39	257
	goals, 30	258
	primitive geometry data, 31–33	259
	running, 30	260
	Set Up, 39, 40	261
	set up with WebGL, 39, 40	262
	Enemies, 713, 714	263
	Explicit Euler Integration, 586	264
	External resources, 131, 132, 135	265
	F	
	findAxisLeastPenetration() function, 573	266
	First-person-shooter (FPS) genre, 718	267
	FontRenderable objects	268
	aString variable, 253	269
	bitmap fonts, 245, 246	270
	CharacterInfo, 250	271
	Consolas-72.png, 248	272
	draw() function, 253	273
	getCharInfo() function, 255	274
	initialize() function, 252	275
	initText() function, 259	276
	loadScene() function, 258	277
	minion_sprite.png, 248	278
	mOneChar variable, 253	279
	mOneChar.setElementUVCoordinate() function, 255	280
		281
		282

283	MyGame scene, 258, 261, 262	319
284	setTextHeight() function, 256	320
285	system-default-font, 248	321
286	unloadScene() function, 259	322
287	Font Support project	323
288	controls, 247	
289	default font adding, 251, 252	
290	FontRenderable object to draw	
291	texts, 253–256	
292	goals, 248	
293	initialize, cleaning, and export font	
294	functionality, 256, 258	
295	loading and storing fonts, 248–250	
296	running, 247	
297	testing, 258–262	
298	Fragment shader, 33, 35, 399	
299	Frames per second (FPS), 112	
300	Front and Chase Project	329
301	controls of, 286	
302	functionality	
303	Brain GameObject, 291	
304	MyGame Scene, 293, 294	
305	GameObject modification,	
306	287–289, 291	
307	goals of, 287	
308	running of, 286	
309	G	
310	Game design	333
311	activation, 520	
312	chiaroscuro techniques, 514	
313	dark environment, 516	
314	development teams, 522	
315	flashlight, 517, 518	
316	game environment, 515	
317	hero character, 515	
318	interaction model, 522	
	lighting, 515	334
	localized environment, 515	335
	logical consistency, 521	336
	metaphors, 519	337
	player controls, 516	338
	sequencing principles, 519	339
	Game engine, 1, 2, 12, 19	340
	core.js, 48–50	
	objects, 3	
	Shader Class, 44–48	
	Game loop	341
	component implementations, 115–119	
	implementations, 113, 114	
	working, 119–122	
	Game Objects Project	342
	chasing behavior, 280	
	definition, 270, 271	
	The DyePack GameObject, 274, 275	
	export classes, 273	
	goals of, 270	
	The Hero GameObject, 275	
	initialize()/update() functions, 269	
	The Minion GameObject, 276	
	MyGame Scene, 278–280	
	new sprite elements, minion_sprite.	
	png image, 270	
	running of, 269	
	set management, 272	
	WASD keys, 270	
	Generalized Per-Pixel Collisions Project	343
	axes-aligned texture, 319, 320	
	controls of, 321	
	GameObject_PixelCollision.js	
	modification, 324, 326	
	goals of, 321	
	rotated texture and component	
	vectors, 320	
	running of, 320, 321	

INDEX

Generalized Per-Pixel Collisions	
Project (<i>cont.</i>)	
357 testing, 326	393
358 TextureRenderable_PixelCollision	
359 modification, 322–324	394
360 vector review, 318, 319	395
361 getDefaultPSTexture() function, 648	396
362 get() function, 33, 207	397
363 getViewPort() function, 373	398
364 glMatrix Library, 77, 78	398
365 glMatrix math library, 4	399
366 Global ambient project	
367 controls, 401	400
368 global shared resources, 402	401
369 GLSL shaders, 401, 402	402
370 goals, 401	403
371 observations, 409	404
372 running, 400	405
373 SimpleShader modifications, 403, 404	406
374 testing, 405–409	407
375 GLSL LightShader, 411	408
376 GLSL shaders	
377 activateShader() function, 198	409
378 aTextureCoordinate attribute, 193	410
379 mShaderVertexPositionAttribute, 198	411
380 SimpleShader object, 197	412
381 texture2D()function, 195	413
382 TextureShader.js, 197	414
383 texture vertex, 192	415
384 TextureVS.gsls, 193	416
385 uPixelColor, 195	417
386 varying vTexCoord, 193	418
387 vertexShaderPath, 198	419
388 gl.uniformMatrix4fv()function, 81	420
389 gl.vertexAttribPointer() function, 38	421
390 gl.viewport() function, 92	422
391 Graphical user interface (GUI), 1, 14	423
392 Graphics processing unit (GPU), 3, 29	424
H	
Heads-up display (HUD) layer	425
Hero/Minion objects	426
Hero casting shadow	427
I	
Illumination model	428
IllumRenderable class	428
IllumShader class	428
Impulse method	428
normal component	428
with rotation	428
tangent component	428
terms	428
velocity components in	428
collision	428
velocity of colliding shapes	428
Inheritance	428
init() function	428
initWebGL() function	428
Integrated development environment	428
(IDE)	428
Interaction model	428
Interpenetration of colliding objects	428
Collision Position Correction project	428
controls	428
goals	428
physics component	428
running	428
testing positional correction in	428
MyGame	428
impulse method	428
penalty method	428
projection method	428
relaxation iteration	428
Interpolation	428
annoyance/confusion	428

429	camera parameters, 347	461
430	constant speed, 348	462
431	isKeyPressed() function, 126, 128, 129, 182	
432	J	
433	JavaScript, 2, 134, 135	
434	JavaScript source file project	
435	creation, 25–28	
436	load and run, 28	
437	running, 25	
438	JavaScript syntax checker, 5	
439	K	
440	Keyboard input support project	
441	add component to	
442	engine, 125–128	
443	controls, 124	
444	engine modification, 129, 130	
445	goals, 124	
446	running, 124	
447	testing, 130, 131	
448	KeyPress, 127	
449	L	
450	Layer Manager project	
451	controls, 691	
452	in engine, 691–694	
453	engine components and objects	
454	modifications, 694, 695	
455	MyGame modifications, 695–697	
456	running, 691	
457	Lerp class, 350–352	
458	LightEffect() function, 432, 433	
459	Lighting model, 398, 514	
460	LightRenderable class, 419, 420, 439	
530	Light shader project	461
531	camera modification	462
532	per-render cache, 423, 424	463
533	transform functions, adding,	464
534	424, 426	465
535	controls, 413	466
536	default LightShader	467
537	instance, 421, 422	468
538	GLSL light fragment shader, 413, 415	469
539	Light Class, defining, 415, 416	470
540	LightRenderable Class, 419, 420	471
541	LightShader Class, 417–419	472
542	observations, 427	473
543	parameters/material properties, 640	474
544	running, 412	475
545	testing, 426, 427	476
546	LiveServer extension, 5	477
547	loadAndCompileShader() function,	478
548	37, 41, 141	479
549	loadDecodeParse() function, 137, 146	480
550	load() function, 139	481
551	loadToShader() function, 419, 467, 483	482
552	loopOnce() function, 117, 595	483
553	loop.start() function, 156, 252	484
554	M	485
555	Mapping	486
556	normalized system, 187	487
557	resolution, 187	488
558	shaders (<i>see</i> Shaders project)	489
559	texel color, 187	490
560	texture space, 188	491
561	uv values, 187	492
562	Material class, 459, 465, 466	493
563	Material and Specularity project	494
564	Camera class, modifying, 470, 471	495
565	controls, 461	496

INDEX

Material and Specularity project (<i>cont.</i>)	
497 GLSL illumination fragment shader,	534
498 462, 463, 465	
499 goals, 462	535
500 IllumRenderable class, 469, 470	
501 IllumShader class, 468, 469	536
502 material property controls, 462	
503 running, 461	537
504 ShaderMaterial class, 466, 467	
505 testing specular reflection, 471	538
506 Math.ceil() function, 678	
507 Matrix operators	539
508 concatenation, 76	
509 glMatrix Library, 77, 78	540
510 4×4 identity matrix, 75	
511 m-rows by n-columns array of	541
512 numbers, 74	
513 rotation operator R(θ), 75	542
514 scaling operator S(sx, sy), 74, 75	
515 translation operator T(tx,ty), 74	543
516 4x1 vector, 75	
517 Matrix Transform Project	544
518 goals of, 79	
519 Renderable objects, 78	545
520 Renderable Object modification, 82	
521 running of, 78, 79	546
522 SimpleShader modification, 81	
523 testing, 82	547
524 vertex shader modification, 79, 80	
525 mCamera.collideWCBound()	548
526 function, 303	
527 mIsKeyClicked array, 127	549
528 Mouse input, cameras	
529 controls, 380	550
530 DC space, 378	
531 index.js, 380	551
532 init() function, 380, 383	
533 mouse button click handler, 382	552
mouse movement event handler, 382	
mouse support in input.js, 381–384	553
position in viewport DC space and WC	
space, 378	556
testing in MyGame, 386, 388	
viewport to WC space transform,	557
384, 386	
Movement	558
constant displacements, 585	
explicit Euler integration, 586	559
Rigid Shape Movements project	
fixed time interval, 595	560
goals, 589	
MyGame class modification, 597	561
RigidCircle class, 592, 593	
RigidRectangle class, 593	562
RigidBody class, 590–592	
running, 588	563
symplectic Euler integration,	
RigidBody class, 596, 597	564
system acceleration and motion	
control, 594	565
symplectic Euler integration, 587	
mShadowCaster, 502	566
Multiple Cameras project	
controls, 370	567
goals, 371	
object modification, 371–373	568
running, 370	
setViewAndCameraMatrix()	569
function, 373	
setViewport() function, 372	570
testing in MyGame, 374–377	
Multiple lights project	571
controls, 430	
GLSL light fragment	572
shader, 431–433	
goals, 431	573

572	Light class, modifying, 433, 434	
573	LightEffect() function, 432, 433	
574	light_fs fragment shader, 431	
575	LightRenderable class, 439	
576	LightSet class, 434	
577	LightShader class, modifying, 437–439	
579	ShaderLightAt class, 435, 436	
580	observations, 440	
581	running, 430	
582	testing, light sources, 439	
583	my_game.draw() function, 377, 511	
584	MyGame initialize() function, 679	
585	MyGame Scene, 278–280	
586	my_game_shadow.js file, 510	
587	MyGame.unload() function, 156	
588	MyGame update() function, 130	
589	N	
590	next() function, 169	
591	_nextValue() function, 363	
592	Normalized Device Coordinates (NDC) System, 90, 91	
593	Normal mapping	
594	color texture image, 444	
595	controls, 446	
596	default IllumShader instance, 454	
597	GLSL illumination fragment shader, 447, 449	
598	IllumRenderable class, 452, 453	
599	IllumShader class, 450, 451	
600	normal and color texture, 444, 445	
601	normal vectors of surfaces, 441	
602	running, 446	
603	testing, 454, 455	
604	texture module modification, 451	
605	Normal vector, 441–443	
540	O	608
541	Object-oriented abstraction, 42	609
542	Object-oriented design principles, 44	610
543	OpenGL Shading Language (GLSL), 3, 29	611
544	Parameterized Fragment Shader project, 58–62	612
545	Shader Source File project (see Shader Source File project)	613
546	source code organization, 57	614
547	vertex and fragment shaders, 34, 35	615
548	Oscillate class, 361, 362	616
549	Oscillation behavior, 363	617
550	P, Q	622
551	Panda3D, 1	623
552	Parallax scrolling, 681, 683	624
553	Parameterized Fragment Shader project drawing with new shader, 60, 61	625
554	running, 58	626
555	simple_fs.gsl Fragment Shader, 59	627
556	SimpleShader class, 59, 60	628
557	Particle Collisions project	629
558	goals, 657	630
559	particle system initializing, 661	631
560	testing, 662	632
561	particle system modification, 658–661	633
562	running, 657	634
563	ParticleEmitter class, 665, 666	635
564	Particle Emitters project	636
565	ParticleEmitter class, 665, 666	637
566	particle set modification, 667, 668	638
567	running, 664	639
568	testing, 668	640

INDEX

644	Particles project	682
645	default ParticleShader	
646	instance, 644–646	
647	engine particle component, 648	
648	GLSL particle fragment shader, 643, 644	
649	goals, 642	
650	loading default particle texture,	
651	647, 648	
652	particle creation, 649–651	
653	ParticleRenderable object, 646	
654	ParticleSet creation, 652, 653	
655	particle system testing, 653, 654	
656	running, 642	
657	Penalty method, 600	
658	Per-pixel collision algorithm, 306–308	
659	Per-Pixel Collisions Project, <i>See also</i>	
660	Sprite Pixel Collisions Project	
661	controls of, 305	
662	Engine_Texture component, 308, 310	
663	GameObject_PixelCollision.js, 316	
664	in GameObject, 316	
665	goals of, 306	
666	limitation with bounding box-based	
667	collision, 304	
668	in MyGame, 317, 318	
669	nontransparent pixels overlapping, 304	
670	running of, 305	
671	TextureRenderable modification	
672	access to texture color array,	
673	311, 313	
674	implementation, 313, 315, 316	
675	source code, 310, 311	
676	TextureRenderable_PixelCollision.js	
677	File, 314, 316	
678	PerRenderCache class, 423, 424	
679	Phong illumination model, 398, 399,	
680	459, 476	
681	Phong specularity model, 457	
	Physics engines, 526, 527, 606	682
	Physics simulations, games	683
	force field blocking access to upper	
	platforms, 637	
	platformer puzzle, 637	
	puzzle levels, 638	
	Rovio's Angry Birds requires	
	players, 634	
	UV light, 635	
	pixelStorei() function, 203	691
	pixelTouches() function, 316, 323	692
	Playable space, 708–710	693
	playCue() function, 175	694
	Procedural programming, 41	695
	processLoadedImage() function, 206	696
	Projection method, 599	697
	Prototype-based scripting	698
	languages, 3	
	Public methods, 144, 149	700
	R	701
	random() function, 654	702
	randomizeVelocity() function, 597	703
	Real-time strategy (RTS) game, 13	704
	Rectangle and Circle Collisions project	705
	checkCircRectVertex(), 579	
	collideRectCirc() function, 580	
	controls, 570, 577	
	goals, 571, 578	
	newly defined function, 583	
	RigidRectangle access file, 578	
	running, 570, 577	
	support point SAT	
	implementation, 571–575	
	_refPosUpdate() function, 686	715
	Relative velocity, rotation, 621	716
	Renderable architecture, 191	717

718	Renderable Objects Project	
719	goals of, 65	
720	matrix transformations, 73	
721	programmability and extensibility, 71	
722	running of, 65	
723	source code structure reorganization	
724	engine access file, 69, 70	
725	internal shader resource sharing, 68	
726	WebGL-specific module, 67	
727	testing, 72, 73	
728	requestAnimationFrame() function, 117	
729	resolveCollision() function, 628	
730	resolveRectPos() function, 659	
731	resolveRigidShapeCollision(), 660	
732	resolveRigidShapeSetCollision(), 660	
733	resource_map engine	
734	component, 134–138	
735	Resource map/shader loader project	
736	add component to engine, 134–138	
737	controls, 133	
738	goals, 134	
739	load shaders asynchronously	
740	shader resources modification,	
741	140, 141	
742	SimpleShader modification,	
743	141, 143	
744	testing, 144	
745	wait for asynchronous loading, 143	
746	running, 133	
747	text module defining, 138, 140	
748	resource_map loadDecodeParse()	
749	function, 139	
750	resource_map waitOnPromises()	
751	function, 141	
752	Restitution coefficient, 610	
753	Rezzing-in assets, 716	
754	Rigid body physics simulation, 527	
755	RigidCircle class, 542, 544, 555–558, 593	
	RigidRectangle class, 537, 538, 540–542, 554	756 757
	RigidShape Base class, 534, 535, 537	758
	Rigid Shapes/Bounds project	759
	controls, 530	760
	engine source code, 531	761
	GameObject class, 544–546	762
	gl-matrix library, 534	763
	goals, 531	764
	RigidCircle class, 542, 544	765
	RigidRectangle class, 537, 538, 540–542	766
	RigidShape Base class, 534, 535, 537	767
	running, 530	768
	supporting debug drawing, 532, 533	769
	testing, RigidShape functionality, 547	770
	Rigid square, continuous motion, 598	771
	rotateObjPointTo()	772
	function, 290, 294, 544	773
	S	774
	Scalable vector graphics (SVG), 3	775
	Scene file project	776
	client implementation	777
	MyGame implementation, 154–156	778
	parser, 151, 152, 154	779
	scene file, defining, 150	780
	client resource loading	781
	coordinate client load and engine	782
	wait, 148	783
	public interface, 149	784
	public methods, 149	785
	controls, 145	786
	goals, 145	787
	objects, 146	788
	running, 145	789
	XML resource module, 146, 147	790
	Scene.load() function, 217	791

INDEX

792	Scene Objects project	
793	Abstract Scene Class, 158–160	
794	engine cleanup support	
795	implementation, 161–164	
796	export Scene to client, 161	
797	game engine modification, 160	
798	running, 157	
799	testing, Scene Class interface	
800	BlueLevel scene, 168–170	
801	MyGame Scene, 164, 166, 168	
802	Scene.stop() function, 160	
803	Separating Axis Theorem (SAT)	
804	description, 563	
805	compute face normals, 564	
806	identify bounds, 565	
807	overlaps, 566	
808	project vertices, 565	
809	support points	
810	algorithm, 569	
811	face normal, 567	
812	least penetration and collision	
813	information, 568	
814	setCameraAndLight() function, 438	
815	setElementPixelPositions() function, 225	
816	setMass() function, 590	
817	setSpriteSequence() function, 244	
818	setViewAndCameraMatrix()	
819	function, 373	
820	setViewport() function, 372	
821	ShadedResults() function, 480	
822	Shader Class, 44–48	
823	ShaderLightAt class, 435, 436, 466, 467,	
824	482, 484	
825	ShaderMaterial class, 466, 467	
826	shader_resources init() function, 141	
827	Shader Source File project	
828	extracting shaders, 54, 55	
829	HTML Code cleaning up, 55, 56	
	loading shaders in	830
	SimpleShader, 53, 54	831
	running, 52	832
	Shaders project	833
	activateTexture() function, 202	834
	blendFunc() function, 203	835
	canvas.getContext(), 203	836
	createTexture() function, 206	837
	draw() function, 202	838
	Engine_DefaultResources, 190, 199	839
	gEngine_Texture.activateTexture(), 217	840
	gEngine_Texture._	841
	processLoadedImage(), 217	842
	GLSL, 189, 192	843
	initialize() function, 196	844
	initializeWebGL(), 203	845
	loadScene() function, 217	846
	pixelStorei() function, 203	847
	retrieveAsset() function, 207	848
	texParameteri() function, 208	849
	TextureRenderable class, 192, 201	850
	transparency, 188	851
	WebGL, 188, 189, 204	852
	TextureVS.glsl, 190	853
	unloadTexture() function, 205	854
	Shadow, 486, 488	855
	ShadowCaster class, 501–507	856
	Shadow caster geometry, 487, 506	857
	Shadow receiver, 487, 488	858
	Shadow Shaders project	859
	controls, 491	860
	GLSL shadow caster fragment	861
	shader, 493–495	862
	GLSL shadow receiver fragment	863
	shader, 495, 496	864
	goals, 492	865
	interfacing GLSL	866
	shadow shaders	867

868	default shadow caster and receiver	906
869	shaders, 498, 499	
870	shadow caster shader, creation,	907
871	497, 498	
872	material property controls, 492	908
873	running, 491	
874	Shadow Caster class,	909
875	defining, 501–507	
876	Shadow Receiver class,	910
877	defining, 507–509	
878	testing, shadow algorithm, 510–512	911
879	updating engine support, 509	
880	WebGL buffers, configuring and	912
881	supporting, 499, 501	
882	Shadow simulation algorithm, 488, 490	913
883	ShakeVec2 class, 364, 365	
884	Shaking effect, 358	914
885	Sound effects, 170	
886	Specular reflection	915
887	ambient lighting, 458	
888	computation cost, 458	
889	diffuse lighting, 456	
890	halfway vector, 458	
891	hero character, 461	
892	integration, material, 459, 460	
893	light source, 457, 458	
894	materials property, 458	
895	Phong illumination model, 456	
896	Phong lighting computation, 460	
897	real-life experience, 457	
898	reflection direction, 457	
899	shininess, 456	
900	shiny surface, 457	
901	testing, 471	
902	uCameraPosition, 462	
903	Spotlight	934
904	angular attenuation, 479	
905	models, 473	
	parameters, 474	935
	project, 479	
	Sprite animations	936
	constructor, 238	
	eAnimateSwing, 237	
	eAnimationType, 237	
	export functionality to client, 242	
	game developer, 233	
	GameLoop _runLoop() function, 241	
	_initAnimation() function, 238	
	initialize() function, 243	
	mCurrentAnimAdvance, 238	
	mFontImage, 233	
	retraction, 234	
	setSpriteElement(), 239	
	setSpriteSequence() function, 244	
	SpriteAnimateRenderable, 236	
	testing, 243–245	
	texture-mapping, 233	
	transformation operators, 233	
	updateAnimation() function, 241	
	Sprite Pixel Collisions Project	927
	goal of, 327	
	in MyGame, 330	
	running of, 326, 327	
	SpriteRenderable_PixelCollision.js.	
	creation, 329	
	TextureRenderable modification, 329	
	Sprite sheets	933
	draw() function, 226	
	dye character, 218	
	Engine_DefaultResources.js, 226	
	Engine_VertexBuffer.js, 221	
	getGLTexCoordRef() function, 221	
	GLSL texture shaders, 222, 223	
	initialize() function, 228	
	minion_sprite.png, 218, 221, 229	
	model space, 219	

INDEX

Sprite sheets (<i>cont.</i>)		
944 MyGame.js file, 228		
945 pixel locations, 218		
946 source code, 220		
947 uv values, 218		
948 setElementPixelPositions() functions,		
949 225, 229		
950 setElementUVCoordinate(), 232		
951 SpriteRenderable objects, 221		
952 texCoord, 223		
953 start() function, 143, 661		
954 Support point algorithm, 576		
955 Symplectic Euler integration, 587		
T		
956 texParameteri() function, 208		
957 Texture coordinates		
958 bitmap fonts, 263		
959 custom-composed images, 185		
960 game design, 263, 264, 266		
961 loading and unloading, 186		
962 rendering, 185		
963 WebGL, 186		
964 Texture mapping functionality		
965 BlueLevel.xml scene, 210		
966 BlueScene scene, 210		
967 draw() function, 216		
968 initialize() functions, 211, 215		
969 loadScene(), 211		
970 minion_collector.jpg, 211		
971 MyGame constructor, 214		
972 ResourceMap, 216		
973 SceneFileParser.js, 211		
974 TextureRenderable object, 213		
975 TextureSquare element, 210		
976 unloadScene() function, 212		
977 update() function, 213, 216		
U		
991 Ubuntu, 4		
992 Unix-based environment, 4		
993 unloadScene() function, 696		
994 updateAnimation() function, 241		
995 update() function, 119, 121, 128, 662		
996 updateInertia() function, 593		
997 uPixelColor variable, 59		
998 User interface (UI) design, 720, 721		
V		
1000 vec2.lerp() function, 353		
1001 vec2.rotateWRT() function, 534		
1002 Vectors		
1003 cross product, 284, 285		
1004 definition, 281, 282		
1005 description, 281		
1006 dot product, 284		
1007 normalized, 282		
1008 object movements and behaviors, 280		
1009 object's velocity/acceleration, 281		
1010 origin to position being rotated by		
1011 angle theta, 283		
1012 angle theta, 283		

1013	2D space with two vectors equal to each other, 283	
1014		
1015	VertexShader, 34	
1016	Video games	
1017	game environments, 184	
1018	responsiveness, 183	
1019	shader Loads project, 184	
1020	translation errors, 183	
1021	Viewport	
1022	description, 92	
1023	working with, 92, 93	
1024	View-Projection transform, <i>See also</i>	
1025	Camera	
1026	goals of, 94	
1027	RenderObject modification, 96	
1028	running of, 93, 94	
1029	SimpleVertex modification, 96	
1030	testing	
1031	design implementation, 98, 100, 101	
1032		
1033	scene designing, 97	
1034	vertex shader modification, 95	
1035	Visceral Games' Dead Space 3, 721	
1036	Visual Studio Code (VS Code) IDE, 4	
	VS code	1037
	HTML5 project creation, 6, 8, 10	1038
	IDE, 6	1039
	W	1040
	wcDirToPixel() function, 423, 484	1041
	WebGL, elementary drawing, <i>see</i>	1042
	Elementary drawing	1043
	WebGL stencil buffer, 488, 489	1044
	WebGL texture maps, 3	1045
	Web Graphics Library (WebGL), 3	1046
	World Coordinate (WC) System	1047
	description, 91	1048
	working with, 91	1049
	working with", 91	1050
	X, Y	1051
	XML-encoded scene file, 146, 147	1052
	Z	1053
	zoomBy() function, 344	1054