

THE EXPERT'S VOICE® IN GAME DEVELOPMENT

Build your own 2D Game Engine and Create Great Web Games Using HTML5, JavaScript, and WebGL

*DEVELOP YOUR OWN GAME ENGINE AND LEARN HOW TO
CREATE WELL-DESIGNED GAME EXPERIENCES THAT ARE
FUN TO BUILD AND EVEN MORE FUN TO PLAY!*

Kelvin Sung, Jebediah Pavleas, Fernando Arnez, and Jason Pace

Apress®

www.allitebooks.com

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

Using HTML5, JavaScript, and WebGL



Kelvin Sung
Jebediah Pavleas
Fernando Arnez
Jason Pace

With

Original Dye Characters Design: **Nathan Evers**

Other Game Character design and game arts: **Kasey Quevedo**

Figures and Illustration: **Clover Wai**

Apress®

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

Copyright © 2015 by Kelvin Sung, Jebediah Pavleas, Fernando Arnez, 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. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.

All visual and audio assets included with the sample projects in this book are protected by the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 license (CC BY-NC-SA 3.0) <http://creativecommons.org/licenses/by-nc-sa/3.0/>. You may adapt and share the materials and create derivative works, but you may not use the material for commercial purposes and all derivative works must be distributed under the CC BY-NC-SA 3.0 license.

ISBN-13 (pbk): 978-1-4842-0953-0

ISBN-13 (electronic): 978-1-4842-0952-3

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: Welmoed Spahr

Lead Editor: Ben Renow-Clarke

Technical Reviewer: Jason Sturges

Editorial Board: Steve Anglin, Mark Beckner, Gary Cornell, Louise Corrigan, Jim DeWolf,

Jonathan Gennick, Jonathan Hassell, Robert Hutchinson, Michelle Lowman, James Markham,

Susan McDermott, Matthew Moodie, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke,

Gwenan Spearing, Matt Wade, Steve Weiss

Coordinating Editor: Melissa Maldonado

Copy Editor: Kim Wimpsett

Compositor: SPi Global

Indexer: SPi Global

Artist: SPi Global

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springer.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 rights@apress.com, or visit www.apress.com.

Apress and friends of ED books 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 Special Bulk Sales-eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this text is available to readers at www.apress.com. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/.

To my wife, Clover, and our girls, Jean and Ruth, for completing my life.

—*Kelvin Sung*

To my nieces, Marley and Monroe, for the utter joy and inspiration they bring to those around them.

—*Jebediah Pavleas*

To my parents whose support carried me so many times, and my grandmother for being forever understanding.

—*Fernando Arnez*

To my husband, Craig, and my mother, Linda, for their constant patience, support, and encouragement.

—*Jason Pace*

Contents at a Glance

About the Authors.....	xv
About the Technical Reviewer	xvii
Acknowledgments	xix
Introduction	xxi
■ Chapter 1: Introducing 2D Game Engine Development with JavaScript.....	1
■ Chapter 2: Working with HTML5 and WebGL	15
■ Chapter 3: Drawing Objects in the World	47
■ Chapter 4: Implementing Common Components of Video Games	77
■ Chapter 5: Working with Textures, Sprites, and Fonts	127
■ Chapter 6: Defining Behaviors and Detecting Collisions	187
■ Chapter 7: Manipulating the Camera.....	233
■ Chapter 8: Implementing Illumination and Shadow	273
■ Chapter 9: Integrating Physics and Particles	365
■ Chapter 10: Supporting Camera Background	419
■ Chapter 11: Building a Sample Game: From Design to Completion.....	441
Index.....	457

Contents

About the Authors.....	xv
About the Technical Reviewer	xvii
Acknowledgments	xix
Introduction	xxi
■ Chapter 1: Introducing 2D Game Engine Development with JavaScript.....	1
The Technologies.....	2
Setting Up Your Development Environment.....	3
Downloading and Installing JavaScript Syntax Checker.....	3
Working in the NetBeans Development Environment.....	4
Creating an HTML5 Project in NetBeans.....	5
The Relationship Between the Project Files and the File System.....	8
How to Use This Book.....	9
How Do You Make a Great Video Game?	9
References	13
Technologies.....	13
■ Chapter 2: Working with HTML5 and WebGL	15
Introduction	15
Canvas for Drawing	15
The HTML5 Canvas Project.....	15
Separating HTML and JavaScript	18
The JavaScript Source File Project.....	18
Observations.....	23

■ CONTENTS

Elementary Drawing with WebGL	23
The Draw One Square Project.....	23
Observations.....	30
Abstraction with JavaScript Objects	30
The JavaScript Objects Project.....	30
Observations.....	39
Separating GLSL from HTML	39
The Shader Source Files Project.....	39
Source Code Organization	43
Changing the Shader and Controlling the Color	43
The Parameterized Fragment Shader Project.....	43
Summary	45
■ Chapter 3: Drawing Objects in the World	47
Introduction	47
Encapsulating Drawing	48
The Renderable Objects Project	48
Observations.....	50
Transforming a Renderable Object	51
Matrices as Transform Operators	51
The glMatrix Library	53
The Matrix Transform Project	54
Observations.....	58
Encapsulating the Transform Operator	58
The Transform Objects Project.....	58
The Transform Object.....	59
View, Projection, and Viewports	61
Coordinate Systems and Transformations	61
The View Projection and Viewport Project.....	64

The Camera	71
The Camera Objects Project	71
Summary.....	76
■ Chapter 4: Implementing Common Components of Video Games	77
Introduction	77
The Game Loop	78
The Game Loop Project.....	79
Keyboard Input	86
The Keyboard Support Project.....	86
Resource Management and Asynchronous Loading	93
The Resource Map and Shader Loader Project	94
Game Level from a Scene File.....	103
The Scene File Project.....	103
Scene Object: Client Interface to the Game Engine.....	110
The Scene Objects Project.....	110
Audio	116
The Audio Support Project.....	117
Summary.....	125
Game Design Considerations.....	125
■ Chapter 5: Working with Textures, Sprites, and Fonts	127
Introduction	127
Texture Mapping and Texture Coordinates	128
The Texture Shaders Project.....	130
Drawing with Sprite Sheets.....	150
The Sprite Shaders Project.....	152
Sprite Animations	161
Overview of Animated Sprite Sheets	162
The Sprite Animation Project.....	163

Fonts and Drawing of Text.....	170
Bitmap Fonts	170
The Font Support Project.....	171
Summary.....	184
Game Design Considerations.....	184
■Chapter 6: Defining Behaviors and Detecting Collisions	187
Introduction	187
Game Objects	188
The Game Objects Project	188
Chasing of a GameObject.....	196
Vectors Review	196
The Front and Chase Project	200
Collisions Between GameObjects.....	207
Bounding Box	207
The Bounding Box and Collisions Project	208
Per-Pixel Collisions.....	213
The Per-Pixel Collisions Project.....	213
Generalized Per-Pixel Collisions	221
Vector Review: Components and Decomposition	221
The General Pixel Collisions Project	224
Per-Pixel Collisions for Sprites	227
The Sprite Pixel Collisions Project	228
Summary.....	231
Game Design Considerations.....	231
■Chapter 7: Manipulating the Camera.....	233
Introduction	233
Camera Manipulations	234
The Camera Manipulations Project.....	234
Interpolation	240
The Camera Interpolations Project	241

Camera Shake Effect.....	246
The Camera Shake Project	247
Multiple Cameras	253
The Multiple Cameras Project.....	254
Mouse Input Through Cameras	259
The Mouse Input Project.....	260
Summary.....	266
Game Design Considerations.....	267
■ Chapter 8: Implementing Illumination and Shadow	273
Introduction	273
Overview of Illumination and GLSL Implementation	274
Ambient Light.....	274
The Global Ambient Project	275
Light Source	281
GLSL Implementation and Integration into the Game Engine	282
The Simple Light Shader Project	283
Multiple Light Sources and Distance Attenuation	293
The Multiple Lights Project.....	294
Diffuse Reflection and Normal Mapping	302
The Normal Maps and Illumination Shaders Project	305
Specular Reflection and Materials	315
Integration of Material in the Game Engine and GLSL Shaders.....	318
The Material and Specularity Project	318
Light Source Types	327
The Directional and Spot Lights Project	328
Shadow Simulation	337
The Shadow Simulation Algorithm	339
The Shadow Shaders Project.....	340
Summary.....	356
Game Design Considerations.....	356

■ Chapter 9: Integrating Physics and Particles	365
Introduction	365
Physics Overview	366
Movement.....	366
Collision Detection.....	368
Collision Resolution	369
Detecting Collisions.....	369
The Rigid Shape Bounds Project	370
Resolving Collisions	381
The Rigid Shape Impulse Project.....	381
Particles and Particle Systems.....	397
The Particles Project.....	397
Particle Emitters.....	408
The Particle Emitters Project.....	408
Summary.....	412
Game Design Considerations.....	413
■ Chapter 10: Supporting Camera Background	419
Introduction	419
Tiling of the Background	420
The Tiled Objects Project.....	421
Simulating Motion Parallax with Parallax Scrolling.....	426
The ParallaxObjects Project.....	428
Layer Management	432
The Layer Manager Project.....	432
Summary.....	438
Game Design Considerations.....	439

■ Chapter 11: Building a Sample Game: From Design to Completion	441
Part 1: Refining the Concept.....	441
Part 2: Integrating a Setting	444
Contextual Images Bring the Setting to Life	444
Defining the Playable Space.....	445
Adding Layout to the Playable Space	446
Tuning the Challenge and Adding Fun	447
Further Tuning: Introducing Enemies.....	448
General Considerations.....	448
Part 3: Integrating Additional Design Elements	449
Visual Design	449
Game Audio	450
Interaction Model.....	451
Game Systems and Meta Game.....	451
User Interface (UI) Design.....	452
Game Narrative.....	453
Bonus Content: Adding a Second Stage to the Level.....	454
Summary.....	455
Index.....	457

About the Authors

Kelvin Sung is a professor with the Computing and Software Systems Division and the principal investigator of the Game-Themed Research Group at University of Washington Bothell (UWB). He received his Ph.D. in computer science from the University of Illinois at Urbana-Champaign in 1992. His background is in computer graphics, hardware, and machine architecture. He came to UWB from Alias|Wavefront (now part of Autodesk) in Toronto, where he played a key role in designing and implementing the Maya renderer, an Academy Award-winning image generation system. Before joining Alias|Wavefront, Kelvin was an assistant professor with the School of Computing, National University of Singapore. Kelvin's research interests are in studying the role of technology in supporting human communication. Funded by Microsoft Research and the National Science Foundation, Kelvin's recent work focused on the intersection of video game mechanics, real-world problems, and mobile technologies. Kelvin teaches both undergraduate and graduate classes in computer graphics, game development, and mobile computing.

Jebediah Pavleas is a graduate student in the Computer Science and Software Engineering program at the University of Washington Bothell (UWB) as well as an intern on the NExT Enable team at Microsoft Research. He is also the coauthor of the book *Learn 2D Game Development with C#*. He received a bachelor's of science degree in 2012 and was the recipient of the Chancellor's Medal for his class. During his time as an undergraduate, he took great interest in both computer graphics and games. His projects included an interactive math application that utilizes Microsoft's Kinect sensor to teach algebra, a 2D role-playing game designed to teach students introductory programming concepts, and a web site where students can compete in various mini-games to control checkpoints around campus. Relating to these projects, he coauthored publications in IEEE Computers and The Journal of Computing Sciences in Colleges (CCSC). When not working toward his graduate degree, he enjoys designing, building, and playing games of all kinds as well as adapting technology for improved accessibility.

Fernando Arnez is an undergraduate student in the Computing and Software Systems Division at the University of Washington Bothell (UWB) working toward his bachelor's degree in computer science and software engineering. He is a member of the Game-Themed Research Group and has participated in projects that built casual games for teaching introductory programming concepts. He coauthored an article in IEEE Computers discussing his work and the results from those projects.

Jason Pace directs the University of Washington Bothell's Digital Future Lab (www.bothell.washington.edu/digitalfuture/about), an interactive media research and development studio modeling startup culture for a diverse group of student developers, designers, artists, musicians, and producers. He started the lab after spending 16 years at Microsoft leading user experience and product development teams on a number of Microsoft's key consumer products, including serving as a creative director and lead producer on the Halo team at 343 Industries and Design Manager for the Microsoft Casual Games group. His work in the lab focuses on exploring how radically diverse teams that seek to maximize differences among contributors can lead to unexpected insights and new directions in design and development. The lab brings undergraduate students together from across majors and schools to create high-performance creative teams engaged in both commercial game development and design research.

About the Technical Reviewer

Jason Sturges is a cutting edge technologist focused in ubiquitous delivery of immersive user experiences. Coming from a visualization background, he's always pushing boundaries of computer graphics to the widest reach cross platform while maintaining natural and intuitive usability per device. From interactivity, motion, animations, and creative design, he has worked with numerous creative agencies on projects from kiosks to video walls to Microsoft Kinect games. Most recently the core of his work has been mobile apps.

Committed to the open source community, he is also a frequent contributor at GitHub and Stack Overflow as a community resource leveraging modern standards, solid design patterns, and best practices in multiple developer tool chains for web, mobile, desktop, and beyond.

Acknowledgments

This book project is 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.

The invaluable collaboration between the technical team in the Game-Themed Research Group (<https://depts.washington.edu/cmmr/GTCS/>) and the design team in the Digital Future Lab (<http://www.bothell.washington.edu/digitalfuture>) at the University of Washington Bothell, where much of our learning occurred during the production of the many casual games for teaching introductory programming concepts, formed the foundation that allowed the development of this book. Thank you to all the participants of this research undertaking, especially to Mike Panitz, Rob Nash, Brian Hecox, Emmett Scout, Nathan Evers, Cora Walker, and Aina Braxton for working with us throughout all these years. The authors would also like to thank the students at the University of Washington Bothell for the games they built from the course CSS385: Introduction to Game Development (see <http://courses.washington.edu/css385>). Their interest and passion for games has provided us with the ideal deployment vehicle and are a source of continuous inspiration. They have tested, retested, contributed to, and assisted in the formation and organization of the contents of this book.

Jebediah Pavleas would like to thank the Computing and Software Systems Division at the University of Washington Bothell for the generous tuition scholarships that funded his education throughout his participation with this book project.

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 composed the background music used in the Audio Support project from Chapter 4, originally for the Linx game, which was designed to teach loops. The background audio for the game in Chapter 11 was composed by David Madden and arranged by Aina Braxton. Thanks to Clover Wai for the figures and illustrations.

We also want to thank Gwenan Spearing at Apress for connecting us to our editor Ben Renow-Clarke. A heartfelt thank-you to Kevin Walter for his patient and diligent organization skills in guiding us, to Melissa Maldonado for tolerating and working with our constantly behind schedule frenzy, and to Kim Wimpsett for the tireless and excellent edits that make much of this book actually readable. Finally, we would like to thank Jason Sturges for his insightful technical feedback.

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

Introduction

Welcome to *Build Your Own 2D Game Engine and Create 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 WebGL, 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.

Who Should Read This Book

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

Assumptions

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

Who Should Not Read This Book

This book is not designed to teach readers how to program, nor does it attempt to explain the intricate details of HTML5, JavaScript, or WebGL. 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.

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.

Organization of This Book

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.

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.

Chapters 2 to 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, WebGL 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 WebGL 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.

Chapters 5 to 7 present 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.

Chapters 8 to 10 elevate the introduced functionality to more advanced levels. Chapter 8 covers the simulation of 3D illumination effects in 2D game scenes. Chapter 9 discusses physically based behavior simulations and particle systems that are suitable for modeling explosions. Chapter 10 examines more advanced camera functionality including infinite scrolling through tiling and parallax.

Chapter 11 summarizes the book by leading you through the design of a complete game based on the game engine you have developed.

Code Samples

Every chapter in this book includes examples that let you interactively experiment with and learn the new materials. You can download the source code for all the projects, including the associated assets (images, audio clips, or fonts), from the following page: www.apress.com/9781484209530.

Follow the instructions to download the 9781484209530.zip file. To install the code samples, unzip the 9781484209530.zip file. You should see a folder structure that is organized by chapter numbers. Within each folder are subfolders containing NetBeans projects that correspond to sections of this book.

CHAPTER 1



Introducing 2D Game Engine Development with JavaScript

Video games are complex, interactive, multimedia software systems. These systems must, in real time, process player input, simulate the interactions of semi-autonomous objects, and generate high-fidelity graphics and audio outputs, all while trying to engage the players. Attempts at building video games can quickly be overwhelmed by 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 and supporting 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 the game developers from 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 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 shadows 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 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 the book. The discussion then 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 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.

Note JavaScript supports inheritance via the language prototype chain mechanism. Technically, there is no class hierarchy to speak of. However, for clarity and simplicity, this book uses standard object-oriented terminology such as *superclass* and *subclass* to refer to parent-child inheritance relationships.

Setting Up Your Development Environment

The game engine you are going to build will be accessible through web browsers that could be running on any operating system (OS). The development environment you are about to set up is also OS agnostic. For simplicity, the following instructions are based on a Windows 7 or Windows 8 OS. You should be able to reproduce a similar environment with minor modifications in a Unix-based environment like the Apple OS X or Ubuntu.

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

- *IDE:* All projects in this book are based on the NetBeans IDE. You can download and install the bundle for HTML5 and PHP from <https://netbeans.org/downloads>.
- *Runtime environment:* You will execute your video game projects in the Google Chrome web browser. You can download and install this browser from <https://www.google.com/chrome/browser/>. Notice that Microsoft Internet Explorer 11 does not support HTML5 AudioContext and thus will not execute any projects after Chapter 4; in addition, Mozilla Firefox (version 39.0) does not support some of the GLSL shaders in Chapter 9.
- *Connector Google Chrome plug-in:* This is a Google Chrome extension that connects the web browser to the NetBeans IDE to support HTML5 development. You can download and install this extension from <https://chrome.google.com/webstore/detail/netbeans-connector/hafdlehgocfcodbgjnpecfajgkeejnaa>. The download will automatically install the plug-in to Google Chrome. You may have to restart your computer to complete the installation.
- *glMatrix math library:* This is a library that implements the foundational mathematical operations. You can download this library from <http://glMatrix.net>. You will integrate this library into your game engine in Chapter 3, so more details will be provided there.

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

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

Downloading and Installing JavaScript Syntax Checker

We have found JSLint to be an effective tool in detecting potential JavaScript source code errors. You can download and install JSLint as a plug-in to the NetBeans IDE with the following steps:

- Download it from <http://plugins.netbeans.org/plugin/40893/jshint>. Make sure to take note of the location of the downloaded file.
- Start NetBeans, select Tools ▶ Plugins, and go to the Downloaded tab.
- Click the Add Plugins button and search for the downloaded file from step 1. Double-click this file to install the plug-in.

The following are some useful references for working with JSLint:

- For instructions on how to work with JSLint, see <http://www.jslint.com/help.html>.
- For details on how JSLint works, see <http://plugins.netbeans.org/plugin/40893/jslint>.

Working in the NetBeans Development Environment

The NetBeans IDE is easy to work with, and the projects in this book require only the editor and debugger. To open a project, select File ➤ Open Projects. Once a project is open, you need to become familiarize with three basic windows, as illustrated in Figure 1-1.

- *Projects window*: This window displays the source code files of the project.
- *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 double-clicking the corresponding file name in the Projects window.
- *Action Items window*: This window displays the error message output from the JSLint checker.

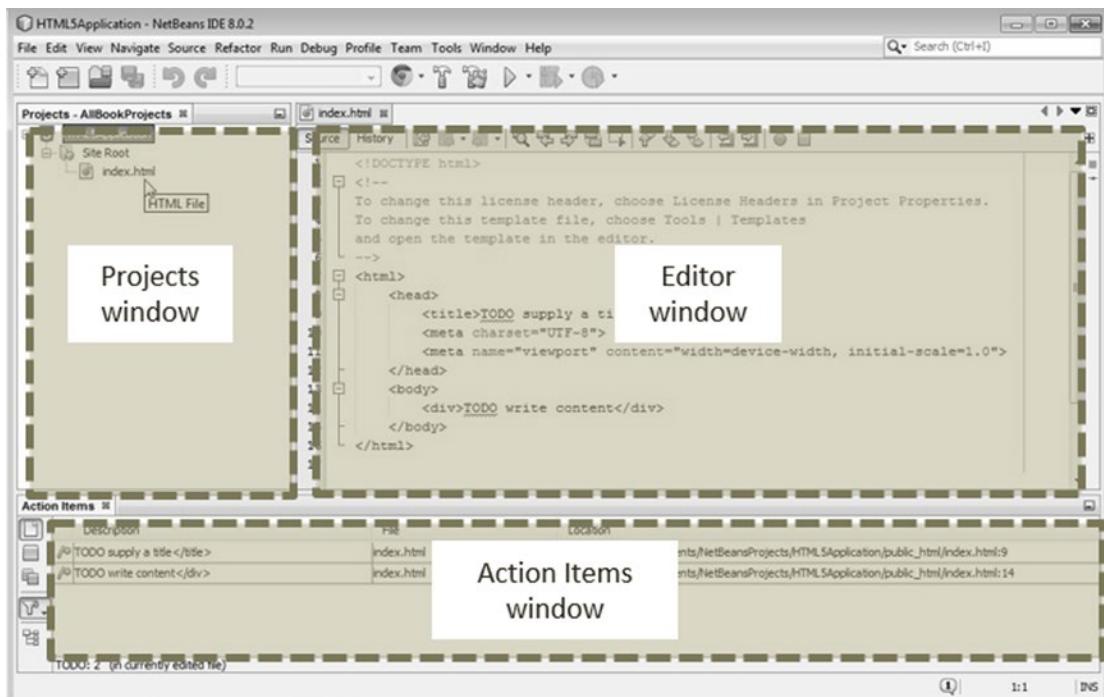


Figure 1-1. The NetBeans IDE

Note If you cannot see a window in the IDE, you can click the Window menu and select the name of the missing window to cause it to appear. For example, if you cannot see the Projects window in the IDE, you can select Window ▶ Projects to open it.

Creating an HTML5 Project in NetBeans

You are now ready to create your first HTML5 project.

- Start NetBeans. Select File ▶ New Project (or press Ctrl+Shift+N), as shown in Figure 1-2. A New Project window will appear.

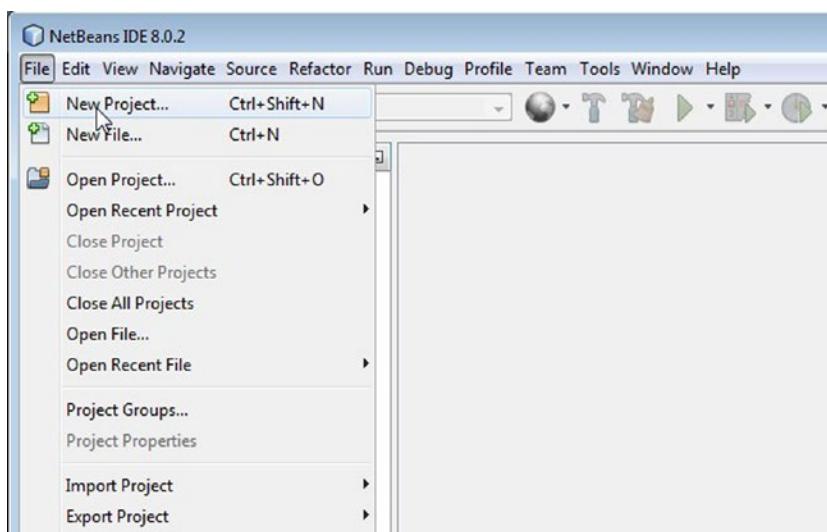


Figure 1-2. Creating a new project

- In the New Project window, select HTML5 in the Categories section, and select HTML5 Application from the Projects section, as shown in Figure 1-3. Click the Next button to bring up the project configuration window.

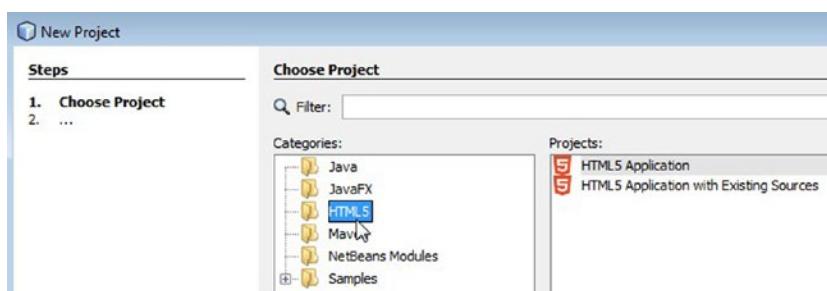


Figure 1-3. Selecting the HTML5 project

- As shown in Figure 1-4, enter the name and location of the project, and click the Finish button to create your first HTML5 project.

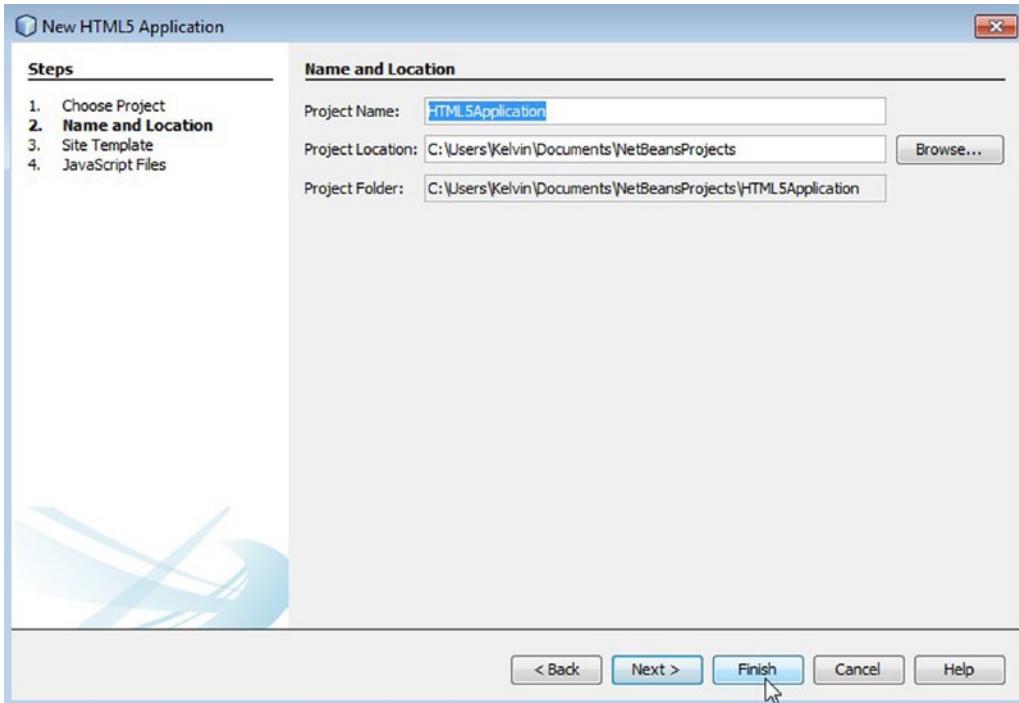


Figure 1-4. Naming the project

NetBeans will generate the template of a simple and complete HTML5 application project for you. Your IDE should look similar to Figure 1-5.

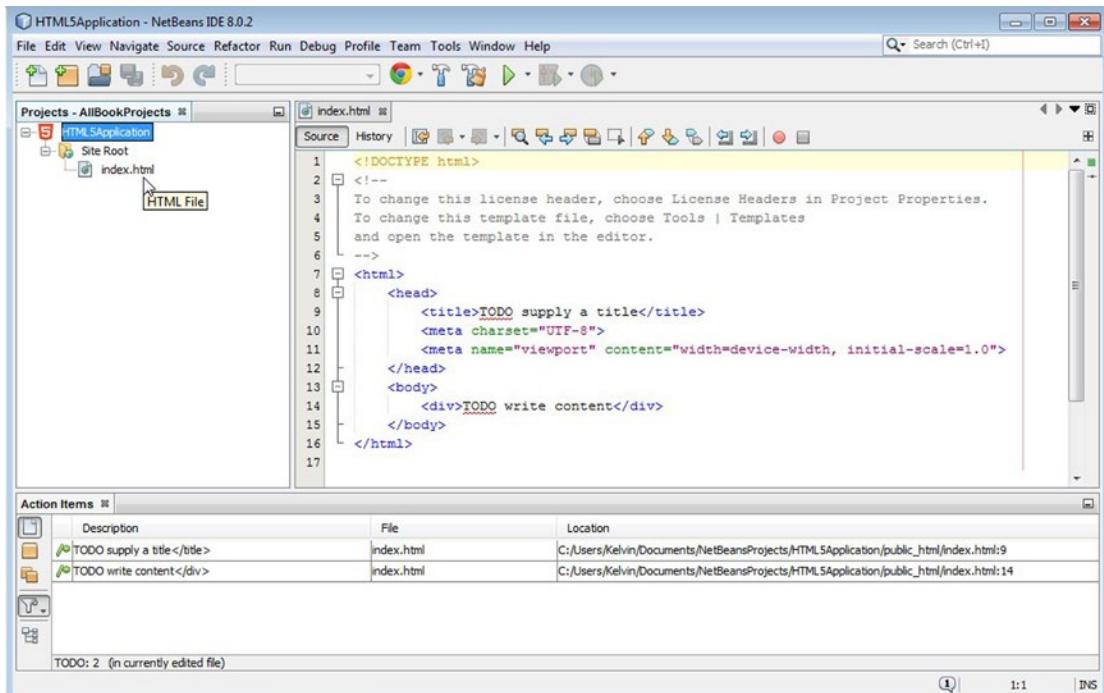


Figure 1-5. The HTML5 application project

By selecting and double-clicking the `index.html` file in the Projects window, you can open it in the Editor window and observe the content of the file. The contents are as follows:

```
<!DOCTYPE html>
<!--
To change this license header, choose License Headers in Project Properties.
To change this template file, choose Tools | Templates
and open the template in the editor.
-->
<html>
    <head>
        <title>TODO supply a title</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
    </head>
    <body>
        <div>TODO write content</div>
    </body>
</html>
```

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

You can run this project by selecting Run ▶ Run Project or by pressing the F6 key. Figure 1-6 shows an example of what the default project looks like when you run it.

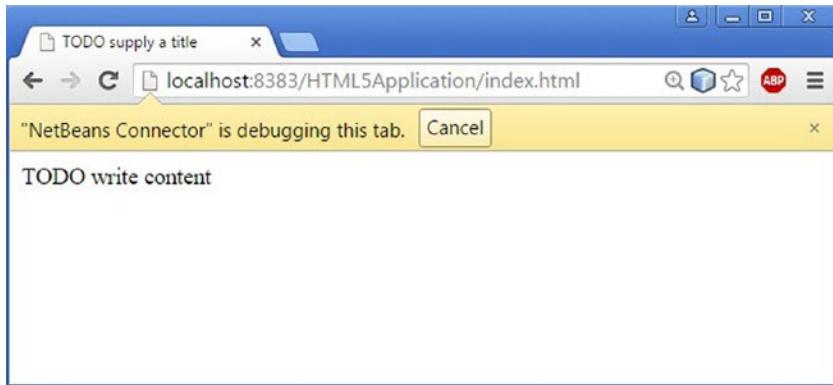


Figure 1-6. Running the simple HTML5 project

Note As will be explained in the next chapter, you cannot double-click the `index.html` file to run the project. You must either select Run ▶ Run Project, press on the F6 key, or click the green triangle button.

To stop the program, either close the web page or click the Cancel button in the browser to stop NetBeans from tracking the web page. You have successfully run your first HTML5 project. You can use this project to understand the IDE environment.

The Relationship Between the Project Files and the File System

Navigate to the `HTML5Application` project location on your file system, for example with the Explorer OS utility in Windows 7. You can observe that in the project folder NetBeans has generated the `nbProject`, `public_html`, and `test` folders. Table 1-1 summarizes the purpose of these folders and the `index.html` file.

Table 1-1. Folders and Files in a NetBeans HTML5 Project

NetBeans HTML5 project: folder/file	Purpose
<code>nbProject/</code>	This folder contains the IDE configuration files. You will not modify any of the files in this folder.
<code>public_html/</code>	This is the root folder of your project. Source code and assets from your project will be created in this folder.
<code>public_html/index.html</code>	This is the default entry point for your web site. This file will be modified to load JavaScript source code files.
<code>test/</code>	This is the default folder for unit testing source code files. This folder is not used in this book and has been removed from all the projects.

How to Use This Book

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

The projects start with demonstrating simple concepts, such as drawing a simple square, but evolve quickly into presenting more complex concepts, such as working with user-defined coordinate systems and implementing pixel-accurate collision detection. Initially, as you have experienced in building the first HTML5 application, 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 you can compare your code with the completed project's code during difficult debugging situations.

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.

How Do You Make a Great Video Game?

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

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

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

- *Technical design:* This includes all game code and the game platform and is generally not directly exposed to players; rather, it forms the foundation and scaffolding for all aspects of the game experience. This book is primarily focused on issues related to the technical design of games, including specific tasks such as the lines of code required to draw elements on the screen and more architectural considerations such as determining the strategy for how and when to load assets into memory. Technical design issues impact the player experience in many ways (for example, the number of times a player experiences “loading” delays during play or how many frames per second the game displays), but the technical design is typically invisible to players because it runs under what's referred to as the presentation layer, or all of the audiovisual and/or haptic feedback 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 system design, level design, and the interaction model (described later in this section).
- *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 (for example, 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. The unexpected global phenomenon Flappy Bird from GEARs Studio, 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 and balanced perfectly against each other, and underestimating systems complexity is perhaps one of the biggest pitfalls encountered by new (and veteran!) game designers. Until you know what you're getting into, always assume the systems you create will prove to be considerably more complex than you anticipate.

- *Level design:* A game's level design reflects the specific ways each of the other eight elements combine within the context of individual "chunks" of gameplay, where players must complete a certain chunk of objectives before continuing to the next section (some games may have only one level, while others will have dozens). Level designs within a single game can all be variations of the same core mechanic and systems design (games like Tetris and Bejeweled are examples of games with many levels all focusing on the same mechanic), while other games will mix and match mechanics and systems designs for variety among levels. Most games feature one primary mechanic and a game-spanning approach to systems design and will add minor variations between levels to keep things feeling fresh (changing environments, changing difficulty, adding time limits, increasing complexity, and the like), although occasionally games will introduce new levels that rely on completely separate mechanics and systems to surprise players and hold their interest. Great level design in games is a balance between creating "chunks" of play that showcase the mechanic and systems design and changing enough between these chunks to keep things interesting for players as they progress through the game (but not changing so much between chunks that the gameplay feels disjointed and disconnected).
- *Interaction model:* The interaction model is the combination of keys, buttons, controller sticks, touch gestures, and so on, used to interact with the game to accomplish tasks and the graphical user interfaces that support those interactions within the game world. Some game theorists break the game's user interface (UI) design into a separate category (game UI includes things such as menu designs, item inventories, heads-up displays [HUDs]), but the interaction model is deeply connected to UI design, and it's good 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 (for example, 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).
- *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. Games settings need not be elaborate to be effective; the game Candy Crush from King has a rather simple setting with a thin narrative wrapper, but the combination of setting, audiovisual design, and level design are uniquely well-matched and contribute significantly to the millions of hours players invest in the experience each month (as of 2015).

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

The magic of video games typically arises from the interplay between these nine elements, and the most successful games finely balance each as part of a unified vision to ensure a harmonious experience; this balance will always be unique to each individual effort and is found in games ranging from King's Candy Crush to Bioware's Mass Effect. The core game mechanic in many successful games is often a variation on one or more fairly simple, common themes (*Angry Birds*, for example, is a relatively basic projectile launching game), but 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.

References

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.

- Computer graphics:
 - Shirley, Ashikhmin, and Marschner. *Fundamentals of Computer Graphics*, 3rd edition. A. K. Peters, 2009.
 - Angle and Shreiner. *Interactive Computer Graphics: A Top Down Approach with WebGL*. Pearson Education, 2014.
- Linear algebra:
 - Johnson, Riess, and Arnold. *Introduction to Linear Algebra*, 5th edition. Addison-Wesley, 2002.
 - Anton and Rorres. *Elementary Linear Algebra: Applications Version*, 11th edition. Wiley, 2013.

Technologies

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

- *JavaScript*: <http://www.w3schools.com/js>
- *HTML5*: http://www.w3schools.com/html/html5_intro.asp
- *WebGL*: <https://www.khronos.org/webgl>
- *OpenGL*: <https://www.opengl.org>
- *NetBeans*: <https://netbeans.org>
- *Chrome*: <https://www.google.com/chrome>
- *glMatrix*: <http://glMatrix.net>
- *JSLint*: <http://www.jslint.com>

CHAPTER 2



Working with HTML5 and WebGL

After completing this chapter, you will be able to:

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

Introduction

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, such as mechanics, logic, and aesthetics.

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

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

Canvas for Drawing

To draw, you must first define and dedicate an area within the web page. We will begin with using the HTML canvas element to define an area for WebGL drawing.

The HTML5 Canvas Project

This project demonstrates how to draw and clear a canvas element on a web page. Figure 2-1 shows an example of running this project, which is defined in the Chapter2/2.1.HTML5Canvas folder.



The above is WebGL draw area!

Figure 2-1. Running the HTML5 Canvas project

The goals of the project are as follows:

- To learn how to set up the HTML canvas element
- To learn how to retrieve the canvas element from an HTML document for use in JavaScript
- To learn how to create a reference context to WebGL from the retrieved canvas element and manipulate the canvas from the WebGL context

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 HTML5 project titled HTML5 Canvas.
2. Open the index.html file in the editor by double-clicking the project name (HTML5Canvas) in the Project view, then double-clicking Site Root, and lastly double-clicking the index.html file, as illustrated in Figure 2-2.



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

3. Create the HTML canvas for drawing by adding the following line in the `index.html` file within the `body` element:

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

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

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

4. Create a `script` element for the inclusion of JavaScript programming code, once again within the `body` element.

```
<script type="text/javascript">
    // JavaScript code goes here.
</script>
```

This takes care of the HTML portion of this example. Now you will write JavaScript for the remainder of the example.

5. Retrieve a reference to the `GLCanvas` in your JavaScript by adding the following line within the `script` element:

```
var canvas = document.getElementById("GLCanvas");
```

The code creates a new variable named `canvas` and stores a reference to the `GLCanvas` drawing area in this variable.

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:

```
var gl = canvas.getContext("webgl") || canvas.getContext("experimental-webgl");
```

As the code indicates, the retrieved reference to the WebGL context is stored in the local variable named `gl`. From this variable, you have access to all WebGL functionality.

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); // set the color to be cleared
    gl.clear(gl.COLOR_BUFFER_BIT); // clear the colors
}
```

This code checks to ensure 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 the later chapters. For now, always assign 1.0 to the alpha channel.

You can refer to the final source code in the `index.html` file in the [Chapter2/2.1.HTML5Canvas](#) project. Run the project, and you should see a light green area on your browser window. This is the 640×480 canvas drawing area you defined.

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

Separating HTML and JavaScript

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

The JavaScript Source File Project

This project demonstrates how to logically separate the source code into appropriate files. This is accomplished by creating a separate JavaScript source code file named `WebGL.js` to implement the corresponding functionality in the `index.html` file. 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.JavaScriptSourceFile](#) folder.



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

Separate JavaScript Source Code File

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.

1. Create a new HTML5 project titled JavaScriptSourceFile.
2. Create a new folder named `src` inside the Site Root folder by right-clicking and creating a new folder, as illustrated in Figure 2-4.

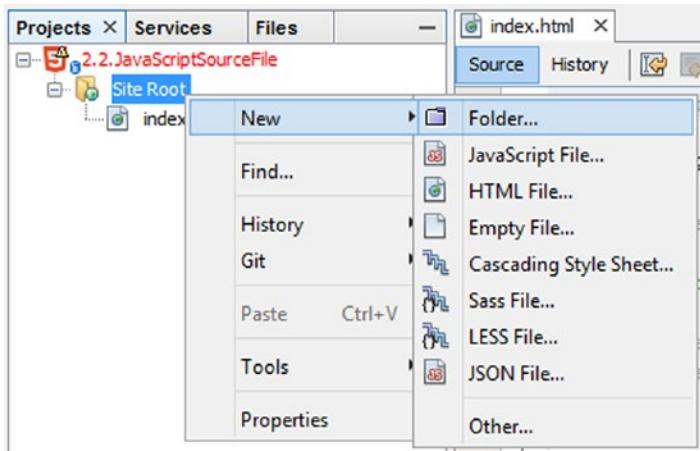


Figure 2-4. Creating a new source code folder

This folder will contain all of your source code.

Note In NetBeans you can create new folders, create new files, copy/paste projects, and rename projects by using the right-click menus in the Projects window.

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

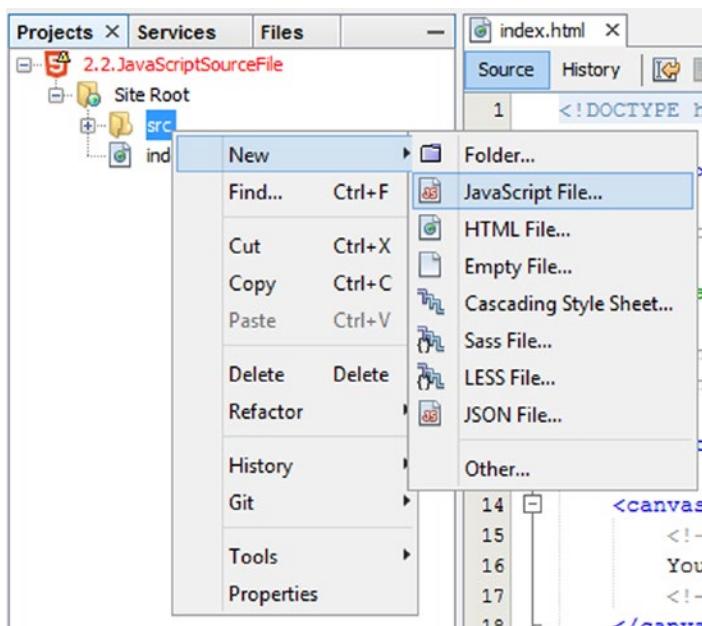


Figure 2-5. Adding a new JavaScript source code file

Name the new source file `WebGL.js`.

4. Open the new `WebGL.js` source file for editing.
5. Create a global variable referencing the WebGL context.

```
"use strict";
var gGL = null;
```

Note All global variable names begin with a lowercase `g`, as in `gGL`.

6. Define the `initializeGL()` function to retrieve `GLCanvas`, bind the drawing area with the WebGL context, and store the results in the global `gGL` variable.

```
function initializeGL() {
    var canvas = document.getElementById("GLCanvas");

    gGL = canvas.getContext("webgl") ||
        canvas.getContext("experimental-webgl");
```

```

if (gGL !== null) {
    gGL.clearColor(0.0, 0.8, 0.0, 1.0); // set the color to be cleared
} else {
    document.write("<br><b>WebGL is not supported!</b>");
}
}

```

Note All public function names begin with a lowercase letter, as in `initializeGL()`.

Notice this function is similar to the JavaScript source code you typed in the previous project.

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

```

function clearCanvas() {
    gGL.clear(gGL.COLOR_BUFFER_BIT); // clear to the color previously set
}

```

8. Define the `doGLDraw()` function to carry out the initialization and clearing of the canvas area.

```

function doGLDraw() {
    initializeGL();
    clearCanvas();
}

```

Load and Run JavaScript Source Code from index.html

With all the JavaScript functionality defined in the `WebGL.js` file, you now need to load this file and invoke the `doGLDraw()` function from your web page, the `index.html` file.

1. Open the `index.html` file for editing.
2. Create the HTML canvas, `GLCanvas`, as in the previous project.
3. Load the `WebGL.js` source code by including the following code within the `body` element:

```
<script type="text/javascript" src="src/WebGL.js"></script>
```

You can include this line either before or after the definition of `canvas`, as long as it is within the `body` element.

4. Execute the `doGLDraw()` function after `WebGL.js` is loaded.

```
<body onload="doGLDraw();">
```

The modification to the `body` tag says once all loading operations are done, the `doGLDraw()` function should be executed.

You can refer to the final source code in the `WebGL.js` and `index.html` files in the Chapter2/2.2.JavaScriptSourceFile project. Although the output from this project is identical to that from the previous project, the organization of your code will allow you to expand, debug, and understand the game engine as you continue to add new functionality.

Observations

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 moved to `WebGL.js`. This clean separation of information allows for easy understanding of the source code and improves support for complex systems. From this point on, all JavaScript source code will be added to separate source code files. In all cases, in the same manner as you have included the loading of `WebGL.js`, you will have to remember to load the new source code, such as `NewSourceFile.js`, by including the following line within the body element of the `index.html` file:

```
<script type="text/javascript" src="src/NewSourceFile.js"></script>
```

Elementary Drawing with WebGL

Drawing with WebGL is a multiple-step process that involves transferring geometric data and OpenGL Shading Language (GLSL) instructions (the shaders) from memory to the drawing hardware, or the graphical 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 and build the game engine.

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

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.DrawOneSquare folder.

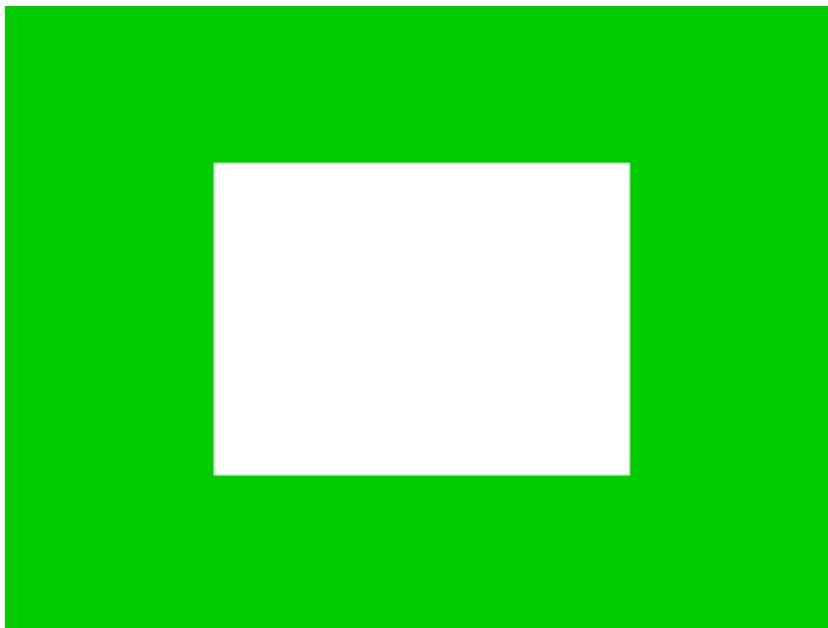


Figure 2-6. Running the Draw One Square project

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 to draw with WebGL

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 global variable. Learning from the previous project, the corresponding JavaScript code will be stored in a new source code file, `VertexBuffer.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 `VertexBuffer.js`.

2. Declare a global variable `gSquareVertexBuffer` to store the reference to the WebGL buffer location.

```
"use strict";
var gSquareVertexBuffer = null;
```

3. Define the `initSquareBuffer()` function to create and load vertices onto the GPU.

```
function initSquareBuffer() {

    // First: define the vertices for a square
    var verticesOfSquare = [
        0.5, 0.5, 0.0,
        -0.5, 0.5, 0.0,
        0.5, -0.5, 0.0,
        -0.5, -0.5, 0.0
    ];

    // Step A: Create a buffer on the gGL context for our vertex positions
    gSquareVertexBuffer = gGL.createBuffer();

    // Step B: Activate vertexBuffer
    gGL.bindBuffer(gGL.ARRAY_BUFFER, gSquareVertexBuffer);

    // Step C: Loads verticesOfSquare into the vertexBuffer
    gGL.bufferData(gGL.ARRAY_BUFFER, new Float32Array(verticesOfSquare),
    gGL.STATIC_DRAW);
}
```

In the code shown, the vertices of a unit square are defined first. Notice the z-dimension is set to 0.0 because you are building a 2D game engine. Step A creates a buffer on the GPU for storing the vertex positions of the square and stores the reference to the GPU buffer in the global variable `gSquareVertexBuffer`. Step B activates the newly created buffer, and step C loads the vertex position of the square into the activated buffer. The keyword `STATIC_DRAW` informs the drawing hardware that this buffer will not be changed.

Tip Remember that the `gGL` global variable is defined in the `WebGL.js` file and initialized by the `initializedGL()` function.

4. Lastly, remember to load the `VertexBuffer.js` source code in your web page by adding the following code within the body element of the `index.html` file:

```
<script type="text/javascript" src="src/VertexBuffer.js"></script>
```

Tip From now on, when you are reminded to “load the new source file in `index.html`,” you should add the `script` line (remember to change `VertexBuffer.js` to your new source code file name) into `index.html`.

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 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 memory the source code for both vertex and fragment shaders, compile and link them into a single shader program, and load the compiled program into the GPU. 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 `ShaderSupport.js` source file.

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

Define the Vertex and Fragment Shaders

GLSL shaders are simply programs consisting of GLSL instructions.

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

```
<script type="x-shader/x-vertex" id="VertexShader">
    attribute vec3 aSquareVertexPosition;
    void main(void) {
        gl_Position = vec4(aSquareVertexPosition, 1.0);
    }
</script>
```

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

The `GLSL` attribute keyword identifies per-vertex data that will be passed to the vertex shader in the GPU. In this case, the `aSquareVertexPosition` attribute is of data type `vec3` or an array of three floating-point numbers. As you will see in later steps, `aSquareVertexPosition` will contain vertex positions for a 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 `aSquareVertexPosition` into a `vec4` and passing the information to WebGL.

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

```
<script type="x-shader/x-fragment" id="FragmentShader">
    void main(void) {
        gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
    }
</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, `ShaderSupport.js`.

- Create a new JavaScript file, `ShaderSupport.js`.
- Create two global variables, `gSimpleShader` and `gShaderVertexPositionAttribute`, to store the reference to the shader program and the vertex position attribute in the GPU.

```
var gSimpleShader = null;
var gShaderVertexPositionAttribute = null;
```

- Create a function to load and compile a shader from `index.html`.

```
function loadAndCompileShader(id, shaderType) {
    var shaderText, shaderSource, compiledShader;

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

    // Step B: Create the shader based on the source type: vertex or fragment
    compiledShader = gGL.createShader(shaderType);

    // Step C: Compile the created shader
    gGL.shaderSource(compiledShader, shaderSource);
    gGL.compileShader(compiledShader);

    // Step D: check for error and return result
    if (!gGL.getShaderParameter(compiledShader, gGL.COMPILE_STATUS)) {
        alert("A shader compiling error occurred: " +
            gGL.getShaderInfoLog(compiledShader));
    }
    return compiledShader;
}
```

Step A of the code finds shader source code from the `index.html` file using the `id` field you specified when defining the shaders, either `VertexShader` or `FragmentShader`. Step B creates a specified shader (either vertex or fragment) in the GPU. Step C specifies the shader source code and compiles the shader. Finally, step D checks and returns the reference to the compiled shader where an error will result in a null value.

4. You are now ready to create and compile a shader program by defining the `initSimpleShader` function.

```
function initSimpleShader(vertexShaderID, fragmentShaderID) {
    // Step A: load and compile the vertex and fragment shaders
    var vertexShader = loadAndCompileShader(vertexShaderID, gGL.VERTEX_SHADER);
    var fragmentShader = loadAndCompileShader(fragmentShaderID,
        gGL.FRAGMENT_SHADER);

    // Step B: Create and link the shaders into a program.
    gSimpleShader = gGL.createProgram();
    gGL.attachShader(gSimpleShader, vertexShader);
    gGL.attachShader(gSimpleShader, fragmentShader);
    gGL.linkProgram(gSimpleShader);

    // Step C: check for error
    if (!gGL.getProgramParameter(gSimpleShader, gGL.LINK_STATUS))
        alert("Error linking shader");

    // Step D: Gets a reference to the aSquareVertexPosition attribute
    gShaderVertexPositionAttribute = gGL.getAttributeLocation(gSimpleShader,
        "aSquareVertexPosition");

    // Step E: Activates the vertex buffer loaded in VertexBuffer.js
    gGL.bindBuffer(gGL.ARRAY_BUFFER, gSquareVertexBuffer);
    // Step F: Describe the characteristic of the vertex position attribute
    gGL.vertexAttribPointer(gShaderVertexPositionAttribute,
        3,           // each vertex element is a 3-float (x,y,z)
        gGL.FLOAT,   // data type is FLOAT
        false,       // if the content is normalized vectors
        0,           // number of bytes to skip in between elements
        0);          // offsets to the first element
}
```

Step A of the code compiles the shader code you defined in `index.html` by calling the `loadAndCompileShader()` function with the corresponding parameters. Step B loads the compiled shader onto the GPU and links the two shaders into a program. The reference to this program is stored in the global variable `gSimpleShader`. After error checking in step C, step D locates and stores the reference to the `aSquareVertexPosition` attribute defined in your vertex shader. Step E activates the vertex buffer you loaded in `VertexBuffer.js`, and step F connects the activated buffer to the `aSquareVertexPosition` attribute by describing the data format of the vertex buffer, where each vertex position is a three-float (x, y, z) position.

5. Finally, as with any new source code file, remember to load the `ShaderSupport.js` file the `index.html` file.

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

Set Up Drawing with WebGL

With the vertex data and shaders 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 stored in the `WebGL.js` file. Now open this file for editing.

1. Modify the `initializeGL()` function to include the initialization of the vertex buffer and the shader program.

```
function initializeGL() {
    var canvas = document.getElementById("GLCanvas");
    gGL = canvas.getContext("webgl") || canvas.getContext("experimental-webgl");

    if (gGL !== null) {
        gGL.clearColor(0.0, 0.8, 0.0, 1.0); // set the color to be cleared

        // A. initialize the vertex buffer
        initSquareBuffer(); // This function is defined VertexBuffer.js

        // B. now load and compile the vertex and fragment shaders
        initSimpleShader("VertexShader", "FragmentShader");
            // the two shaders are defined in the index.html file
            // initSimpleShader() function is defined in ShaderSupport.js
    } else {
        document.write("<br><b>WebGL is not supported!</b>");
    }
}
```

The code in bold shows you should modify the `initializeGL()` function to call the `initSquareBuffer()` and `initSimpleShader()` functions after successfully obtaining the WebGL context.

2. Replace the `clearCanvas()` function with the `drawSquare()` function for drawing the defined square.

```
function drawSquare() {
    gGL.clear(gGL.COLOR_BUFFER_BIT);

    // Step A: Activate the shader to use
    gGL.useProgram(gSimpleShader);

    // Step B: Enable the vertex position attribute
    gGL.enableVertexAttribArray(gShaderVertexPositionAttribute);

    // Step C: Draw with the above settings
    gGL.drawArrays(gGL.TRIANGLE_STRIP, 0, 4);
}
```

This code shows the steps to draw with WebGL. Step A activates the shader program to use. Step B enables the vertex attribute for the vertex shader. Finally, step C issues the draw command. In this case, you are issuing a command to draw the four vertices as two connected triangles that form a square.

3. Lastly, modify doGLDraw() to call the drawSquare() function.

```
function doGLDraw() {
    initializeGL();           // Binds gGL context to WebGL functionality
    drawSquare();             // Clears the GL area and draws one square
}
```

Recall that doGLDraw() is the function called by `index.html` after all source code files are completely loaded. For this reason, WebGL will be initialized and the white square drawn. For reference, you can refer to the source code in the Chapter[2/2.3.DrawOneSquare](#) project.

Observations

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 this chapter.

You can try editing the fragment shader in `index.html` by changing the color set in the `gl_FragColor` function to change the color of the white square. Notice that a value of less than 1 in the alpha channel will result in the white square becoming transparent and showing through some of the greenish canvas color.

Finally, note that this project defines many global variables with little attempt at hiding information. This organization does not lend itself to supporting changes in functionality or growth in complexity. In the next sections, you will encapsulate and abstract portions of this example to form the basis of the game engine framework.

Abstraction with JavaScript Objects

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

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

The JavaScript Objects Project

This project demonstrates how to abstract the global functions and variables from the Draw One Square project into JavaScript objects. This object-oriented abstraction will result in a framework that offers manageability and expandability for subsequent projects. As illustrated in Figure [2-7](#), when running, this project displays a white rectangle in a greenish canvas, identical to that from the Draw One Square project. The source code to this project is defined in the Chapter[2/2.4.JavaScriptObjects](#) folder.

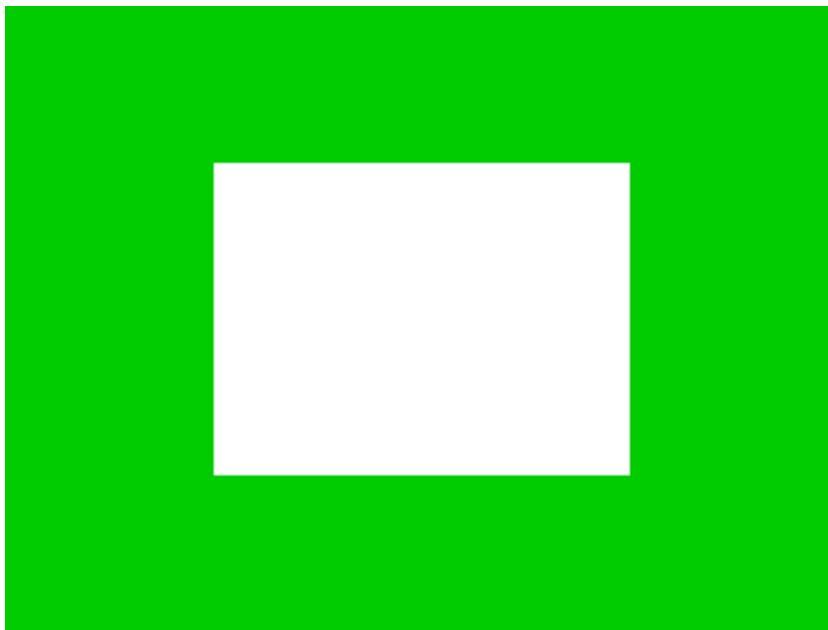


Figure 2-7. Running the JavaScript Objects project

The goals of the project are as follows:

- To separate the game engine from the game logic code
- To demonstrate the implementation of a Singleton-like object based on the JavaScript Module pattern
- To understand how to build abstractions with JavaScript objects

The steps for creating the project are as follows:

1. Create separate folders to organize the source code for the game engine and the logic of the game.
2. Define JavaScript objects to abstract the game engine functionality: `Core`, `VertexBuffer`, and `SimpleShader`. These objects will be defined in corresponding JavaScript source code files.
3. Define a JavaScript object to implement the drawing of one square, which is the logic of your simple game for now.

Source Code Organization

Create a new HTML5 project with the NetBeans IDE with a source code folder named `src`. Within `src`, create `Engine` and `MyGame` as subfolders, as illustrated in Figure 2-8.

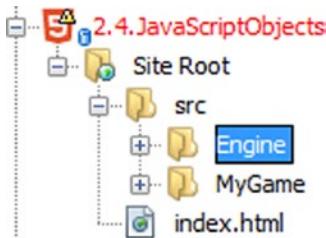


Figure 2-8. Creating Engine under the src folder

The src/Engine folder will contain all the source code to the game engine, and the src/MyGame 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.

Tip The source code in the MyGame folder implements the game by relying on the functionality provided by the game engine defined in the Engine folder. For this reason, in this book, the source code in the MyGame folder is often referred to as the *client* of the game engine.

Abstracting the Game Engine

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

These subsystems will be covered in later chapters of this book. This section focuses on establishing the mechanism and organization for implementing these single-instance or Singleton-like objects based on the JavaScript Module pattern.

The Core of the Game Engine: gEngine.Core

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

1. Create a new source file in the src/Engine folder and name the file `Engine_Core.js`. Remember to load this new source file in `index.html`.
2. At the beginning of the file, create `gEngine`, a new global variable representing the core of the game engine as follows:

```

"use strict"; // Operate in Strict mode

var gEngine = gEngine || { };
// initialize the variable while ensuring it is not redefined

```

This line instructs the system to retain a current `gEngine` variable if it is defined; otherwise, it creates a new instance with an initial empty object. Different subsystems of the game engine will be implemented as distinct properties of the `gEngine` object in separate files. This line ensures that properties defined in separate source files will be retained independent from the order upon which they are loaded.

3. Define the `gEngine.Core` property as follows:

```
gEngine.Core = (function() {
    // instance variable: the graphical context for drawing
    var mGL = null;

    // Accessor of the webgl context
    var getGL = function() { return mGL; };

    // Contains the functions and variables that will be accessible.
    var mPublic = {
        getGL: getGL
    };

    return mPublic;
}());
```

Note All instance variable names begin with an *m* and are followed by a capital letter, as in `mVariable`. Though not enforced by JavaScript, you should never access an instance variable from outside the object. For example, you should never access `gEngine.Core.mGL` directly; instead, call the `gEngine.Core.getGL()` function to access the variable.

The code defines the `gEngine.Core` property as a global object. This object contains one private instance variable and a public accessor. It is important to note the following about the syntax:

- a. The `mPublic` object defines public methods for `gEngine.Core`. In this case, `getGL()` is the only public method of the `gEngine.Core` object.
 - b. The `()` at the end of the function block signals to the system to execute the function immediately, and thus only a single instance of `gEngine.Core` will be created.
 - c. This is the JavaScript Module pattern, which will be used to implement all the subsystems in the game engine.
4. Inside the `gEngine.Core` object, add a function to initialize the WebGL context from the HTML canvas ID.

```
// initialize the WebGL, the vertex buffer and compile the shaders
var initializeWebGL = function(htmlCanvasID) {
    var canvas = document.getElementById(htmlCanvasID);
```

```

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

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

// now initialize the VertexBuffer
gEngine.VertexBuffer.initialize();
};

```

Apart from the syntax involved with the JavaScript Module pattern, the previous function should appear very similar to the `initializeGL()` function from the previous project and actually accomplishes the same task as it. A difference to note is the call to `gEngine.VertexBuffer.initialize()`. This function call sets up the WebGL vertex buffer and will be described in detail.

5. Add an additional function to clear the canvas to the desired color.

```

// Clears the draw area and draws one square
var clearCanvas = function(color) {
    mGL.clearColor(color[0], color[1], color[2], color[3]); // set the color to
    be cleared
    mGL.clear(mGL.COLOR_BUFFER_BIT); // clear to the color previously set
};

```

6. Finally, export the public functions through the returned `mPublic` object.

```

var mPublic = {
    getGL: getGL,
    initializeWebGL: initializeWebGL,
    clearCanvas: clearCanvas
};

```

The Shared Vertex Buffer

In the game engine that you are building, all graphical objects will be drawn based on the unit square. For this reason, the geometry subsystem is rather simplistic. The `gEngine.VertexBuffer` object implements the geometry subsystem.

1. Create a new source file in the `src/Engine` folder and name the file `Engine_VertexBuffer.js`. Remember to load this new source file in your `index.html` file.
2. Follow the JavaScript Module pattern and define `VertexBuffer` as a property of `gEngine`, as follows:

```

"use strict"; // Operate in Strict mode

var gEngine = gEngine || { };

// The VertexBuffer object
gEngine.VertexBuffer = (function() {

```

```

// First: define the vertices for a square
var verticesOfSquare = [
    0.5, 0.5, 0.0,
    -0.5, 0.5, 0.0,
    0.5, -0.5, 0.0,
    -0.5, -0.5, 0.0
];

// reference to the vertex positions for the square in the gl context
var mSquareVertexBuffer = null;

var getGLVertexRef = function() { return mSquareVertexBuffer; };

var initialize = function() {
    var gl = gEngine.Core.getGL();

    // Step A: Create a buffer on the gGL context for our vertex positions
    mSquareVertexBuffer = gl.createBuffer();

    // Step B: Activate vertexBuffer
    gl.bindBuffer(gl.ARRAY_BUFFER, mSquareVertexBuffer);

    // Step C: Loads verticesOfSquare into the vertexBuffer
    gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(verticesOfSquare),
        gl.STATIC_DRAW);
};

var mPublic = {
    initialize: initialize,
    getGLVertexRef: getGLVertexRef
};

return mPublic;
}();

```

The `gEngine.VertexBuffer` object implements the same functionality as the source code in `VertexBuffer.js` from the previous project: it creates and loads the unit square geometry to the WebGL vertex buffer. Notice that instead of being global, `mSquareVertexBuffer`, the reference to the WebGL vertex buffer, is now a private variable with a proper accessor. The `initialize()` function contains the same three setup steps as the `initSquareBuffer()` function from the previous project.

Recall that the `gEngine.VertexBuffer.initialize()` function is called from the end of the `gEngine.Core.initializeWebGL()` function. In this way, after the `InitialzeWebGL()` function, the WebGL will be initialized and loaded with the vertex positions of the unit square, and the vertex buffer reference will be ready for drawing operations.

The Shader Object

Although the code in the `ShaderSupport.js` file from the previous project properly implements the required functionality, the global variables and functions do not lend themselves well to modification and code reuse. This section follows the object-oriented design principles and creates a `SimpleShader` object to abstract the behaviors and hide internal representations of shaders.

1. Create a new source file in the `src/Engine` folder and name the file `SimpleShader.js` to implement the `SimpleShader` object. Remember to load this new source file in `index.html`.
2. Define the constructor for `SimpleShader` to load, compile, and link the shaders into a program and to create a reference for loading from the WebGL vertex buffer for drawing.

```

function SimpleShader(vertexShaderID, fragmentShaderID) {
    // instance variables (Convention: all instance variables: mVariables)
    this.mCompiledShader = null;
        // reference to the compiled shader in webgl context
    this.mShaderVertexPositionAttribute = null;
        // reference to SquareVertexPosition in shader

    var gl = gEngine.Core.getGL();

    // start of constructor code
    //
    // Step A: load and compile vertex and fragment shaders
    var vertexShader = this._loadAndCompileShader(vertexShaderID, gl.VERTEX_SHADER);
    var fragmentShader = this._loadAndCompileShader(fragmentShaderID,
        gl.FRAGMENT_SHADER);

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

    // Step C: check for error
    if (!gl.getProgramParameter(this.mCompiledShader, gl.LINK_STATUS)) {
        alert("Error linking shader");
        return null;
    }

    // Step D: Gets a reference to the aSquareVertexPosition attribute
    this.mShaderVertexPositionAttribute = gl.getAttribLocation(this.mCompiledShader,
        "aSquareVertexPosition");

    // Step E: Activates the vertex buffer loaded in Engine.Core_VertexBuffer
    gl.bindBuffer(gl.ARRAY_BUFFER, gEngine.VertexBuffer.getGLVertexRef());

    /// Step F: Describe the characteristic of the vertex position attribute
    gl.vertexAttribPointer(this.mShaderVertexPositionAttribute,
        3,           // each element is a 3-float (x,y,z)
        gl.FLOAT,    // data type is FLOAT
        false,       // if the content is normalized vectors
        0,           // number of bytes to skip in between elements
        0);          // offsets to the first element
}

```

Notice that this constructor is similar to the `initSimpleShader()` function from the previous project, with the following exceptions:

- a. The global `gShaderVertexPositionAttribute` and `gSimpleShader` variables are now the private instance variables `mShaderVertexPositionAttribute` and `mCompiledShader`.
- b. The reference to WebGL context, the `gl` variable, is now accessed through `gEngine.Core`.
3. Add a `_loadAndCompileShader()` method to the `SimpleShader` prototype to perform the actual loading and compiling functionality.

```
// Returns a compiled shader from a shader in the dom.
// The id is the id of the script in the html tag.
SimpleShader.prototype._loadAndCompileShader = function(id, shaderType) {
    var shaderText, shaderSource, compiledShader;
    var gl = gEngine.Core.getGL();

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

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

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

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

    return compiledShader;
};
```

Notice that this method is almost identical to the `LoadAndCompile()` function from the previous project. The only modification is that the WebGL context is now accessed through `gEngine.Core`.

4. Add a function to activate the shader for drawing.

```
SimpleShader.prototype.activateShader = function() {
    var gl = gEngine.Core.getGL();
    gl.useProgram(this.mCompiledShader);
    gl.enableVertexAttribArray(this.mShaderVertexPositionAttribute);
};
```

Note Functions that are meant to be private will have names that begin with an underscore (_), as in `_loadAndCompileShader()`. Though not enforced by JavaScript, you should never call a function with a name that begins with an underscore from outside the object. For example, the `_loadAndCompileShader()` function is not designed to be called from outside the `SimpleShader` object.

5. Finally, add an accessor for the actual WebGL shader program.

```
SimpleShader.prototype.getShader = function() { return this.mCompiledShader; };
```

The `src/Engine` folder contains the source code to the game engine. In this way, the game engine is simply a library that provides functionality for creating games. For now, your game engine consists of three objects that support the initialization of WebGL and the drawing of a unit square. This is the folder that you will continue to add source files and functionality to, which eventually will become a complete and sophisticated game engine.

The Client Source Code

The `src/MyGame` folder will contain the actual source code to a game. As mentioned, the code in this folder will be referred to as the *client* of the game engine. For now, the source code in the `MyGame` folder will focus on testing the functionality of the simple game engine.

1. Create a new source file in the `src/MyGame` folder, or the client folder, and name the file `MyGame.js`. Remember to load this new source file in `index.html`.
2. Create a constructor that receives an HTML canvas ID as a parameter and contains a shader as an instance variable.

```
function MyGame(htmlCanvasID) {
    // The shader for drawing
    this.mShader = null;
    ...
}
```

3. Within the constructor, initialize the WebGL context and the `VertexBuffer`.

```
gEngine.Core.initializeWebGL(htmlCanvasID);
```

4. Then, create, load, and compile the shaders by doing the following:

```
this.mShader = new SimpleShader("VertexShader", "FragmentShader");
```

5. Finally, clear the canvas, activate the shader, and draw.

```
// Step C1: Clear the canvas
gEngine.Core.clearCanvas([0, 0.8, 0, 1]);

// Step C2: Activate the proper shader
this.mShader.activateShader();
```

```
// Step C3: Draw with the currently activated geometry and the activated shader
var gl = gEngine.Core.getGL();
gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);
```

Remember to create the `MyGame` object from within the `index.html` `body` element.

```
<body onload="new MyGame('GLCanvas');">
```

Observations

Although you're accomplishing the same tasks as with the previous project, with this project you have created infrastructure that supports subsequent modifications and expansions of your game engine. You have organized your source code into separate and logical folders, created Singleton-like objects to implement core functionality of the engine, and gained experience with abstracting the `SimpleShader` object that will support future design and code reuse. With the engine now comprised of well-defined objects with clean interface methods, you can learn new concepts and build new abstractions, which you can continually add to your engine.

Separating GLSL from HTML

Recall that in your projects thus far 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 from HTML source files; logically, continuously adding to `index.html` will result in an unmanageable, cluttered file that would become difficult to work with. For these reasons, the GLSL shaders should be stored in separate source files.

The Shader Source Files Project

This project demonstrates how to separate the GLSL shaders into separate files. As illustrated in Figure 2-9, when running this project, a white rectangle is displayed on a greenish canvas, identical to the previous projects. The source code to this project is defined in the [Chapter2/2.5.ShaderSourceFiles](#) folder.

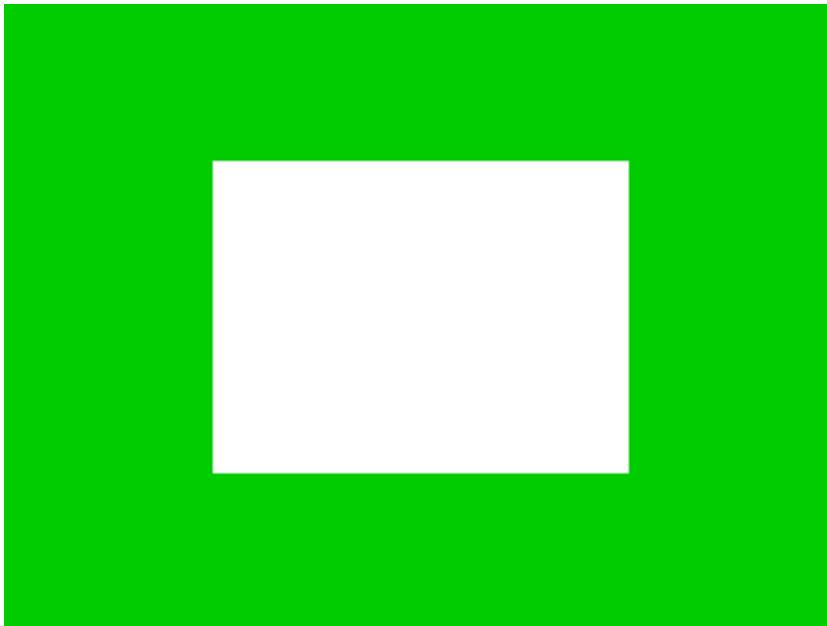


Figure 2-9. Running the Shader Source Files project

The goals of the project are as follows:

- To separate the GLSL shaders from the HTML source code
- To demonstrate how to load the shader source files during 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 previous project and open the `SimpleShader.js` file to edit the `_loadAndCompileShader()` function to receive a file path instead of an HTML ID:

```
SimpleShader.prototype._loadAndCompileShader = function(filePath, shaderType)
```

2. Replace the HTML element retrieval code with the following `XMLHttpRequest` to load a file:

```
xmlReq = new XMLHttpRequest();
xmlReq.open('GET', filePath, false);
try {
    xmlReq.send();
} catch (error) {
    alert("Failed to load shader: " + filePath);
    return null;
}
shaderSource = xmlReq.responseText;
```

```

if (shaderSource === null) {
    alert("WARNING: Loading of:" + filePath + " Failed!");
    return null;
}

```

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

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

Note The `XMLHttpRequest()` object requires a running web server to fulfill the HTTP get request. This means you will be able to test this project from within the NetBeans IDE. However, unless there is a web server running on your machine, you will not be able to run this project by double-clicking the `index.html` file directly. This is because there is no server to fulfill the HTTP get requests and the GLSL shader loading will fail.

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

Extracting Shaders into Their Own Files

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

1. Create a new folder in the `src` folder and name it `GLSLShaders`, as illustrated in Figure 2-10.

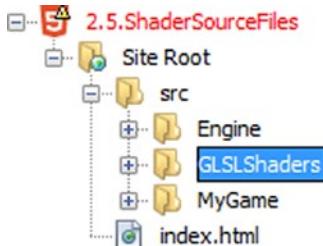


Figure 2-10. Creating the `GLSLShaders` folder

This will be the folder that contains all of the GLSL shader source code files.

2. Create two new text files within the `GLSLShaders` folder and name them `SimpleVS.gls1` and `WhiteFS.gls1`.

Note All GLSL shader source code files will end with the `.gls1` extension. The `vs` in the shader file names signifies that the file contains a vertex shader, while `FS` signifies a fragment shader.

3. To create the GLSL vertex shader, edit `SimpleVS.gls1` to add the existing vertex shader code in the `index.html` file from the previous project.

```
attribute vec3 aSquareVertexPosition; // Expects one vertex position
void main(void) {
    gl_Position = vec4(aSquareVertexPosition, 1.0);
}
```

4. To create the GLSL fragment shader, edit `WhiteFS.gls1` to add the fragment shader code in the `index.html` file from the previous project.

```
void main(void) {
    gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
}
```

Cleaning Up HTML Code

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.

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

```
<!DOCTYPE html>
<head>
    <title>2.5: The Shader Source Files Project</title>
</head>
<body onload="new MyGame('GLCanvas');">
    <!-- Engine code -->
    <script type="text/javascript" src="src/Engine/Engine_Core.js"></script>
    <script type="text/javascript" src="src/Engine/Engine_VertexBuffer.js">
    </script>
    <script type="text/javascript" src="src/Engine/SimpleShader.js"></script>
    <!-- Client game code -->
    <script type="text/javascript" src="src/MyGame/MyGame.js"></script>
    ...
</body>
```

Notice that `index.html` no longer contains any GLSL shader and only a single line of JavaScript code (to create the `MyGame` object). With this organization, the `index.html` file can properly be considered as representing the web page where you will not need to edit it to modify a shader.

2. Modify the game to load the shader files instead of HTML element IDs. Edit `MyGame.js` to modify step B to refer to the shader files created.

```
this.mShader = new SimpleShader(
    "src/GLSLShaders/SimpleVS.gls1",    // Path to the VertexShader
    "src/GLSLShaders/WhiteFS.gls1");    // Path to the FragmentShader
```

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 all of the source code for your game.
- `src/GLSLShaders`: This is the folder that contains all the GLSL source code files that shade the elements of your game.
- `src/Engine`: This is the folder that contains all the source code for your game engine.
- `src/MyGame`: 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, `WhiteFS.glsl`, with a shader that can be parameterized to draw with any color.

The Parameterized Fragment Shader Project

This project replaces `WhiteFS.glsl` with a `SimpleFS.glsl` that supports the drawing with any color. Figure 2-11 shows the output of running the Parameterized Fragment Shader project; notice that a blue square replaces the white square from previous projects. The source code for this project is defined in the Chapter2/2.6.ParameterizedFragmentShader folder.

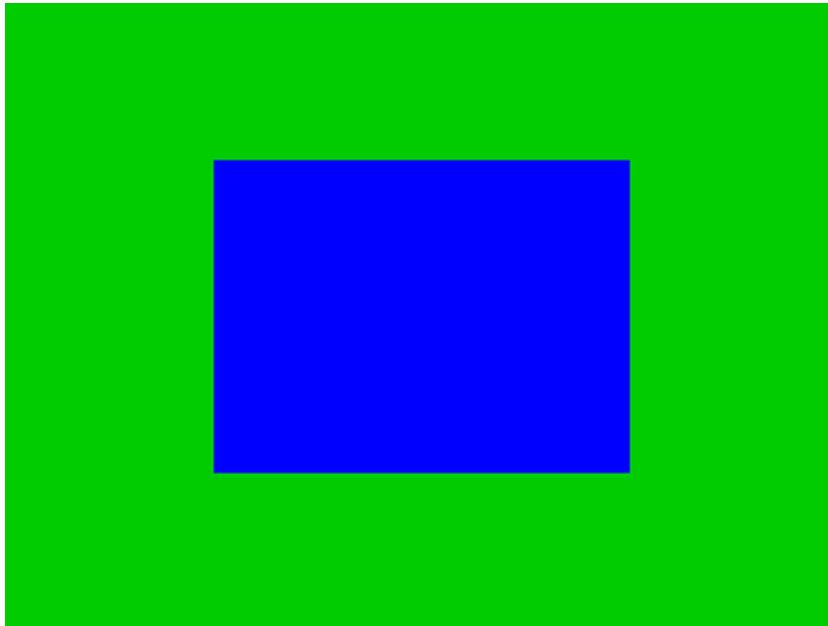


Figure 2-11. Running the Parameterized Fragment Shader project

The goals of the project are as follows:

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

Defining the SimpleFS.glsL Fragment Shader

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/GLSLShaders` folder and name it `SimpleFS.glsL`. Edit this file to add the following:

```
precision mediump float; // sets the precision for floating point computation
uniform vec4 uPixelColor; // to transform the vertex position
void main(void) {
    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.

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

Modify the SimpleShader to Support the Color Parameter

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

1. Edit `SimpleShader.js` and add a new instance variable for referencing the `uPixelColor`.

```
this.mPixelColor = null;
// reference to the pixelColor uniform in the fragment shader
```

2. Add code to the end of the constructor to create the reference.

```
// Step G: Gets a reference to the uniform variable uPixelColor in the
//          fragment shader
this.mPixelColor = gl.getUniformLocation(this.mCompiledShader, "uPixelColor");
```

3. Modify the shader activation to allow the setting of the pixel color.

```
SimpleShader.prototype.activateShader = function (pixelColor) {
    var gl = gEngine.Core.getGL();
    gl.useProgram(this.mCompiledShader);
    gl.enableVertexAttribArray(this.mShaderVertexPositionAttribute);
    gl.uniform4fv(this.mPixelColor, pixelColor);
};
```

The `gl.uniform4fv()` function copies four floating-point values from `pixelColor` to the `mPixelColor`, or the `uPixelColor`, in the `SimpleFS.gls1` fragment shader.

Drawing with the New Shader

To test `SimpleFV.gls1`, modify the `MyGame` constructor to create and draw with the new shader.

```
function MyGame(htmlCanvasID) {
    // Step A: Initialize the webGL Context and the VertexBuffer
    gEngine.Core.initializeWebGL(htmlCanvasID);

    // Step B: Create, load and compile the shaders
    this.mShader = new SimpleShader(
        "src/GLSLShaders/SimpleVS.gls1",           // Path to the VertexShader
        "src/GLSLShaders/SimpleFS.gls1";          // Path to the FragmentShader

    // Step C: Draw!
    // Step C1: Clear the canvas
    gEngine.Core.clearCanvas([0, 0.8, 0, 1]);

    // Step C2: Activate the proper shader
    this.mShader.activateShader([0, 0, 1, 1]);

    // Step C3: Draw with the currently activated geometry and the activated shader
    var gl = gEngine.Core.getGL();
    gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);
}
```

Notice that the `mConstColorShader` is created with the new `SimpleFS.gls1` (instead of `WhiteFS`) and that it is now important to set the drawing color when activating the shader. With the new `SimpleFS`, you can now experiment with drawing the squares with any desired color.

As you have experienced in this project, the source code structure supports simple and localized changes when the game engine is expanded, in this case only changes to `SimpleShader.js` file. This demonstrates the benefit of proper encapsulation and source code organization.

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 needed in order to build an excellent foundation for the game engine. You have also structured the source code in a way that allows you 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.

CHAPTER 3



Drawing Objects in the World

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

Introduction

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.

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, drawing support for 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 upon which they play. Additionally, to support proper presentation, the game engine must allow drawing to specific subregions on the canvas such 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, builds JavaScript objects 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 functionality of a game engine, the details of how drawings are accomplished can actually be distractions to the gameplay programming. For example, it is important to create, control the locations of, and draw soccer players in a soccer game. However, leaving the details of how each player is actually defined (by a collection of vertices that form triangles) exposed can quickly overwhelm and complicate the process of the game's development. 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

This project introduces the `Renderable` object to encapsulate the drawing operation. Over the next few projects you will learn more supporting concepts to refine the implementation of the `Renderable` object such that multiple instances of this object 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.RenderableObjects](#) folder.

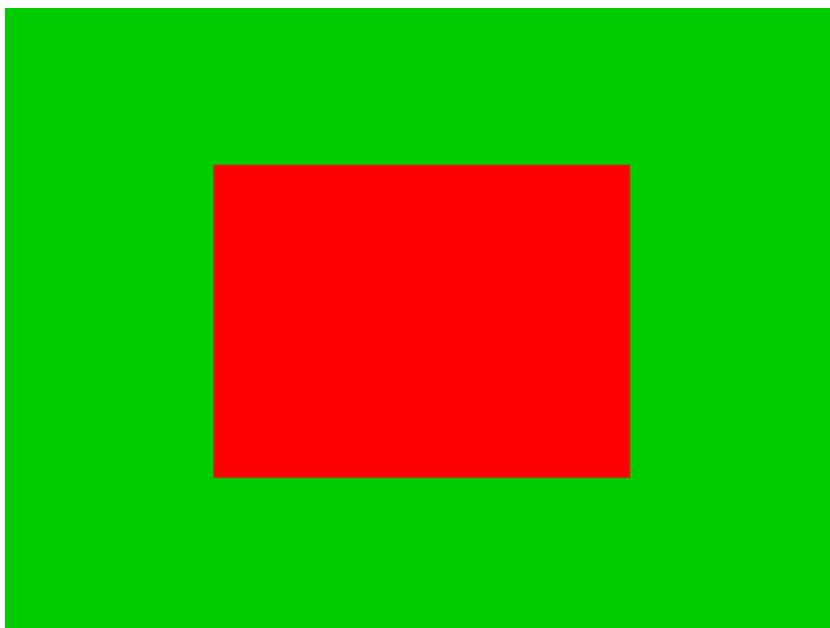


Figure 3-1. Running the `Renderable Objects` project

The goals of the project are as follows:

- To begin the process of building an object to encapsulate the drawing operations by first abstracting the drawing functionality
- To demonstrate how to create different instances of `SimpleShader`
- To demonstrate the ability to create multiple `Renderable` objects

The Renderable Object

This project introduces the `Renderable` object to encapsulate the drawing process and reorganizes the source code folders to maintain the structure as the number of source code files increases. You will continue to see this pattern of learning new concepts followed by defining abstractions to hide the details of the concepts to support programmability and extensibility.

1. Define the `Renderable` object in the game engine by creating a new source code file in the `src/Engine` folder and name the file `Renderable.js`. Remember to load this new source file in `index.html`.
2. Open `Renderable.js` and create a constructor that receives a `SimpleShader` as a parameter with a `color` instance variable.

```
function Renderable(shader) {
    this.mShader = shader; // the shader for shading this object
    this.mColor = [1, 1, 1, 1]; // Color for fragment shader
}
```

3. Define a `draw` function for `Renderable`.

```
Renderable.prototype.draw = function() {
    var gl = gEngine.Core.getGL();
    this.mShader.activateShader(this.mColor);
    gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);
};
```

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

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

```
Renderable.prototype.setColor = function(color) { this.mColor = color; };
Renderable.prototype.getColor = function() { return this.mColor; };
```

5. Finally, as the number of source files in the `src/Engine` continues to increase, it is important to constantly keep this folder organized. In this case, create a folder under `src/Engine` and name it `Core`. Move `Engine_Core.js` and `Engine_VertexBuffer.js` into this new folder. The `src/Engine/Core` folder will store all components that belong to the core of the game engine.

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

Testing the Renderable Object

To test Renderable objects in MyGame, a white instance and a red instance of the object are created and drawn as follows:

```
function MyGame(htmlCanvasID) {
    // Step A: Initialize the webGL Context
    gEngine.Core.initializeWebGL(htmlCanvasID);

    // Step B: Create the shader
    this.mConstColorShader = new SimpleShader(
        "src/GLSLShaders/SimpleVS.glsl", // Path to the VertexShader
        "src/GLSLShaders/SimpleFS.glsl"); // Path to the FragmentShader

    // Step C: Create the Renderable objects:
    this.mWhiteSq = new Renderable(this.mConstColorShader);
    this.mWhiteSq.setColor([1, 1, 1, 1]);
    this.mRedSq = new Renderable(this.mConstColorShader);
    this.mRedSq.setColor([1, 0, 0, 1]);

    // Step D: Draw!
    gEngine.Core.clearCanvas([0, 0.8, 0, 1]); // Clear the canvas

    // Step D1: Draw Renderable objects with the white shader
    this.mWhiteSq.draw();

    // Step D2: Draw Renderable objects with the red shader
    this.mRedSq.draw();
}
```

In the code from `src/MyGame.js`, the `MyGame` constructor is modified to include the following steps:

1. Step A initializes the `gEngine.Core`.
2. Step B creates the `mConstColorShader` based on the `SimpleVS.glsl` and `SimpleFS.glsl`.
3. Step C creates two instances of `Renderable` using the shader and sets the colors of the new `Renderable` objects accordingly.
4. Step D clears the canvas; steps D1 and D2 simply call the respective `draw` functions of the white and red squares. Although both of the squares are drawn, for now you will only be able to see the last drawn square in the canvas. Please refer to the following discussion for the details.

Observations

Run the project and you will notice 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 needs to support the ability to be drawn at different locations, with different sizes and with different orientations so 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.

Matrices as Transform Operators

Before we begin, it is important to recognize that matrices and transformations are general topic areas in mathematics. The following discussion does not attempt to include a comprehensive study of these subjects. Instead, the focus is on the application of a small collection of relevant concepts and operators from the perspective of what the game engine requires (or, rather, how to utilize the operators and not study the theories behind the mathematics). If you are interested in the specifics of matrices and how they relate to computer graphics, please refer to the discussion in Chapter 1 where you can learn more about these topics in depth by delving into relevant books on linear algebra and computer graphics.

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

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

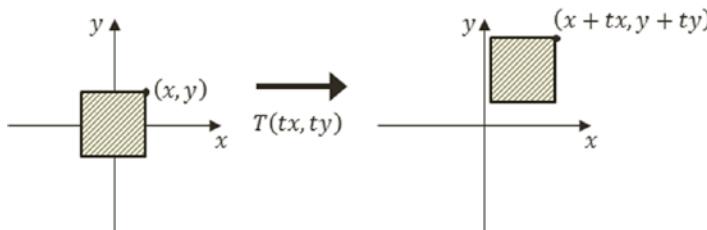
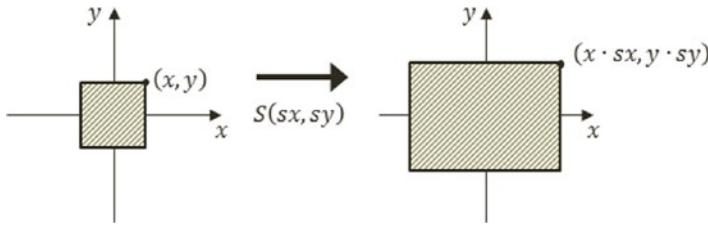
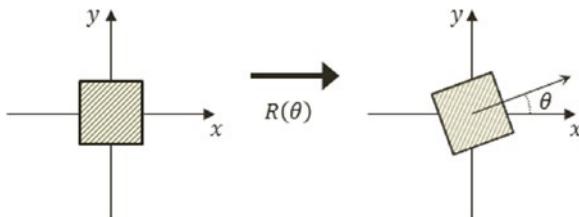


Figure 3-2. Translating a square by $T(tx, ty)$

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

**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:

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

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:

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

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

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

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

Concatenation of Matrix Operators

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

$$p' = TRSp$$

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

$$M = TRS$$

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

$$p' = Mp$$

The M operator is a convenient and efficient way to record and re-apply the results of multiple operators.

Finally, notice that when working with transformation operators, the order of precedence 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:

$$ST \neq TS$$

The glMatrix Library

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.

glMatrix

Javascript Matrix and Vector library for High Performance WebGL apps

[View the Project on GitHub](#)
to/glmatrix

[Download ZIP File](#) [Download TAR Ball](#) [View On GitHub](#)

glMatrix

Javascript has evolved into a language capable of handling realtime 3D graphics, via WebGL, and computationally intensive tasks such as physics simulations. These types of applications demand high performance vector and matrix math, which is something that Javascript doesn't provide by default. glMatrix to the rescue!

glMatrix is designed to perform vector and matrix operations stupidly fast! By hand-tuning each function for maximum performance and encouraging efficient usage patterns through API conventions, glMatrix will help you get the most out of your browsers Javascript engine.

Documentation

Documentation for all glMatrix functions can be found [here](#)

What's new in 2.0?

glMatrix 2.0 is the result of a lot of excellent feedback from the community, and features:

- Revamped and consistent API (not backward compatible with 1.x, sorry!)
- New functions for each type, based on request.

Figure 3-5. Downloading the glMatrix library

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

3. Just like any of your own JavaScript source files, remember to load this new source file in `index.html` by adding the following:

```
<script type="text/javascript" src="src/lib/gl-matrix.js"></script>
```

The Matrix Transform Project

This project introduces and demonstrates how to use transformation matrices as operators to manipulate the position, size, and orientation of Renderable objects drawn on the canvas. In this way, a Renderable can now be drawn to any location, with any size and any orientation. Figure 3-6 shows the output of running the Matrix Transform project. The source code to this project is defined in the Chapter3/3.2.MatrixTransform folder.

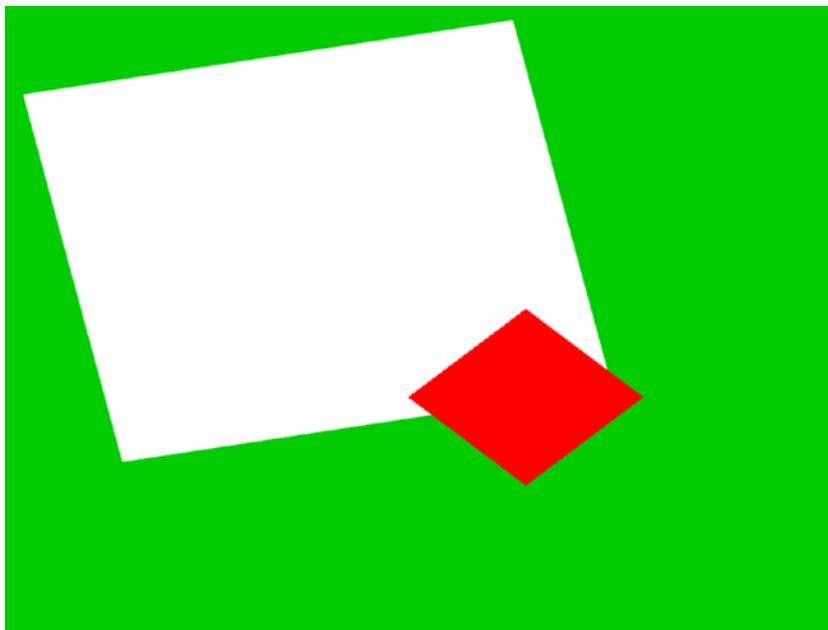


Figure 3-6. Running the Matrix Transform project

The goals of the project are as follows:

- To introduce transformation matrices as operators for drawing a Renderable
- To understand how to work with the transform operators to manipulate a Renderable

Modify the Vertex Shader to Support Transforms

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.

You will continue working with the previous project to support the transformation operator in the vertex shader.

1. Edit `SimpleVS.gls1` to declare a uniform 4×4 matrix.

```
// to transform the vertex position
uniform mat4 uModelTransform;
```

Recall that the `uniform` keyword in a GLSL shader declares a variable with values that do not change for all the vertices within that shader. In this case, the `uModelTransform` variable is the transform operator, and it maintains the operator values for all vertices of the square.

Note GLSL uniform variable names always begin with a *u*.

2. In the main() function, apply the `uModelTransform` to each vertex position in the shader.

```
gl_Position = uModelTransform * vec4(aSquareVertexPosition, 1.0);
```

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

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

Modify SimpleShader to Load the Transform Operator

Follow these steps:

1. Edit `SimpleShader.js` and add an instance variable to hold the reference to the `uModelTransform` matrix in the vertex shader.

```
this.mModelTransform = null;
```

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

```
// Step G: Gets a reference to the uniform variables:  
//         uPixelColor and uModelTransform  
this.mPixelColor = gl.getUniformLocation(this.mCompiledShader, "uPixelColor");  
this.mModelTransform = gl.getUniformLocation(this.mCompiledShader,  
"uModelTransform");
```

3. Add a new function to `SimpleShader` to load the transform operator to the vertex shader.

```
// Loads per-object model transform to the vertex shader  
SimpleShader.prototype.loadObjectTransform = function(modelTransform) {  
    var gl = gEngine.Core.getGL();  
    gl.uniformMatrix4fv(this.mModelTransform, false, modelTransform);  
};
```

The `gl.uniformMatrix4fv()` function copies `modelTransform` to the vertex shader location identified by `mModelTransform` or the `uModelTransform` operator in the vertex shader.

Modify Renderable Object to Set the Transform Operator

Edit `Renderable.js` to modify the `draw()` function to receive a transform operator as a parameter, and after activating the proper GLSL shader with the `mShader.activateShader()` function, send this operator into the shader before the actual drawing operation.

```
Renderable.prototype.draw = function(modelTransform) {
    var gl = gEngine.Core.getGL(); // always activate the shader first!
    this.mShader.activateShader(this.mColor);
    this.mShader.loadObjectTransform(modelTransform);
    gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);
};
```

In this way, when the vertices of the unit square are processed by the vertex shader, the `uModelTransform` will contain the proper operator for transforming the vertices and thus drawing the square in the desired location.

Testing the Transforms

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

1. Edit `MyGame.js`; after step D, instead of activating and drawing the two squares, replace steps D1 and D2 to create a new identity transform operator.

```
// create a new identify transform operator
var xform = mat4.create();
```

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

```
// Step E: compute the white square transform
mat4.translate(xform, xform, vec3.fromValues(-0.25, 0.25, 0.0));
mat4.rotateZ(xform, xform, 0.2); // rotation is in radian
mat4.scale(xform, xform, vec3.fromValues(1.2, 1.2, 1.0));
```

```
// Step F: draw the white square with the computed transform
this.mWhiteSq.draw(xform);
```

Step E 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. The concatenation order applies the scaling operator first, followed by rotation, with translation being the last operation, or `xform=TRS`. In step F after the shader is activated, the `Renderable` object is drawn with the `xform` operator or a 1.2×1.2 white rectangle slightly rotated and located somewhat to the upper left from the center.

3. Finally, steps G and H are used to define and draw the red square.

```
// Step G: compute the red square transform
mat4.identity(xform); // restart
mat4.translate(xform, xform, vec3.fromValues(0.25, -0.25, 0.0));
mat4.rotateZ(xform, xform, -0.785); // rotation of about -45-degrees
mat4.scale(xform, xform, vec3.fromValues(0.4, 0.4, 1.0));
```

```
// Step H: draw the red square with the computed transform
this.mRedSq.draw(xform);
```

Step G defines the `xform` operator that draws a 0.4×0.4 square that is rotated by 45 degrees and located slightly toward the lower right from the center of the canvas.

Observations

Run the project, and you should see the corresponding white and red rectangles drawn on the canvas. You can gain some intuition of the operators by changing the values; for example, move and scale the squares to different locations with different sizes. You can try changing the order of concatenation by moving the corresponding line of code; for example, move `mat4.scale()` to before `mat4.translate()`. You will notice that, in general, the transformed results do not correspond to your intuition. In this book, you will always apply the transformation operators in the fixed TRS order.

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

Encapsulating the Transform Operator

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

The Transform Objects Project

This project defines the `Transform` object to provide a logical interface for manipulating transformation operators and to hide the details of computing matrix transformation operators. Figure 3-7 shows the output of running the Matrix Transform project; notice that the output of this project is identical to that from the previous project. The source code to this project is defined in the [Chapter3/3.3.TransformObjects](#) folder.

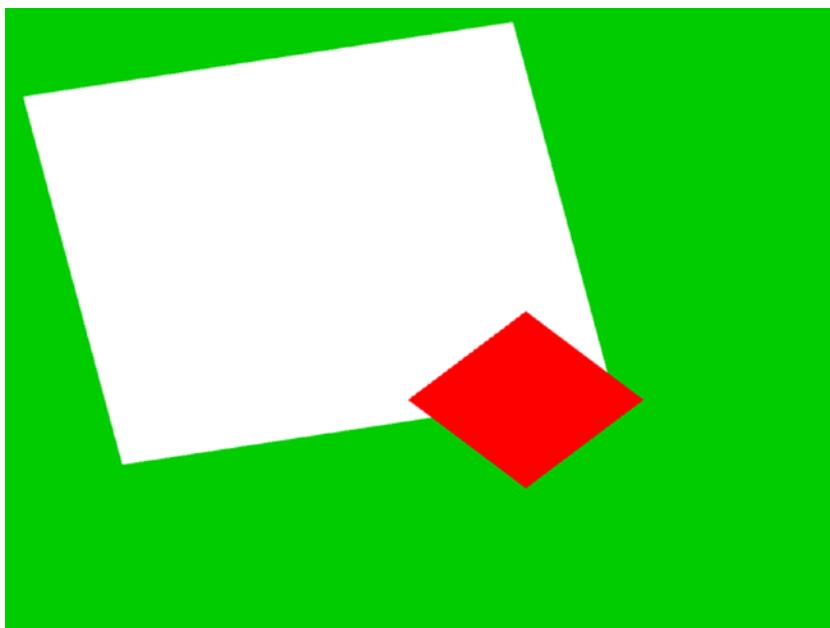


Figure 3-7. Running the Transform Objects project

The goals of the project are as follows:

- To create the `Transform` object so it can encapsulate the matrix transformation functionality
- To integrate the `Transform` object into the game engine
- To demonstrate how to work with the `Transform` object

The Transform Object

Continue working with the previous project.

1. Define the `Transform` object in the game engine by creating a new source code file in the `src/Engine` folder, and name the file `Transform.js`. Remember to load the new source file in `index.html`.
2. Add the constructor for `Transform`.

```
function Transform() {
    this.mPosition = vec2.fromValues(0, 0); // translation operator
    this.mScale = vec2.fromValues(1,1);      // Scaling operator
    this.mRotationInRad = 0.0;               // Rotation in radians!
};
```

These instance variables store the values for the corresponding operators: `mPosition` for translation, `mScale` for scaling, and `mRotationInRad` for rotation.

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

```
// Position getters and setters
Transform.prototype.setPosition = function(xPos,yPos) { this.setXPos(xPos);
    this.setYPos(yPos); };
Transform.prototype.getPosition = function() { return this.mPosition; };
// ... additional get and set functions for position not shown
// Size setters and getters
Transform.prototype.setSize = function(width, height) { this.setWidth(width);
    this.setHeight(height); };
Transform.prototype.getSize = function(){ return this.mScale; };
// ... additional get and set functions for size not shown
// Rotation getters and setters
Transform.prototype.setRotationInRad = function(rotationInRadians) {
    this.mRotationInRad = rotationInRadians;
    while (this.mRotationInRad > (2*Math.PI))
        this.mRotationInRad -= (2*Math.PI);
};
Transform.prototype.setRotationInDegree = function (rotationInDegree)
    { this.setRotationInRad(rotationInDegree * Math.PI/180.0); };
// ... additional get and set functions for rotation not shown
```

- Add a function to compute and return the concatenated transform operator, TRS.

```
Transform.prototype.getXform = function() {
    // Creates a blank identity matrix
    var matrix = mat4.create();

    // Step 1: compute translation, for now z is always at 0.0
    mat4.translate(matrix, matrix, vec3.fromValues(this.getXPos(),
        this.getYPos(), 0.0));
    // Step 2: concatenate with rotation.
    mat4.rotateZ(matrix, matrix, this.getRotationInRad());
    // Step 3: concatenate with scaling
    mat4.scale(matrix, matrix, vec3.fromValues(this.getWidth(),
        this.getHeight(), 1.0));
    return matrix;
};
```

This code is similar to steps E and G in `MyGame.js` from the previous project. The concatenated operator TRS performs scaling first, followed by rotation, and last by translation.

Transformable Renderable Objects

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

- Edit `Renderable.js` and add a `Transform` instance variable.

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

- Define an accessor for the transform operator.

```
Renderable.prototype.getXform = function() { return this.mXform; }
```

- Modify the `draw()` function to load the `mXform` operator to the vertex shader before sending the vertex positions of the unit square.

```
Renderable.prototype.draw = function() {
    var gl = gEngine.Core.getGL();
    this.mShader.activateShader(this.mColor);
    // always activate the shader first!
    this.mShader.loadObjectTransform(this.mXform.getXform());
    gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);
};
```

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

Modify Drawing to Support Transform Object

To test the `Transform` object and the modified `Renderable` object, the `MyGame` constructor can be modified to set the transform operators in each of the `Renderable` objects accordingly.

```

// Step E: 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 F: draws the white square (transform behavior in the object)
this.mWhiteSq.draw();

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

```

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

View, Projection, and Viewports

When designing and building a video game, the game programmers must be able to focus on the intrinsic logic and presentation of the games themselves. To facilitate this, it is important that the programmer 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 itself. 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. 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 unit-less $n \times n$ 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 the corresponding View and Projection matrices.

In all cases, to support 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 1:1 drawing range of the WebGL in order to draw geometries proportionally.

Coordinate Systems and Transformations

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

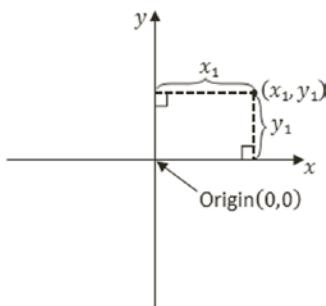


Figure 3-8. Working with a 2D Cartesian coordinate system

Modeling and Normalized Device Coordinate Systems

So far in this book, you have experience with two distinct coordinate systems. The first is the coordinate system that defines the vertices for the 1×1 square in the vertex buffer. This is referred to as the Modeling Coordinate System, which defines the Model Space. The Model Space is unique for each geometric object, as in the case of the unit square. The Model Space is defined to describe the geometry of a single model. Second, the coordinate space that the WebGL draws to, the one where the x/y-axes ranges are bounded to ± 1.0 , is defined by the Normalized Device Coordinates (NDC) System. As you have experienced, WebGL always draws to NDC space and shows the results in the canvas.

The Modeling transform, typically defined by a matrix transformation operator, is the operation that transforms geometries from its Model Space into another coordinate space that is convenient for drawing. In the previous project, the `uModelTransform` variable in `SimpleVS.glsl` is the Modeling transform. As illustrated in Figure 3-9, in that case, the Modeling transform transformed the unit square into the WebGL's NDC space.

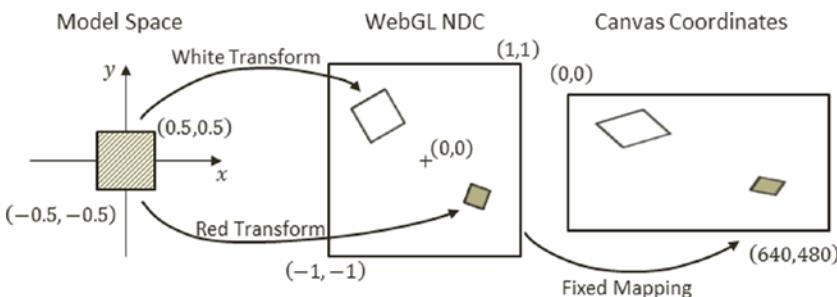


Figure 3-9. Transforming the square from Model to NDC space

The World Coordinate System

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 describing a convenient World Space was introduced to remedy 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.

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 dimensions or width and height. The reference position can be either the lower-left corner or the center of the WC.

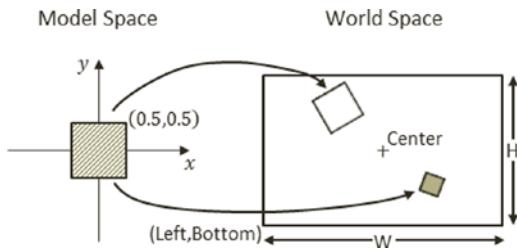


Figure 3-10. Working with a World Coordinate (WC) System

The WC is a convenient coordinate system for designing games. However, it is not the space that WebGL draws to. For this reason, it is important to transform WC to NDC. This transform is referred to as the View-Projection transform. To accomplish this transform, you will take advantage of two important functions provided by the `glMatrix` library.

```
mat4.lookAt(viewMatrix,
    [cx, cy, 10], // (cx,cy) is center of the WC
    [cx, cy, 0],
    [0, 1, 0]); // orientation

mat4.ortho(projMatrix,
    -distToLeft, // distant from (cx,cy) to left of WC
    distToRight, // distant from (cx,cy) to right of WC
    -distToBottom, // distant from (cx,cy) to bottom of WC
    distToTop, // distant from (cx,cy) to top of WC
    0, // the z-distant to near plane
    1000 // the z-distant to far plane
);
```

As shown, the `mat4.lookAt()` function defines the center, and the `mat4.ortho()` function defines the dimension of the WC. In both cases, results are returned as matrix operators: the View matrix (`viewMatrix`) and the Projection matrix (`projMatrix`). Since this book focuses on the 2D space, for now you do not need to be concerned with the z-component of the parameters. Notice that the distances in the `mat4.ortho()` function are signed quantities; in other words, the distance to the right/up is positive, while the distance to the left/bottom is negative. For example, 5 units to the left is -5.

The View-Projection transform operator, `vpMatrix`, is simply the concatenation of the View and Projection matrices, the results from the `mat4.lookAt()` and `mat4.ortho()` functions.

$$\text{vpMatrix} = \text{projMatrix} \times \text{viewMatrix}$$

Note Interested readers can consult a 3D computer graphics reference book to learn more about the View and Projection transforms.

The Viewport

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

```
gl.viewport(
  x,      // x position of bottom-left corner of the area to be drawn
  y,      // y position of bottom-left corner of the area to be drawn
  width, // width of the area to be drawn
  height // height of the area to be drawn
);
```

The `gl.viewport()` function defines a viewport for all subsequent drawings. Figure 3-11 illustrates the View-Projection transform and drawing with a viewport.

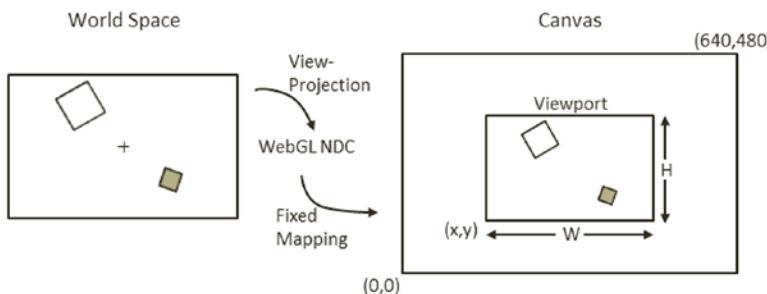


Figure 3-11. Working with the WebGL viewport

The View Projection and Viewport Project

This project demonstrates how to use a View-Projection 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 View Projection and Viewport project. The source code to this project is defined in the Chapter3/3.4.ViewProjectionAndViewport folder.

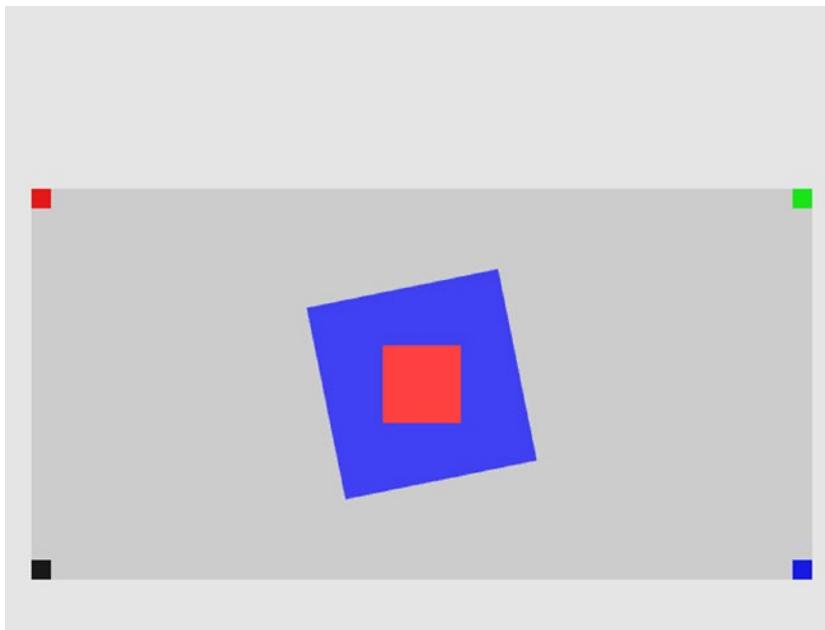


Figure 3-12. Running the View Projection and Viewport project

The goals of the project are as follows:

- To understand the different coordinate systems
- To experience working with a WebGL viewport, define and draw to subregions within the canvas
- To understand the View and Projection transformations
- To begin drawing to the user-defined World Coordinate System

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

Modifying the Vertex Shader to Support the View-Projection Transform

Relatively minor changes are required to add the support for the View-Projection transform.

1. Edit SimpleVS.gls1 to add a new uniform matrix operator to represent the View-Projection transform.

```
uniform mat4 uViewProjTransform;
```

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

```
gl_Position = uViewProjTransform * uModelTransform *
    vec4(aSquareVertexPosition, 1.0);
```

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

Modifying SimpleVertex to Support the View-Projection Transform

The `SimpleShader` object must be modified to pass the View-Projection matrix to the vertex shader.

1. Edit `SimpleShader.js` and in the constructor add an instance variable for storing the reference to the View-Projection transform operator in `SimpleVS.glsL`.

```
this.mViewProjTransform = null;
```

2. At the end of the `SimpleShader` constructor, retrieve the reference to the View-Projection transform operator after retrieving those for the `uModelTransform` and `uPixelColor`.

```
// Step G: references: uniforms: uModelTransform, uPixelColor, and
//           uViewProjTransform
this.mModelTransform = gl.getUniformLocation(this.mCompiledShader,
    "uModelTransform");
this.mPixelColor = gl.getUniformLocation(this.mCompiledShader, "uPixelColor");
this.mViewProjTransform = gl.getUniformLocation(
    this.mCompiledShader, "uViewProjTransform");
```

3. Modify the `activateShader` function to receive a View-Projection matrix and pass it to the shader, as shown here:

```
SimpleShader.prototype.activateShader = function(vpMatrix) {
    var gl = gEngine.Core.getGL();
    gl.useProgram(this.mCompiledShader);
    gl.uniformMatrix4fv(this.mViewProjTransform, false, vpMatrix);
    gl.enableVertexAttribArray(this.mShaderVertexPositionAttribute);
    gl.uniform4fv(this.mPixelColor, pixelColor);
};
```

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

Modifying RenderObject to Support the View-Projection Transform

Recall that shaders are activated in the `Renderable` object's `draw()` function; as such, `Renderable` must also be modified to receive and pass `vpMatrix` to activate the shaders.

```
Renderable.prototype.draw = function(pixelColor, vpMatrix) {
    var gl = gEngine.Core.getGL();
    this.mShader.activateShader(this.mColor, vpMatrix); // activate first!
    this.mShader.loadObjectTransform(this.mXform.getXform());
    gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);
};
```

Testing the View-Projection Transform and the Viewport

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

Designing the Scene

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 corners of the WC.

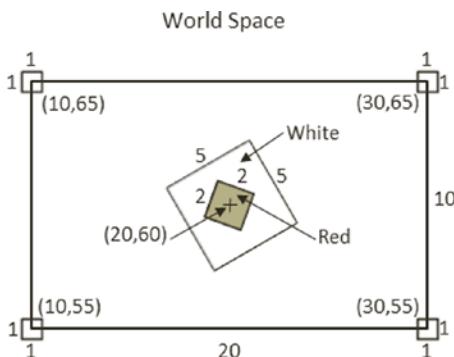


Figure 3-13. Designing a WC to support drawing

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 matches that of the 600:300.

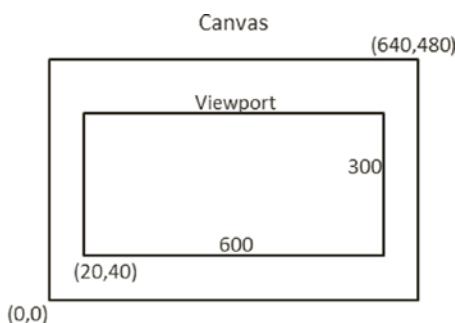


Figure 3-14. Drawing the WC to the viewport

Implementing the Design

The MyGame object will be modified to implement the design.

1. Edit MyGame.js to initialize the WebGL, create a constant color shader, and create six Renderable objects (two to be drawn at the center, with four at each corner of the WC) with corresponding colors.

```
// Step A: Initialize the webGL Context
gEngine.Core.initializeWebGL(htmlCanvasID);
var gl = gEngine.Core.getGL();

// Step B: Create the shader
this.mConstColorShader = new SimpleShader(
    "src/GLSLShaders/SimpleVS.glsL",           // Path to the VertexShader
    "src/GLSLShaders/SimpleFS.glsL");           // Path to the FragmentShader

// Step C: Create the Renderable objects:
this.mBlueSq = new Renderable(this.mConstColorShader);
this.mBlueSq.setColor([0.25, 0.25, 0.95, 1]);
this.mRedSq = new Renderable(this.mConstColorShader);
this.mRedSq.setColor([1, 0.25, 0.25, 1]);
this.mTLSq = new Renderable(this.mConstColorShader);
this.mTLSq.setColor([0.9, 0.1, 0.1, 1]);        // Top-Left shows red
this.mTRSq = new Renderable(this.mConstColorShader);
this.mTRSq.setColor([0.1, 0.9, 0.1, 1]);        // Top-Right shows green
this.mBRSq = new Renderable(this.mConstColorShader);
this.mBRSq.setColor([0.1, 0.1, 0.9, 1]);        // Bottom-Right shows blue
this.mBLSq = new Renderable(this.mConstColorShader);
this.mBLSq.setColor([0.1, 0.1, 0.1, 1]);        // Bottom-Left shows dark gray
```

2. Clear the entire canvas, set up the viewport, and clear the viewport to a different color.

```
// Step D: Clear the entire canvas first
gEngine.Core.clearCanvas([0.9, 0.9, 0.9, 1]);    // Clear the canvas

// Step E: Setting up Viewport
// Step E1: Set up the viewport: area on canvas to be drawn
gl.viewport(
    20,      // x position of bottom-left corner of the area to be drawn
    40,      // y position of bottom-left corner of the area to be drawn
    600,     // width of the area to be drawn
    300);   // height of the area to be drawn
);
// Step E2: set up the corresponding scissor area to limit clear area
gl.scissor(
    20,      // x position of bottom-left corner of the area to be drawn
    40,      // y position of bottom-left corner of the area to be drawn
    600,     // width of the area to be drawn
    300);   // height of the area to be drawn
);
```

```
// Step E3: enable the scissor area, clear, and then disable the scissor area
gl.enable(gl.SCISSOR_TEST);
    gEngine.Core.clearCanvas([0.8, 0.8, 0.8, 1.0]); // clear the scissor area
gl.disable(gl.SCISSOR_TEST);
```

Step E1 defines the viewport, and step E2 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. Define the WC by setting up the View-Projection transform operator.

```
// Step F: Set up View and Projection matrices
var viewMatrix = mat4.create();
var projMatrix = mat4.create();

// Step F1: define the view matrix
mat4.lookAt(viewMatrix,
    [20, 60, 10], // camera position
    [20, 60, 0], // look at position
    [0, 1, 0]); // orientation

// Step F2: define the projection matrix
mat4.ortho(projMatrix,
    -10, // distant to left of WC
    10, // distant to right of WC
    -5, // distant to bottom of WC
    5, // distant to top of WC
    0, // z-distant to near plane
    1000 // z-distant to far plane
);

// Step F3: concatenate to form the View-Projection operator
var vpMatrix = mat4.create();
mat4.multiply(vpMatrix, projMatrix, viewMatrix);
```

In step F1, the `mat4.lookAt()` function defines the center of WC to be located at (20,60). Step F2 defines the distances from the center position to the left and right boundaries to be 10 units and to the top and bottom boundaries to be 5 units away. Together, these define the WC as follows:

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

Recall that the order of multiplication is important and that the order of `projMatrix` and `viewMatrix` should not be swapped.

4. Set up the slightly rotated 5x5 blue square at the center of WC and draw with the `vpMatrix` operator.

```
// Step G: Draw the blue square
// Centre Blue, slightly rotated square
this.mBlueSq.getXform().setPosition(20, 60);
this.mBlueSq.getXform().setRotationInRad(0.2); // In Radians
this.mBlueSq.getXform().setSize(5, 5);
this.mBlueSq.draw(vpMatrix);
```

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

```
// Step H: Draw with the red shader
// centre red square
this.mRedSq.getXform().setPosition(20, 60);
this.mRedSq.getXform().setSize(2, 2);
this.mRedSq.draw(vpMatrix);

// top left
this.mTLSq.getXform().setPosition(10, 65);
this.mTLSq.draw(vpMatrix);

// top right
this.mTRSq.getXform().setPosition(30, 65);
this.mTRSq.draw(vpMatrix);

// bottom right
this.mBRSq.getXform().setPosition(30, 55);
this.mBRSq.draw(vpMatrix);

// bottom left
this.mBLSq.getXform().setPosition(10, 55);
this.mBLSq.draw(vpMatrix);
```

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.

It is now possible to define any convenient WC system and any rectangular subregions of the canvas for drawing. With the Modeling and View-Projection 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 create abstraction to hide the details of View-Projection matrix computation.

The Camera

The View-Projection transform allows the definition of a WC to draw from. 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 viewfinder are the dimensions of WC. With this analogy, the act of taking the photograph is equivalent to computing the image drawing of each object in the WC. Lastly, the viewport describes the location to display the computed image.

The Camera Objects Project

This project demonstrates how to abstract the View-Projection transform and the viewport to hide the details of matrix operator computation and WebGL configurations. Figure 3-15 shows the output of running the Camera Objects project; notice the output of this project is identical to that from the previous project. The source code to this project is defined in the [Chapter3/3.5.CameraObjects](#) folder.

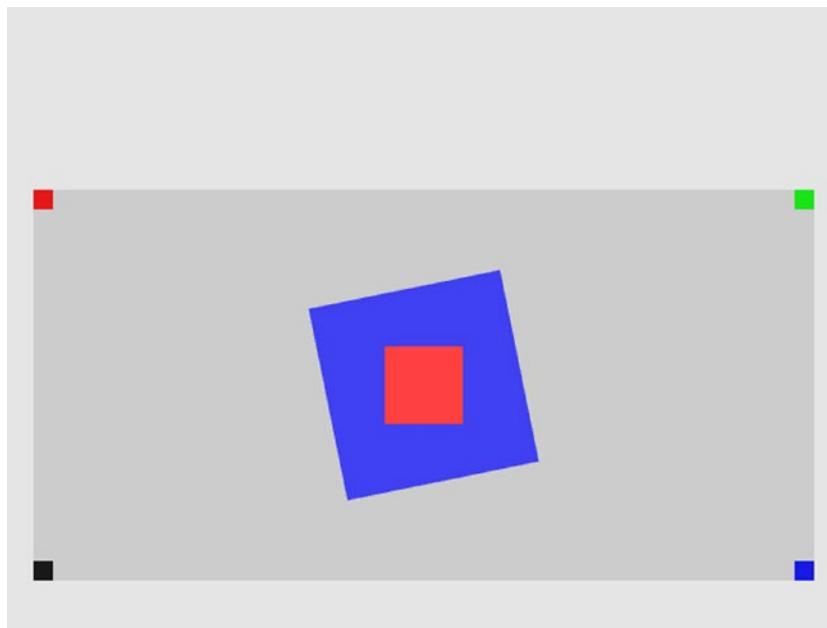


Figure 3-15. Running the Camera Objects project

The goals of the project are as follows:

- To define the `Camera` object to encapsulate the definition of WC and the viewport functionality
- To integrate the `Camera` object into the game engine
- To demonstrate how to work with the `Camera` object

The Camera Object

The Camera object basically encapsulates the functionality defined by the View-Projection and viewport setup code in the MyGame constructor from the previous example. A clean and reusable object design can be completed with appropriate getter and setter functions.

1. Define the Camera object in the game engine by creating a new source file in the `src/Engine` folder, and name the file `Camera.js`. Remember to load the new source file in `index.html`.
2. Add the constructor for Camera.

```
function Camera(wcCenter, wcWidth, viewportArray) {
    // WC and viewport position and size
    this.mWCCenter = wcCenter;
    this.mWCWidth = wcWidth;
    this.mViewport = viewportArray; // [x, y, width, height]
    this.mNearPlane = 0;
    this.mFarPlane = 1000;

    // transformation matrices
    this.mViewMatrix = mat4.create();
    this.mProjMatrix = mat4.create();
    this.mVPMatrix = mat4.create();

    // background color
    this.mBgColor = [0.8, 0.8, 0.8, 1]; // RGB and Alpha
}
```

The Camera object defines the WC center and width, the viewport, and the View-Projection transform operator. Take note of the following:

- a. The `mWCPosition` is a `vec2` (`vec2` is defined in the `glMatrix` library). It is a float array of two elements. The first element, index position 0, of `vec2` is the x, and the second element, index position 1, is the y position.
 - b. The four elements of the `viewportArray` are the x and y positions of the lower-left corner and the width and height of the viewport, in that order. This compact representation of the viewport keeps the number of instance variables to a minimum and helps keep the Camera object manageable.
 - c. `mBgColor` is an array of four floats representing the red, green, blue, and alpha components of a color.
 - d. The `mWCWidth` is the width of the WC. To guarantee a matching aspect ratio between WC and the viewport, the height of the WC is always computed from the aspect ratio of the viewport and `mWCWidth`.
3. Add getters and setters for the instance variables.

```
// Setter/getter of WC and viewport
Camera.prototype.setWCCenter = function(xPos, yPos) {
    this.mWCCenter[0] = xPos;
    this.mWCCenter[1] = yPos;
};
```

```

Camera.prototype.getWCCenter = function() { return this.mWCCenter; };
Camera.prototype.setWCWidth = function(width) { this.mWCWidth = width; };

Camera.prototype.setViewport = function(viewportArray) {
    this.mViewport = viewportArray;
};
Camera.prototype.getViewport = function() { return this.mViewport; };

Camera.prototype.setBackgroundColor = function(newColor) {
    this.mBgColor = newColor;
};
Camera.prototype.getBackgroundColor = function() { return this.mBgColor; };

// Getter for the View-Projection transform operator
Camera.prototype.getVPMatrix = function() { return this.mVPMatrix; };

```

4. Create a function to compute the View-Projection operator for this Camera:

```

// Initializes the camera to begin drawing
Camera.prototype.setupViewProjection = function() {
    var gl = gEngine.Core.getGL();
    // Step A: Configure the viewport
    // ... details to follow

    // Step B: define the View-Projection matrix
    // ... details to follow
};

```

Note that this function is called `setupViewProjection()` because it configures WebGL to draw to the desire viewport and sets up the View-Projection transform operator. The following steps explain the details of steps A and B.

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

```

// Step A: Set up and clear the Viewport
// Step A1: Set up the viewport: area on canvas to be drawn
gl.viewport(this.mViewport[0], // x position of bottom-left corner
            this.mViewport[1], // y position of bottom-left corner
            this.mViewport[2], // width of the area to be drawn
            this.mViewport[3]); // height of the area to be drawn

// Step A2: set up the corresponding scissor area to limit clear area
gl.scissor( this.mViewport[0], // x position of bottom-left corner
            this.mViewport[1], // y position of bottom-left corner
            this.mViewport[2], // width of the area to be drawn
            this.mViewport[3]); // height of the area to be drawn

// Step A3: set the color to be clear to black
gl.clearColor(this.mBgColor[0], this.mBgColor[1],
              this.mBgColor[2], this.mBgColor[3]); // set the color to be cleared

```

```
// Step A4: enable and clear the scissor area
gl.enable(gl.SCISSOR_TEST);
gl.clear(gl.COLOR_BUFFER_BIT);
gl.disable(gl.SCISSOR_TEST);
```

Notice the similarity of these steps to the viewport setup code in `MyGame` of the previous example. The only difference is the use of the corresponding `Camera` instance variables.

6. The code to set up the View-Projection transform operator under step B is as follows:

```
// Step B: Set up the View-Projection transform operator
// Step B1: define the view matrix
mat4.lookAt(this.mViewMatrix,
    [this.mWCCenter[0], this.mWCCenter[1], 10], // WC center
    [this.mWCCenter[0], this.mWCCenter[1], 0], //
    [0, 1, 0]); // orientation

// Step B2: define the projection matrix
var halfWCWidth = 0.5 * this.mWCWidth;
var halfWCHeight = halfWCWidth * this.mViewport[3] / this.mViewport[2];
// WCHeight = WCWidth * viewportHeight / viewportWidth
mat4.ortho(this.mProjMatrix,
    -halfWCWidth, // distant to left of WC
    halfWCWidth, // distant to right of WC
    -halfWCHeight, // distant to bottom of WC
    halfWCHeight, // distant to top of WC
    this.mNearPlane, // z-distant to near plane
    this.mFarPlane // z-distant to far plane);

// Step B3: concatenate view and project matrices
mat4.multiply(this.mVPMMatrix, this.mProjMatrix, this.mViewMatrix);
```

Once again, this code is similar to that from the previous example. In addition, take note that to guarantee a matching aspect ratio between WC and viewport, in step B2, the WC height, `halfWCHeight`, is computed based on the WC width, `mWCWidth`, and the aspect ratio of the viewport, which is height divided by width (`mViewport[3]/mViewport[2]`).

Testing the Camera

With the `Camera` object properly defined, testing it from `MyGame.js` is straightforward.

1. Edit `MyGame.js`; after the initialization of WebGL, create an instance of the `Camera` object with settings that define the WC and viewport from the previous project.

```
function MyGame(htmlCanvasID) {
    // Step A: Initialize the WebGL Context
    gEngine.Core.initializeWebGL(htmlCanvasID);
```

```
// Step B: Setup the camera
this.mCamera = new Camera(
    vec2.fromValues(20, 60), // center of the WC
    20, // width of WC
    [20, 40, 600, 300] // viewport (orgX, orgY, width, height)
);
...
```

2. Continue with the creation of the SimpleShader, the six Renderable objects, and the clearing of the canvas.

```
// Step C: Create the shader
this.mConstColorShader = new SimpleShader(
    "src/GLSLShaders/SimpleVS.gls", // Path to the VertexShader
    "src/GLSLShaders/SimpleFS.gls"); // Path to the simple FragmentShader

// Step D: Create the Renderable objects:
this.mBlueSq = new Renderable(this.mConstColorShader);
this.mBlueSq.setColor([0.25, 0.25, 0.95, 1]);
this.mRedSq = new Renderable(this.mConstColorShader);
this.mRedSq.setColor([1, 0.25, 0.25, 1]);
this.mTLSq = new Renderable(this.mConstColorShader);
this.mTLSq.setColor([0.9, 0.1, 0.1, 1]);
this.mTRSq = new Renderable(this.mConstColorShader);
this.mTRSq.setColor([0.1, 0.9, 0.1, 1]);
this.mBRSq = new Renderable(this.mConstColorShader);
this.mBRSq.setColor([0.1, 0.1, 0.9, 1]);
this.mBLSq = new Renderable(this.mConstColorShader);
this.mBLSq.setColor([0.1, 0.1, 0.1, 1]);

// Step E: Clear the canvas
gEngine.Core.clearCanvas([0.9, 0.9, 0.9, 1]); // Clear the canvas
```

3. Now, draw with the Camera object.

```
// Step F: Starts the drawing by activating the camera
this.mCamera.setupViewProjection();
var vpMatrix = this.mCamera.getVPMMatrix();

// Step G: Draw the blue square
// Centre Blue, slightly rotated square
this.mBlueSq.getXform().setPosition(20, 60);
this.mBlueSq.getXform().setRotationInRad(0.2); // In Radians
this.mBlueSq.getXform().setSize(5, 5);
this.mBlueSq.draw(vpMatrix);

// Step H: Draw the center and the corner squares
// centre red square
this.mRedSq.getXform().setPosition(20, 60);
this.mRedSq.getXform().setSize(2, 2);
this.mRedSq.draw(vpMatrix);

... rest of the code is identical to the previous project ...
```

Step F calls `mCamera.setupViewProjection()` to compute the View-Projection transform operator. This operator is used to activate the shaders in steps G and H. The rest of the squares' model transform and drawing code are identical to the previous project.

Summary

In this chapter, you learned how to create a system that can support the drawing of many objects. The system is composed of three parts: the objects drawn, the scene drawn to, and how the scene is displayed on the browser's canvas. The objects being drawn are encapsulated by a `Renderable` object, which uses a `Transform` object to define its position, size, and rotation. Position, though, is relative to the scene you are drawing to.

You also learned that objects are all drawn relative to the scene and that a World Space, a convenient coordinate system, can be defined in order to compose the entire scene by utilizing coordinate transformations. Lastly, the View-Projection transformation is used to select which portion of the world to actually display on the canvas within a browser. This was achieved by defining an area that is viewable by the camera and using the viewport functionality provided through WebGL.

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

CHAPTER 4



Implementing Common Components of Video Games

After completing this chapter, you will be able to:

- Control the Renderable object's position, size, and rotation to construct complex movements and animations
- Receive keyboard input from the player 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

Introduction

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.

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 input functionality into the game engine. 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 (for example, 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 a normal human'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 to 60 cycles in a second. Since there is usually one drawing operation in each game loop cycle, the game loop cycle's rate can also be expressed 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

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

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

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 which operations to emphasize and which to skip. 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;
```

```
input();
while( elapsedTime >= UPDATE_TIME_RATE ) {
    update();
    elapsedTime -= UPDATE_TIME_RATE;
}
draw();
}
```

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 is 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 play input response and frames skipped. However, the game logic will continue to be correct.

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.

To ensure focusing on the core game loop operation, `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 updating and drawing the squares accordingly. 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.GameLoop](#) folder.

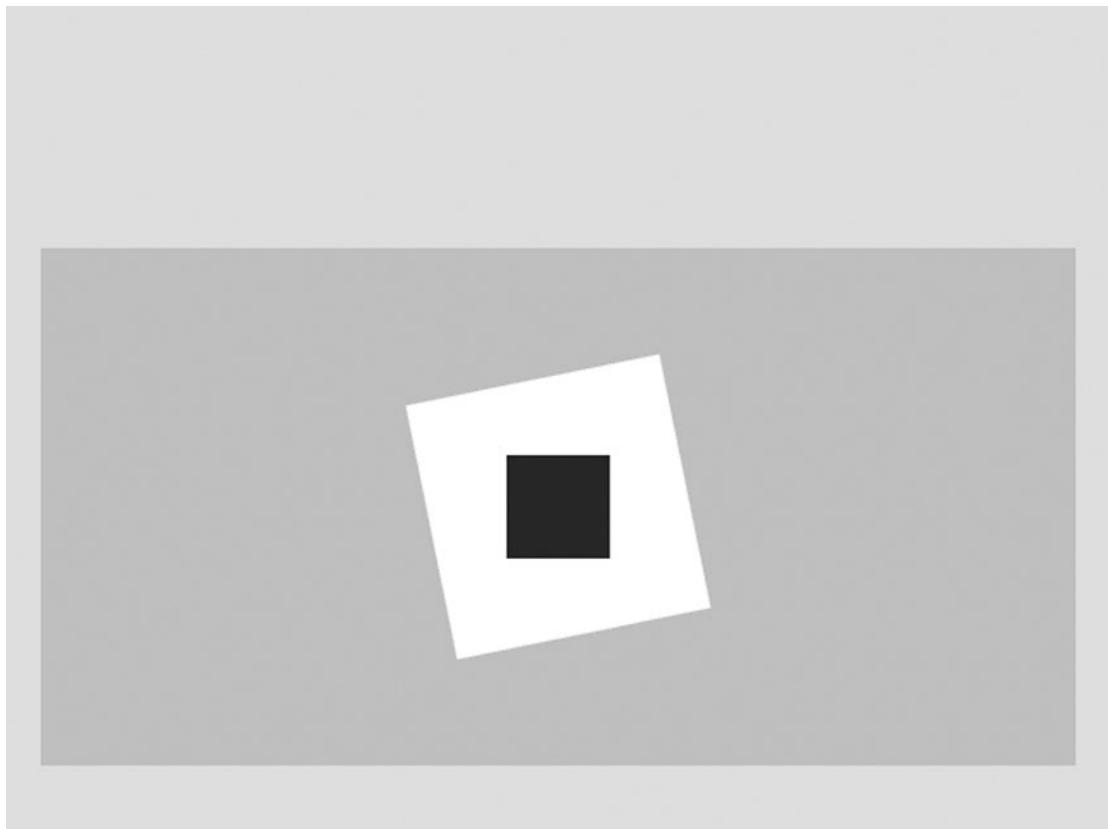


Figure 4-1. Running the Game Loop project

The goals of the project are as follows:

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

Implement the Game Loop Component

The game loop component is a core game engine functionality and thus should be implemented similar to `Engine_Core` or `Engine_VertexBuffer`, as a property of the `gEngine` object in a file defined in the `src/Engine/Core` folder. The actual implementation is similar to the pseudocode listing discussed; for clarity it's shown without the input support for now.

1. Create a new file in the `src/Engine/Core` folder and name the file `Engine_GameLoop.js`. Define the `GameLoop` component following the JavaScript module pattern as follows:

```
var gEngine = gEngine || { };

gEngine.GameLoop = (function() {
    var mPublic = { };
    return mPublic;
}());
```

Tip This book refers to core game engine functionality as engine components and implements each component as an encapsulated property to the `gEngine` object based on this same pattern.

2. Add the following instance variables to keep track of frame rate, processing time, and referencing to the currently running game:

```
var kFPS = 60;           // Frames per second
var kMPF = 1000 / kFPS; // Milliseconds per frame.

// Variables for timing gameloop.
var mPreviousTime;
var mLagTime;
var mCurrentTime;
var mElapsedTime;

// The current loop state (running or should stop)
var mIsLoopRunning = false;

// Reference to game logic
var mMyGame = null;
```

Notice that `kFPS` is the frame per second discussed and that `kMPF` is milliseconds per frame. It is important to maintain the game at an update interval of `kFPS`.

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

```
// This function assumes it is sub-classed from MyGame
var _runLoop = function () {
    if(mIsLoopRunning) {
        // Step A: set up for next call to _runLoop and update input!
        requestAnimationFrame( function(){_runLoop.call(mMyGame);} );
    }
}
```

```

    // Step B: compute elapsed time since last RunLoop was executed
    mcurrentTime = Date.now();
    mElapsedTime = mcurrentTime - mPreviousTime;
    mPreviousTime = mcurrentTime;
    mLagTime += mElapsedTime;

    // Step C: update the game the appropriate number of times.
    //           Update only every Milliseconds per frame.
    //           If lag larger then update frames, update until caught up.
    while ((mLagTime >= kMPF) && mIsLoopRunning) {
        this.update();          // call MyGame.update()
        mLagTime -= kMPF;
    }

    // Step D: now let's draw
    this.draw();      // Call MyGame.draw()
}
};


```

Notice the similarity between the pseudocode examined and the steps B, C, and D of the `_runLoop()` function shown previously. The main difference is that the functionality of the outermost `while` loop is implemented with the `requestAnimationFrame()` function call at step A, where the `_runLoop()` function is set up to be called continuously. More specifically:

- The `requestAnimationFrame()` function registers the `_runLoop()` function with the browser where the browser will call on the next available frame. Notice that each call to the `requestAnimationFrame()` function will result in one execution of the corresponding `_runLoop()` function.
- The syntax of the function passed to `requestAnimationFrame()`, `_runLoop.call(mMyGame)`, binds the 'this' context to `mMyGame`. In this way, the `this.update()` and `this.draw()` function calls within the loop are those defined in the `mMyGame` object.

Note The `mIsLoopRunning` condition of the `while` loop in step C is a redundant check for now. This condition will become important in later sections when `this.update()` can call `GameLoop.stop()` to stop the loop (for example, for level transitions or the end of the game).

- Add the function to start the game loop as follows:

```

var start = function(myGame) {
    mMyGame = myGame;

    // Step A: reset frame time
    mPreviousTime = Date.now();
    mLagTime = 0.0;
}


```

```

    // Step B: remember that loop is now running
    mIsLoopRunning = true;

    // Step C: request _runLoop to start when loading is done
    requestAnimationFrame(function(){_runLoop.call(mMyGame);});
}

```

This function initializes the game loop internal state by setting `mMyGame`, initializing the frame time, and setting the loop running flag to true, and then it starts the game loop by passing the `_runLoop()` function to `requestAnimationFrame()`.

5. Remember to add the `start()` function to the public interface of the `GameLoop` object.

```

var mPublic = {
    start: start
};
return mPublic;

```

Using the Game Loop

To test the game loop implementation, a game class must define the `update()` and `draw()` functions. In this case, the `MyGame` object will also define a constructor and an `initialize()` function.

1. In `MyGame.js`, replace the `MyGame` constructor with the following:

```

function MyGame(htmlCanvasID) {
    // variables of the constant color shader
    this.mConstColorShader = null;

    // variables for the squares
    this.mWhiteSq = null;           // these are the renderable objects
    this.mRedSq = null;

    // The camera to view the scene
    this.mCamera = null;

    // Initialize the webGL Context
    gEngine.Core.initializeWebGL(htmlCanvasID);

    // Initialize the game
    this.initialize();
}

```

2. Add an initialization function to the prototype as follows:

```

MyGame.prototype.initialize = function() {
    // Step A: set up the cameras
    this.mCamera = new Camera(
        vec2.fromValues(20, 60),   // position of the camera
        20,                      // width of camera
        [20, 40, 600, 300]       // viewport (orgX, orgY, width, height)
    );
}

```

```

this.mCamera.setBackgroundColor([0.8, 0.8, 0.8, 1]);
    // sets the background to dark gray

// Step B: create the shader
this.mConstColorShader = new SimpleShader(
    "src/GLSLShaders/SimpleVS.gls1", // Path to the VertexShader
    "src/GLSLShaders/SimpleFS.gls1"); // Path to the FragmentShader

// Step C: Create the renderable objects:
this.mWhiteSq = new Renderable(this.mConstColorShader);
this.mWhiteSq.setColor([1, 1, 1, 1]);
this.mRedSq = new Renderable(this.mConstColorShader);
this.mRedSq.setColor([1, 0, 0, 1]);

// Step D: Initialize the white renderable object: centred, 5x5, rotated
this.mWhiteSq.getXform().setPosition(20, 60);
this.mWhiteSq.getXform().setRotationInRad(0.2); // In Radian
this.mWhiteSq.getXform().setSize(5, 5);

// Step E: Initialize the red renderable object: centered 2x2
this.mRedSq.getXform().setPosition(20, 60);
this.mRedSq.getXform().setSize(2, 2);

// Step F: Start the game loop running
gEngine.GameLoop.start(this);
};

```

The initialization is rather similar to previous examples, where a camera is defined and two squares are set up. The interesting change in this case is the last step, step F, where the game loop is started with `MyGame` as the parameter. Recall that the game loop `_runLoop()` function is set up such that the `update()` and `draw()` functions of `MyGame` will be called.

3. Add an `update()` function to animate a moving white square and a pulsing red square.

```

// The update function, updates the application state. Make sure to _NOT_ draw
// anything from this function!
MyGame.prototype.update = function() {
    // For this very simple game, let's move the white square and pulse the red

    // Step A: move the white square
    var whiteXform = this.mWhiteSq.getXform();
    var deltaX = 0.05;
    if (whiteXform.getXPos() > 30) // this is the right-bound of the window
        whiteXform.setPosition(10, 60);
    whiteXform.incXPosBy(deltaX);
    whiteXform.incRotationByDegree(1);

    // Step B: pulse the red square
    var redXform = this.mRedSq.getXform();

```

```

        if (redXform.getWidth() > 5)
            redXform.setSize(2, 2);
        redXform.incSizeBy(0.05);
    };
}

```

Recall that the `update()` function is called at about 60 FPS, and each time the following happens:

- *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.
- *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.

Since the previous operations are performed continuously at about 60 times a second, you can expect to see the following:

- A white square rotating while moving toward the right and reappearing when it reaches the right boundary
- A red square increasing in size and reducing to a size of 2 when the size reaches 5, thus appearing to be pulsing
- 4. Draw the scene as before by clearing the canvas, setting up the camera, and drawing each square.

```

// This is the draw function, make sure to setup proper drawing environment,
// and more importantly, make sure to _NOT_ change any state.
MyGame.prototype.draw = function() {
    // Step A: clear the canvas
    gEngine.Core.clearCanvas([0.9, 0.9, 0.9, 1.0]); // clear to light gray

    // Step B: Activate the drawing Camera
    this.mCamera.setupViewProjection();

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

    // Step D: Activate the red shader to draw
    this.mRedSq.draw(this.mCamera.getVPMatrix());
};

```

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

Notice that when the loop is running quickly, it is entirely possible for the `_runLoop()` function to be called multiple times within a single kMPF interval. With the given `_runLoop()` implementation, the `draw()` function will be called multiples times without any `update()` function calls. This way, the game loop can end up drawing the same game state multiple times. Please refer to the following references for discussions of supporting extrapolations in the `draw()` function to take advantage of efficient game loops:

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

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

Keyboard Input

It is obvious that proper support to receive player input is important to interactive video games. For a PC, 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 simpler to support at this point in the engine's development. 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.

The Keyboard Support Project

This project examines keyboard input support and incorporates 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.KeyboardSupport](#) folder.

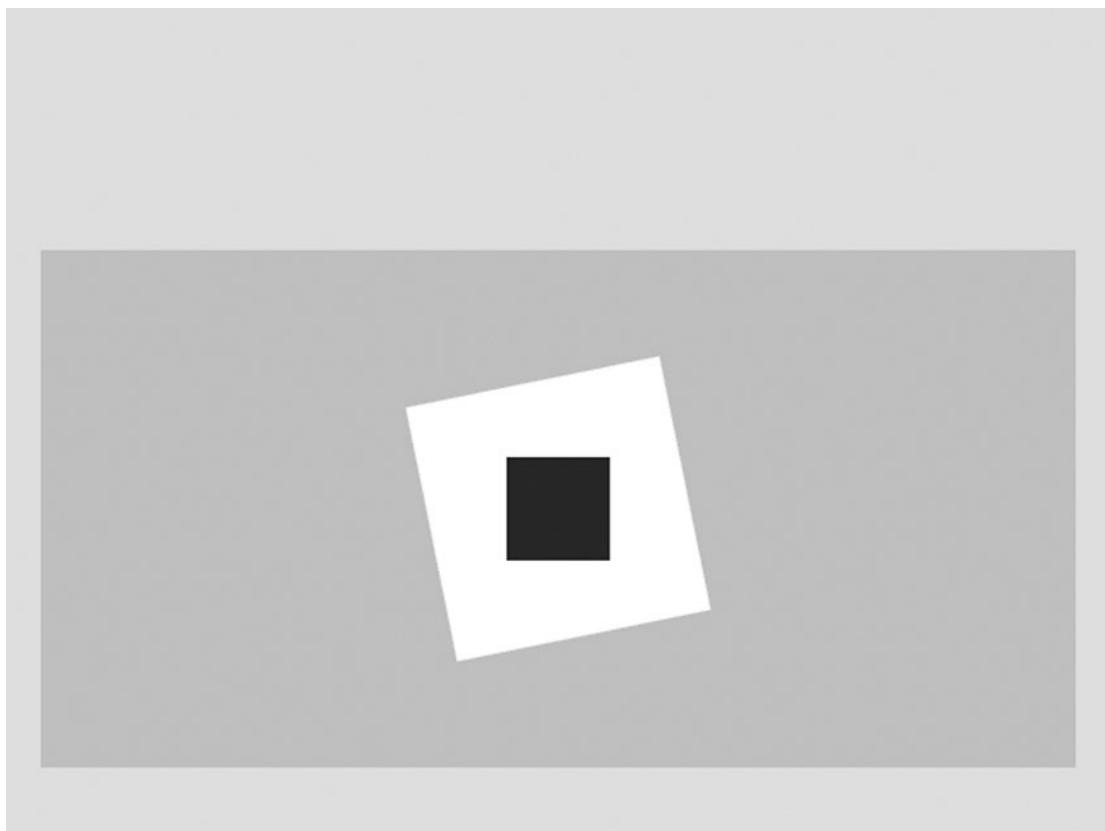


Figure 4-2. Running the Keyboard Support project

The controls of the project are as follows:

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

The goals of the project are as follows:

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

Add an Input Component to the Engine

The input component of the game follows the same pattern as the other core engine modules such as the vertex buffer or the game loop. A well-defined input module should allow the rest of the game engine to query keyboard state changes without being distracted by any details. To accurately capture keyboard state changes, the input component will be integrated with the core of game loop.

1. Create a new file in the `src/Engine/Core/` folder and name it `Engine_Input.js`. Similar to other engine components (for example, `gEngine_GameLoop` or `gEngine_VertexBuffer`), add the following:

```
var gEngine = gEngine || { };

gEngine.Input = (function() {
    var mPublic = { };
    return mPublic;
}());
```

2. Define a set of keyboard keys to map key codes.

```
// Key code constants
var kKeys = {
    // 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,
    F : 70,
    G : 71,
    I : 73,
    J : 74,
```

```

K : 75,
L : 76,
R : 82,
S : 83,
W : 87,
LastKeyCode : 222
};
```

Key codes are the codes used by the keyboard handler where each keyboard character has its own unique number, the corresponding key code. Note that there are up to 222 keys tracked. In the previous listing, only the constants are shown in the public interface.

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 <http://www.cambiaresearch.com/articles/15/javascript-char-codes-key-codes>.

3. Create objects to track key states.

```

// Previous key state
var mKeyPreviousState = [];

// The pressed keys.
var mIsKeyPressed = [];

// Click events: once an event is set, it will remain there until polled
var mIsKeyClicked = [];
```

Each of the three objects contains all the key states as booleans. The `mKeyPreviousState` records the key states of the previous update cycle, and the `mIsKeyPressed` object records the current state of the keys. The key code entries of these two objects are true when the corresponding keyboard keys are pressed, and they are false otherwise. The `mIsKeyClicked` object captures key click events. The key code entries of this object are true only when the corresponding keyboard key goes from being released to being pressed.

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 she releases it, for the duration of that entire second, `KeyPress` for A is true, while `KeyClick` for A is true only once when the player releases the key.

4. Add functions to capture the actual keyboard state changes.

```

// Event service functions
var _onKeyDown = function (event) {
    mIsKeyPressed[event.keyCode] = true;  };
var _onKeyUp = function (event)  {
    mIsKeyPressed[event.keyCode] = false; };
```

When the previous functions are called, they use their corresponding key code to record keyboard state changes.

5. Add a function to initialize all the key states and register the key event handlers to the browser.

```
var initialize = function () {
    var I;
    for (i = 0; i < kLastKeyCode; i++) {
        mIsKeyPressed[i] = false;
        mKeyPreviousState[i] = false;
        mIsKeyClicked[i] = false;
    }

    // register handlers
    window.addEventListener('keyup', _onKeyUp);
    window.addEventListener('keydown', _onKeyDown);
};
```

Notice that the `window.addEventListener()` function registers the `_onKeyUp/Down()` event handler functions with the browser such that these functions will be called to register the keyboard state changes.

6. Add an update() function to derive key click events.

```
var update = function() {
    var I;
    for (i = 0; i < kLastKeyCode; i++) {
        mIsKeyClicked[i] = (!mKeyPreviousState[i]) && mIsKeyPressed[i];
        mKeyPreviousState[i] = mIsKeyPressed[i];
    }
};
```

The `update()` function uses `mIsKeyPressed` and `mKeyPreviousState` to determine whether a key clicked event has occurred.

7. Add public functions for clean inquiries to current keyboard states.

```
// Function for GameEngine programmer to test if a key is pressed down
var isKeyPressed = function (keyCode) {
    return mIsKeyPressed[keyCode]; };
var isKeyClicked = function(keyCode) {
    return (mIsKeyClicked[keyCode]); };
};
```

8. Finally, add the public functions and the key constants to the interface.

```
var mPublic =
{
    initialize: initialize,
    update: update,
    isKeyPressed: isKeyPressed,
    isKeyClicked: isKeyClicked,
    keys: kKeys;
};
```

Modify the Engine to Support Keyboard Input

To properly support input, the engine must first initialize the arrays that represent the keyboard state, in other words, `mIsKeyPressed`, `mIsKeyClicked`, and `mKeyPreviousState`, and be followed by a continuous update of these arrays in the core of the game loop.

- For clean and clear initialization of the game engine core, WebGL-specific initializations are collected into a single function. In `Engine_Core.js`, modify the `_initializeWebGL()` function to the following:

```
var _initializeWebGL = function(htmlCanvasID) {
    var canvas = document.getElementById(htmlCanvasID);

    mGL = canvas.getContext("webgl") ||
        canvas.getContext("experimental-webgl");

    if (mGL === null) {
        document.write("<br><b>WebGL is not supported!</b>");
    }
};
```

This function only initializes WebGL.

- The engine core initialization can now be cleanly defined as the initialization of each essential component, as follows:

```
// initialize all of the EngineCore components
var initializeEngineCore = function(htmlCanvasID) {
    _initializeWebGL(htmlCanvasID);
    gEngine.VertexBuffer.initialize();
    gEngine.Input.initialize();
};
```

- Remember to include the correct functions in the public interface.

```
var mPublic = {
    getGL: getGL,
    initializeEngineCore: initializeEngineCore,
    clearCanvas: clearCanvas
};
```

- Finally, it is important to update the input states from within the game loop. In `Engine_GameLoop.js`, within the `_runLoop()` function, add a call to update the input before each update to the game in step C, as follows:

```
var _runLoop = function () {
    if(mIsLoopRunning) {
        // ... rest of the code is identical ...
        // Step C: Make sure we update the game the appropriate number of times.
```

```

        while ((mLagTime >= kMPF) && (mIsLoopRunning)) {
            gEngine.Input.update();
            this.update();           // call Scene.update()
            mLagTime -= kMPF;
        }
        // ... rest of the code is identical ...
    }
}

```

Test Keyboard Input

You can test the input functionality by modifying renderable objects in MyGame.

1. First you must modify MyGame to work with the new engine core initialization function. In MyGame.js, within the MyGame constructor, call the new initialization function as follows:

```

// Initialize the webGL Context
function MyGame(htmlCanvasID) {
    // ... identical code as previous project ...
    gEngine.Core.initializeEngineCore(htmlCanvasID);
    // ... identical code as previous project ...
}

```

2. Replace the code in the MyGame.prototype.update() function with the following:

```

MyGame.prototype.update = function() {
    // For this very simple game, let's move the white square and pulse the red
    var whiteXform = this.mWhiteSq.getXform();
    var deltaX = 0.05;

    // Step A: test for white square movement
    if (gEngine.Input.isKeyPressed(gEngine.Input.keys.Right)) {
        if (whiteXform.getXPos() > 30) // the right-bound of the window
            whiteXform.setPosition(10, 60);
        whiteXform.incXPosBy(deltaX);
    }

    // Step B: test for white square rotation
    if (gEngine.Input.isKeyClicked(gEngine.Input.keys.Up))
        whiteXform.incRotationByDegree(1);

    var redXform = this.mRedSq.getXform();
    // Step C: test for pulsing the red square
    if (gEngine.Input.isKeyPressed(gEngine.Input.keys.Down)) {
        if (redXform.getWidth() > 5)
            redXform.setSize(2, 2);
        redXform.incSizeBy(0.05);
    }
};

```

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 a key press event and thus will not cause the white square to 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 upwards, A leftwards, S downwards, and D rightwards.

Resource Management and Asynchronous Loading

Video games typically utilize a multitude of artistic assets, or resources, including audio clips and images. When a game first begins to execute, these resources are typically stored externally on a system hard drive or a server across the network. For this reason, these resources are sometimes referred to as *external resources*. External resources must be explicitly loaded into a game.

Since there can be a large number of required resources to support an entire game, storing them with the running game can potentially be memory intensive. A game should load and unload resources dynamically based on necessity. However, loading external resources may involve input/output device operations or network packet latencies and thus can be time intensive and potentially affect real-time interactivity. For these reasons, only a portion of resources are kept in memory, with loading operations strategically executed to avoid interrupting the game. In most cases, resources required in each level are kept in memory to support real-time interaction during the game play of that level. With this approach, external resource loading can be implemented during level transitions where players are expecting a new game environment and slight delays for loadings can be tolerated.

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

At this point, the game engine you have been building handles only one type of resource—the GLSL shader files. Recall that the `SimpleShader` object loads and compiles the `SimpleVS.gls1` and `SimpleFS.gls1` files in its constructor. So far, the shader file loading has been accomplished via synchronous `XMLHttpRequest.open()`. This synchronous loading is an example of inefficient resource management because no operations can occur while the browser attempts to open and load the shader file. An efficient alternative would be to issue an asynchronous load command and allow additional operations to continue while the file is being opened and loaded.

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

The Resource Map and Shader Loader Project

This project guides you to develop the `ResourceMap`, an infrastructural object for resource management, and demonstrates how to work with this object 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.ResourceMapAndShaderLoader](#) folder.

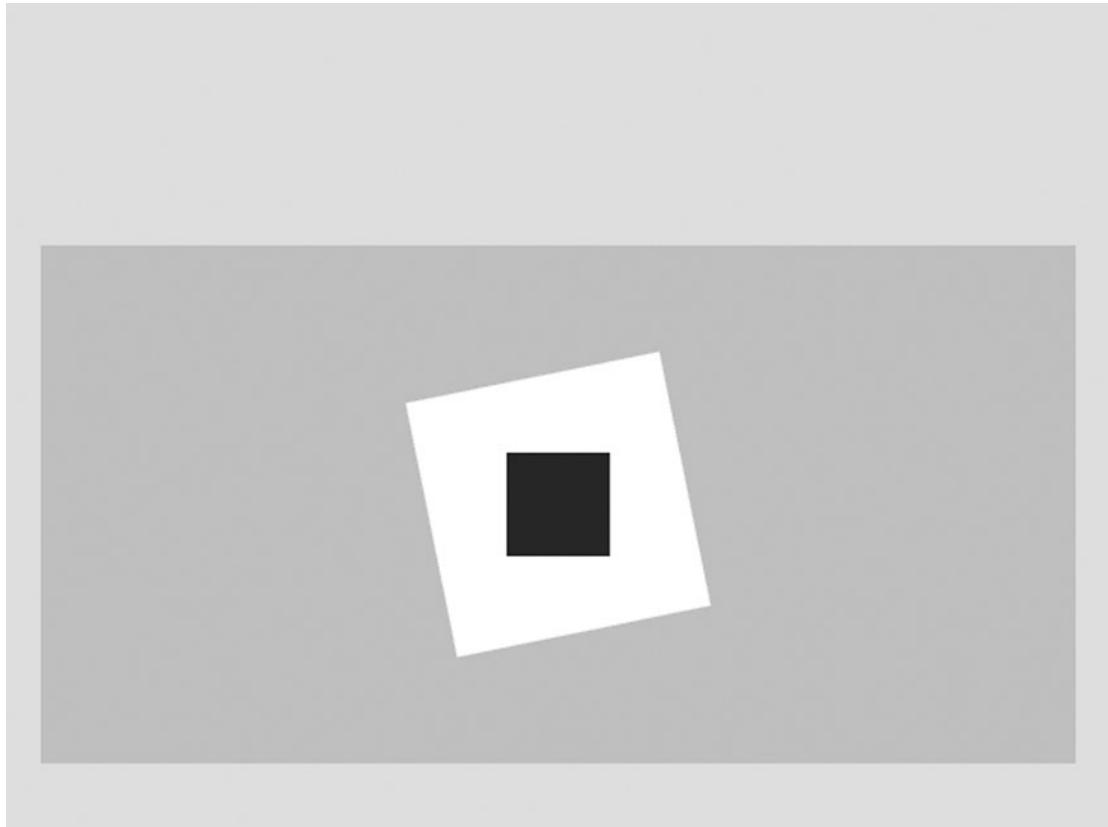


Figure 4-3. Running the Resource Map and Shader Loader project

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

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

The goals of the project are as follows:

- To understand the handling of asynchronous loading
- To build an infrastructure that supports future resource loading and accessing
- To experience asynchronous resource loading via loading of the GLSL shader files

Add ResourceMap Component to the Engine

The ResourceMap engine component manages resource loading, storage, and retrieval after the resources are loaded. As in the case of all core engine components (for example, input or game loop), the implementation pattern is as follows:

1. Create a new folder in `src/Engine/Core/` 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.
2. Create a new file in the `src/Engine/Core/Resources` folder and name it `Engine_ResourceMap.js`. Similar to other engine components, add the following:

```
var gEngine = gEngine || { };

gEngine.ResourceMap = (function(){
    var mPublic = { };
    return mPublic;
}());
```

3. Define a simple object for storing each resource.

```
var MapEntry = function(rName) {
    this.mAsset = rName;
};
```

In the constructor, the `mAsset` is initialized to the name of the resource. As will be shown in the following steps, this will be changed, and `mAsset` will store the reference to the loaded resource.

4. Add the following instance variables:

```
// Resource storage
var mResourceMap = {};

// Number of outstanding load operations
var mNumOutstandingLoads = 0;

// Callback function when all textures are loaded
var mLoadCompleteCallback = null;
```

The `mResourceMap` is the hashmap container of `MapEntry` that will support the actual storage and retrieval according to the unique names of resources. Names of resources are typically their corresponding file paths, which are unique. The `mNumOutstandingLoads` is the number of resource load requests issued and not completed. When `mNumOutstandingLoads` becomes zero, the function referenced by `mLoadCompleteCallback` will be called.

5. Define the support for setting and executing the callback.

```

var _checkForAllLoadCompleted = function() {
    if ((mNumOutstandingLoads === 0) && (mLoadCompleteCallback !== null) ) {
        // ensures the load complete call back will only be called once!
        var funToCall = mLoadCompleteCallback;
        mLoadCompleteCallback = null;
        funToCall();
    }
};

// Make sure to set the callback AFTER all load commands are issued
var setLoadCompleteCallback = function (funct) {
    mLoadCompleteCallback = funct;
    // in case all loading are done
    _checkForAllLoadCompleted();
};

```

When all resources are loaded or when `mNumOutstandingLoads` is zero and if the callback function `mLoadCompleteCallback` is defined, it is called. Note that in `_checkForAllLoadCompleted()`, the `mLoadCompleteCallback` is reset to ensure each callback can be called only once.

6. Define functions to record asynchronous loading requests and completions.

```

var asyncLoadRequested = function(rName) {
    mResourceMap[rName] = new MapEntry(rName);
    // place holder for the resource to be loaded
    ++mNumOutstandingLoads;
};

var asyncLoadCompleted = function(rName, loadedAsset) {
    if (!isAssetLoaded(rName))
        alert("gEngine.asyncLoadCompleted: [" + rName + "] not in map!");
    mResourceMap[rName].mAsset = loadedAsset;
    --mNumOutstandingLoads;
    _checkForAllLoadCompleted();
};

```

The `asyncLoadRequested()` and `asyncLoadCompleted()` functions keep track of the number of outstanding asynchronous loads issued. Notice that `asyncLoadRequested()` allocates a new `MapEntry` and stores it in the `_ResourceMap` according to the resource name. Whereas the `asyncLoadCompleted()` function stored the reference to the loaded asset in `MapEntry.mAsset`, `_checkForAllLoadCompleted()` is always called at the end of the `asyncLoadCompleted()` function to test for the condition where all outstanding asynchronous loadings have been completed, and the callback should be executed.

7. Add functions for testing the load status and retrieving and unloading resources.

```

var isAssetLoaded = function(rName) {
    return (rName in mResourceMap);
};

var retrieveAsset = function(rName) {
    var r = null;
    if (rName in mResourceMap)
        r = mResourceMap[rName].mAsset;
    return r;
};

```

```
var unloadAsset = function(rName) {
    if (rName in mResourceMap) {
        delete mResourceMap[rName];
    }
};
```

8. Lastly, remember to add all public functions to the public interface.

```
// Public interface for this object.
var mPublic = {
    // asynchronous resource loading support
    asyncLoadRequested: asyncLoadRequested,
    asyncLoadCompleted: asyncLoadCompleted,
    setLoadCompleteCallback: setLoadCompleteCallback,

    // resource storage
    retrieveAsset: retrieveAsset,
    unloadAsset: unloadAsset,
    isAssetLoaded: isAssetLoaded
};
return mPublic;
```

Notice that ResourceMap is not designed to perform the actual loading of any resources. Instead, it is a manager that allows an asynchronous resource loader to register loading commands issued and to store and retrieve loaded results.

Define a Text File Loader as an Engine Component

This section will define a TextFileLoader object to work with ResourceMap to load text files asynchronously. This object serves as an excellent example of how to take advantage of the ResourceMap facility and how to replace the synchronous loading of GLSL shader files.

1. Create a new file in the `src/Engine/Core/Resources` folder and name it `Engine_TextFileLoader.js`.
2. Define the TextFileLoader similar to other engine components.

```
var gEngine = gEngine || {};
gEngine.TextFileLoader = (function() {
    var mPublic = {};
    return mPublic;
}());
```

3. Define the different types of text files to be loaded, such as XML or plain text (for example, GLSL shaders).

```
var eTextFileType = Object.freeze({
    eXMLFile: 0,
    eTextFile: 1
});
```

Note `Object.freeze()` disables modification and makes an object immutable.

4. Add a function to asynchronously load a given text file.

```

var loadTextFile= function(fileName, fileType, callbackFunction) {
    if (!(gEngine.ResourceMap.isAssetLoaded(fileName))) {
        // Update resources in load counter.
        gEngine.ResourceMap.asyncLoadRequested(fileName);

        // Asyncrounly request the data from server.
        var req = new XMLHttpRequest();
        req.onreadystatechange = function () {
            if ((req.readyState === 4) && (req.status !== 200)) {
                alert(fileName + ": loading failed!
                    [Hint: you cannot double click index.html to run this
                    project. " +
                    "The index.html file must be loaded by a web-server.]");
            }
        };
        req.open('GET', fileName, true);
        req.setRequestHeader('Content-Type', 'text/xml');

        req.onload = function () {
            var fileContent = null;
            if (fileType === eTextFileType.eXMLFile) {
                var parser = new DOMParser();
                fileContent = parser.parseFromString(req.responseText, "text/xml");
            } else {
                fileContent = req.responseText;
            }
            gEngine.ResourceMap.asyncLoadCompleted(fileName, fileContent);

            if ((callbackFunction !== null) &&(callbackFunction !== undefined))
                callbackFunction(fileName);
        };
        req.send();
    } else {
        if ((callbackFunction !== null) && (callbackFunction !== undefined))
            callbackFunction(fileName);
    }
};

```

With the listing, notice the following:

- a. The `fileName` parameter is the path to the text file, and it will be used as the unique resource name.
- b. The `fileType` parameter is of `eTextFileType` indicating whether the load request is meant for an XML file or a simple text file.

- c. The `ResourceMap.isAssetLoaded()` function is first called to ensure that the requested resource is not already loaded before an actual loading command is issued. This ensures that the same resource will be loaded only once and will be shared.
 - d. The `ResourceMap.asyncLoadRequested(fileName)` function is called to register a new asynchronous XMLHttpRequest loading request. Take note of how these two classes collaborate; `ResourceMap` registers that there is one outstanding load operation, while `TextFileLoader` is performing the actual loading operation.
 - e. The `ResourceMap.asyncLoadCompleted()` function is called and passed the loaded results when it becomes available. In this way, the `ResourceMap` entry identified by `fileName` will have its `MapEntry.mAsset` set to reference `fileContent`, the loaded asset.
 - f. Finally, when loading is done and if `callbackFunction` is defined, it will be called.
5. Complete the loader object by defining an appropriate unload function.

```
var unloadTextFile= function(fileName) {
    gEngine.ResourceMap.unloadAsset(fileName);
};
```

6. Lastly, remember to add public functions to the public interface.

```
// Public interface for this object.
var mPublic = {
    loadTextFile: loadTextFile,
    unloadTextFile: unloadTextFile,
    eTextFileType: eTextFileType
};
return mPublic;
```

Load Shaders Asynchronously

The `TextFileLoader` object can now be used to load the shader files asynchronously as plain-text files. Since it is impossible to predict when an asynchronous loading operation will be completed, it is important to issue the load commands before the resources are needed and to ensure that the loading operations are completed before proceeding to retrieve the resources. The callback function in the `ResourceMap` serves as the excellent indicator that loadings are completed and resources can be used.

Implement Default Resources Support

There are two interesting observations on the `SimpleShader` object, and together these observations lead to the definition of an engine-wide default resource storage support, or the `DefaultResources` component. First, to avoid loading the GLSL shader files synchronously, the files must be loaded before the creation of a `SimpleShader` object. Second, the `SimpleShader` object serves as a conduit for passing information and can be shared by `Renderable` objects for supporting the drawing of different constant color squares. Based on these observations, a `DefaultResources` engine component can be created to load GLSL

shader files during engine initialization and to create a sharable instance of `SimpleShader`. In general, the `DefaultResources` component can serve as the infrastructure that supports all future loading and sharing of game engine resources.

1. Create a new file in the `src/Engine/Core/Resources` folder and name it `Engine_DefaultResources.js`.
2. Define the `DefaultResources` similar to all other engine components.

```
var gEngine = gEngine || { };

gEngine.DefaultResources = (function() {
    var mPublic = {};
    return mPublic;
}());
```

3. Define the constant file paths to the GLSL shaders and the variable and assessor to the `SimpleShader` object.

```
// Simple Shader GLSL Shader file paths
var kSimpleVS = "src/GLSLShaders/SimpleVS.glsl"; // Path to the VertexShader
var kSimpleFS = "src/GLSLShaders/SimpleFS.glsl"; // Path to the simple
// FragmentShader

var mConstColorShader = null; // variable for the SimpleShader object
var _getConstColorShader = function() { return mConstColorShader; }; // assessor
```

4. Define the initialization function to initiate the GLSL shader loadings and set the target callback function to create the `SimpleShader` when all loading has completed.

```
// callback function after loadings are done
var _createShaders = function(callBackFunction) {
    mConstColorShader = new SimpleShader(kSimpleVS, kSimpleFS);
    callBackFunction();
};

// initiate asynchronous loading of GLSL Shader files
var _initialize = function(callBackFunction) {
    // constant color shader: SimpleVS, and SimpleFS
    gEngine.TextFileLoader.loadTextFile(kSimpleVS,
        gEngine.TextFileLoader.eTextFileType.eTextFile);
    gEngine.TextFileLoader.loadTextFile(kSimpleFS,
        gEngine.TextFileLoader.eTextFileType.eTextFile);

    gEngine.ResourceMap.setLoadCompleteCallback(
        function() {_createShaders(callBackFunction);});
};
```

The `_initialize()` function issues the two loads for the GLSL vertex and fragment shader files and sets the `_createShaders()` to be the callback when the load operations are completed. The `_createShaders()` function in turn calls the `SimpleShader` constructor with the shader file paths to create the constant color shader. In the following, you will see how these file paths are used as resource names for retrieving the contents of the shader files.

- Once again, remember to add all public functions to the public interface.

```
var mPublic = {
    initialize: _initialize,
    getConstColorShader: _getConstColorShader
};
return mPublic;
```

With `DefaultResources`, the game engine can now load and create constant color shaders during internal initialization and offers the `mConstColorShader` as a shared `SimpleShader` for all instances of `Renderables`. In the following, the required modifications to the `SimpleShader` implementation and the overall integration of all components are examined.

Modify SimpleShader to Retrieve 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 `ResourceMap`.

- Since no loading operations are required, you should change the `_loadAndCompileShader()` function name to simply `_compileShader()` and change the file-loading commands to content retrievals.

```
SimpleShader.prototype._compileShader = function(filePath, shaderType) {
    var gl = gEngine.Core.getGL();
    var shaderSource = null, compiledShader = null;

    // Step A: Access the shader textfile
    shaderSource = gEngine.ResourceMap.retrieveAsset(filePath);
    // ... identical to previous code ...
```

Notice that the synchronous loading operations are replaced by a single call to `ResourceMap.retrieveAsset()` to retrieve the file content based on the `filePath` or the unique resource name for the shader file.

- Remember that in the `SimpleShader` constructor, the calls to `_loadAndCompileShader()` functions should be replaced by `_compileShader()` functions, as follows:

```
function SimpleShader(vertexShaderPath, fragmentShaderPath) {
    // ... identical to previous code ...

    // Step A: load and compile vertex and fragment shaders
    var vertexShader = this._compileShader(vertexShaderPath, gl.VERTEX_SHADER);
    var fragmentShader = this._compileShader(fragmentShaderPath, gl.FRAGMENT_SHADER);
```

Recall that the instance of `SimpleObject` is created from the `DefaultResources._createShaders()` function where the path to the vertex and fragment shaders are passed in as parameters to the constructor. In the previous code, the file paths are used as unique resource names to retrieve the corresponding GLSL shader file contents from the `ResourceMap`.

Initialize Engine Core

The DefaultResources component must be initialized when the engine core is first initialized. In addition, an appropriate callback function must be provided for the DefaultResources._createShaders() function to call after the SimpleShader object is created.

1. Initialize DefaultResources in Engine_Core.js.

```
var initializeEngineCore = function(htmlCanvasID, myGame) {
    _initializeWebGL(htmlCanvasID);
    gEngine.VertexBuffer.initialize();
    gEngine.Input.initialize();

    // Inits DefaultResources, when done, invoke startScene(myGame).
    gEngine.DefaultResources.initialize(function() { startScene(myGame); } );
};
```

2. Define the startScene() function to initialize myGame and start the game loop.

```
var startScene = function(myGame) {
    myGame.initialize.call(myGame); // Called in this way to keep correct context
    gEngine.GameLoop.start(myGame); // start the game loop after initialization
};
```

With these modifications, Engine.Core initialization will call the DefaultResources initialization, causing the asynchronous loading of the GLSL shader files. DefaultResources in turn will call the startScene() function after the SimpleShader object is created. In this way, MyGame initialization will be properly called, and the system game loop will be properly started.

Modify index.html for Proper Engine Core Initialization

The last step in integrating all the changes is to ensure that the initializeEngineCore() function is properly activated, before the initialization of MyGame. To accomplish this, the index.html file's onload event handler must be modified to create an instance of MyGame and to call the initializeEngineCore() function with the instance of MyGame as a parameter.

```
<body onload="
    var myGame = new MyGame();
    gEngine.Core.initializeEngineCore('GLCanvas', myGame);
">
```

Logically, the modified onload event handler initializes the engine core by passing the area upon which the game should output to, GLCanvas, and a game object upon which the game should begin, myGame. In the following sections, the MyGame parameter to initializeEngineCore() will be generalized into any arbitrary game levels.

Test the Asynchronous Shader Loading

With `DefaultResources` handling the storage of the shaders and `index.html` initializing the engine core, `MyGame` object does not need to be concerned with any special initialization calls to engine core, and it can access `DefaultResources` for the sharable instance of `SimpleShader`.

1. Remove the call to `gEngine.Core.initializeEngineCore()` from the constructor.

```
function MyGame() {
    // variables for the squares
    this.mWhiteSq = null;           // these are the renderable objects
    this.mRedSq = null;

    // The camera to view the scene
    this.mCamera = null;
}
```

2. Remove the creation of `SimpleShaders` and reference to `DefaultResources` for the sharable `SimpleShader` to create `Renderable` objects as follows:

```
// Step C: Create the renderable objects:
var constColorShader = gEngine.DefaultResources.getConstColorShader();
this.mWhiteSq = new Renderable(constColorShader);
this.mWhiteSq.setColor([1, 1, 1, 1]);
this.mRedSq = new Renderable(constColorShader);
this.mRedSq.setColor([1, 0, 0, 1]);
```

You can now run the project with shaders being loaded asynchronously. Though the output and interaction experience are identical to the previous project, you now have a game engine that is much better equipped to manage the loading and accessing of external resources efficiently.

The rest of this chapter further develops and formalizes the interface between the client, in other words, `MyGame`, and the rest of the game engine. The goal is to define the interface to the client such that multiple instances can be created and interchanged during runtime. With this new interface, you will be able to define what a game level is and allow the game engine to load any level in any order.

Game Level from a Scene File

The scene file is a formal interface between the game engine and its client because it triggers a sequence of function calls to create a playable game level. With a game level defined in a scene file, the game engine must first initiate asynchronous loading, wait for the load completion, and then initialize the client for the game loop. These steps present a complete functional interface between the game engine and the client. By examining and deriving the proper support for these steps, the interface between the game engine and its client can be refined.

The Scene File Project

This project uses the loading of a scene file as the vehicle to examine the necessary public methods for a typical game level. You can see an example of this project running in Figure 4-4. This project appears and interacts identically to the previous project with the only difference being that the scene definition is asynchronously loaded from a file. The source code to this project is defined in the Chapter4/4.4.SceneFile folder.

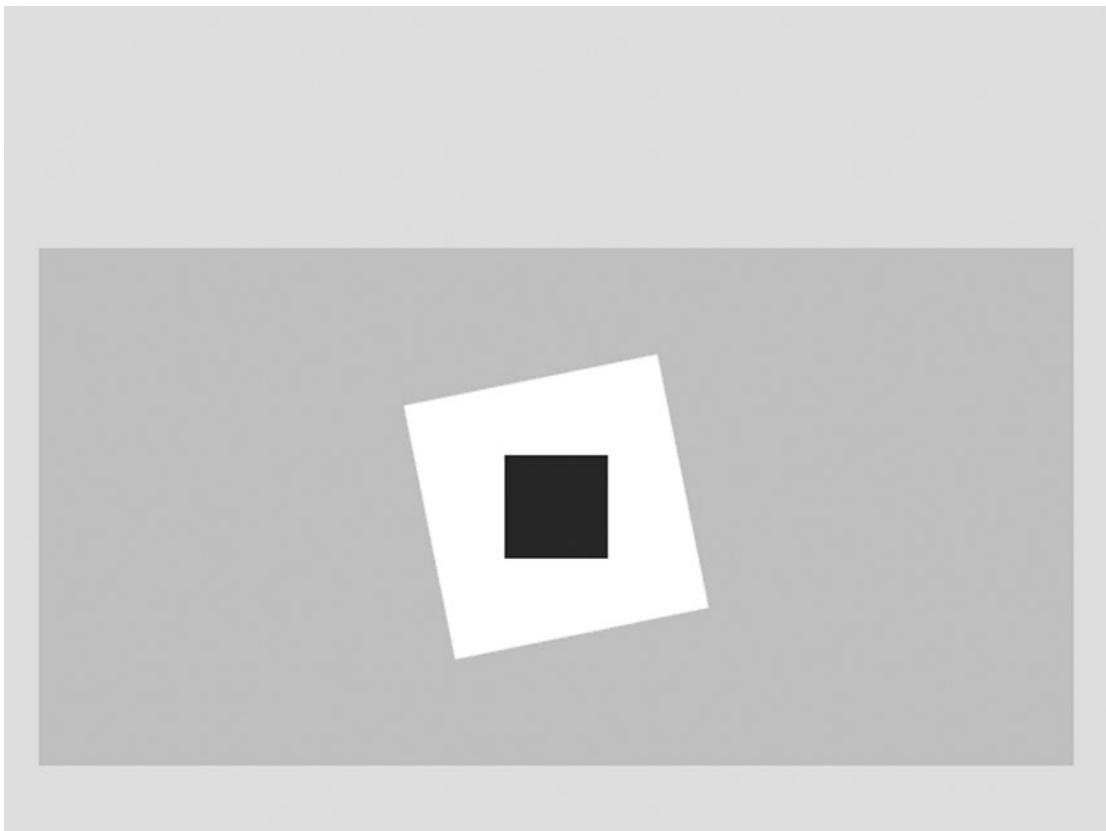


Figure 4-4. Running the Scene File project

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

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

The goals of the project are as follows:

- To introduce the protocol for supporting asynchronous loading of the resources of a game
- To develop the proper game engine support for the protocol
- To identify and define the public interface methods for a general game level

Keep in mind the ultimate goal of this project is to define the public interface methods between the game engine and a game level, or the client. While the definition/loading of a scene file is interesting, in this case it is but a vehicle. The following describes the definition and parsing utility for the scene file. It is important to remember these are only the tools for examining the required public methods for interfacing to the game engine.

The Scene File

Instead of hard-coding the creation of all objects to a game in the `initialize()` function, the information can be encoded in a file, and the file can be loaded and parsed during runtime. The advantage of such encoding in an external file is the flexibility to modify a scene without the need to change the game source code, while the disadvantages are the complexity and time required for loading and parsing. In general, the importance of flexibility dictates that all game engines support the loading of game scenes from a file.

Objects in a game scene can be defined in many ways. The key decision factors are that the format can properly describe the game objects and be easily parsed. Extensible Markup Language (XML) is well-suited to serve as the encoding scheme for scene files.

1. Create a new folder at the same level as the `src` folder and name it `assets`. This is the folder where all external resources, or assets, of a game will be stored including the scene files, audio clips, texture images, and fonts.

Tip It is important to differentiate between the `src/Engine/Core/Resources/` folder that is created for organizing game engine source code files and the `assets/` folder that you just created for storing client resources. Although GLSL shaders are also loaded at runtime, they are considered as source code and will continue to be stored in the `src/GLSLShaders` folder.

2. Create a new file in the `assets` folder and name it `scene.xml`. This file will store the client's game scene. Add the following content:

```
<MyGameLevel>
    <Camera CenterX="20" CenterY="60" Width="20"
        Viewport="20 40 600 300"
        BgColor="0.8 0.8 0.8 1.0"
    />
    <!-- Squares Rotation is in degree -->
    <Square PosX="20" PosY="60" Width="5" Height="5" Rotation="30" Color="1 1 1 1" />
    <Square PosX="20" PosY="60" Width="2" Height="2" Rotation="0" Color="1 0 0 1" />
</MyGameLevel>
```

The listed XML content describes the same scene as defined in the `initialize()` functions from previous `MyGame` objects.

Tip Delimiting attributes with commas is not supported.

Parser for 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 MyGame folder.

1. Create a new folder in the `src/MyGame` folder and name it `Util`. Add a new file in the `Util` folder and name it `SceneFileParser.js`. This file will contain the specific parsing logic to decode the listed scene file.
2. Define the constructor for the `SceneFileParser` object.

```
function SceneFileParser(sceneFilePath) {
    this.mSceneXml = gEngine.ResourceMap.retrieveAsset(sceneFilePath);
}
```

Note that `sceneFilePath` is the complete path to the scene file. In this case, the file path is once again used as a unique resource name for retrieving the loaded asset from the `ResourceMap`.

Note The following XML parsing is based on JavaScript XML API. Please refer to <http://www.w3schools.com/dom/> for more details.

3. Add a function to parse for contents of an XML element.

```
SceneFileParser.prototype._getElm = function(tagElm) {
    var theElm = this.mSceneXml.getElementsByTagName(tagElm);
    if (theElm.length === 0)
        console.error("Warning: Level element:[" + tagElm + "]: is not found!");
    return theElm;
};
```

4. Add a function to retrieve the definition of a camera.

```
SceneFileParser.prototype.parseCamera = function() {
    var camElm = this._getElm("Camera");
    var cx = Number(camElm[0].getAttribute("CenterX"));
    var cy = Number(camElm[0].getAttribute("CenterY"));
    var w = Number(camElm[0].getAttribute("Width"));
    var viewport = camElm[0].getAttribute("Viewport").split(" ");
    var bgColor = camElm[0].getAttribute("BgColor").split(" ");
    // make sure viewport and color are number
    for (var j = 0; j<4; j++) {
        bgColor[j] = Number(bgColor[j]);
        viewport[j] = Number(viewport[j]);
    }
    var cam = new Camera(
        vec2.fromValues(cx, cy), // position of the camera
        w,                      // width of camera
        viewport                // viewport (orgX, orgY, width, height)
    );
```

```

        cam.setBackgroundColor(bgColor);
        return cam;
    };
}

```

The camera parser finds a camera element and constructs a camera object with the retrieved information. Notice that the viewport and background colors are arrays of four numbers. These are input as string of four numbers delimited by spaces. Strings can be split into arrays, which is the case here with the space delimiter. The JavaScript `Number()` function ensures all strings are converted into numbers.

5. Add a function to parse for squares.

```

SceneFileParser.prototype.parseSquares = function(sqSet) {
    var elm = this._getElm("Square");
    var 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 Renderable(gEngine.DefaultResources.getConstColorShader());
        // make sure color array contains numbers
        for (j = 0; j<3; j++)
            c[j] = Number(c[j]);
        sq.setColor(c);
        sq.getXform().setPosition(x, y);
        sq.getXform().setRotationInDegree(r); // In Radian
        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.

Integrate Game Resource Loading

Though slightly involved, the details of XML-parsing specifics are less important than the fact that now XML files can be parsed. It is now possible to use the asynchronous loading of an external resource to study the required public methods for interfacing a game level to the game engine.

Public Methods of MyGame

At this point, it is established that `MyGame` should define the following:

- *Constructor*: For declaring variables and defining constants
- `initialize()`: For instantiating the variables and setting up the game scene
- `update() / draw()`: For interfacing to the game loop with these two functions being called continuously

With the requirement of loading a scene file, two additional public methods will be defined.

- `loadScene()`: For initiating the asynchronous loading of external resources, in this case, the scene file
- `unloadScene()`: For unloading of external resources when the game has ended

The implementations of these functions are as follows:

1. MyGame constructor:

```
function MyGame() {
    // scene file name
    this.kSceneFile = "assets/scene.xml"
    // all squares
    this.mSqSet = new Array();           // these are the renderable objects

    // The camera to view the scene
    this.mCamera = null;
};
```

The constructor defines the scene file path, the array `mSqSet` for storing the Renderable objects, and the camera.

2. Define the functions to load and unload the scene file.

```
MyGame.prototype.loadScene = function() {
    gEngine.TextFileLoader.loadTextFile(this.kSceneFile,
        gEngine.TextFileLoader.eTextFileType.eXMLFile);
};

MyGame.prototype.unloadScene = function() {
    gEngine.TextFileLoader.unloadTextFile(this.kSceneFile);
};
```

The `loadScene()` function initiates the asynchronous XML text file loading, while the `unloadScene()` function frees the resources.

3. Change the `initialize()` function to create objects based on the scene parser, as follows:

```
MyGame.prototype.initialize = function() {
    var sceneParser = new SceneFileParser(this.kSceneFile);

    // Step A: Parse the camera
    this.mCamera = sceneParser.parseCamera();

    // Step B: Parse for all the squares
    sceneParser.parseSquares(this.mSqSet);
};
```

Once again, notice that the file path to the scene file is passed into the constructor of `SceneFileParser` and will be used as the resource name for retrieving the scene file contents from the `ResourceMap`.

4. The draw and update functions are similar to the previous examples with the exception of referencing the corresponding array elements.

```
MyGame.prototype.draw = function() {
    // ... identical code to previous project ...
    // Step C: draw all the squares
    for (var i = 0; i<this.mSqSet.length; i++) {
        this.mSqSet[i].draw(this.mCamera.getVPMatrix());
    }
};

MyGame.prototype.update = function() {
    // work with the xform of the white square ...
    var xform = this.mSqSet[0].getXform();
    // ... identical code to previous project ...

    // work with the red square ...
    xform = this.mSqSet[1].getXform();
    // ... identical code to previous project ...
};
```

Integration with the Game Engine

With the load and unload functionality defined in MyGame, the important task for the game engine is to ensure that MyGame initialization is called only after the load operation has completed. The game engine can ensure this sequence by coordinating the operations of engine core initialization and starting the game loop.

1. In Engine_Core.js, at the end of `initializeEngineCore()` after `DefaultResources` initialization is completed, the callback `startScene()` function needs to be modified to initiate the loading of the external resources of MyGame with the call to the `myGame.loadScene()` function.

```
var startScene = function(myGame) {
    myGame.loadScene.call(myGame); // Called in this way to keep correct context
    gEngine.GameLoop.start(myGame); // call initialize() only after async
    loading is done
};
```

2. In Engine_GameLoop.js, always wait for any potential MyGame asset loading to be completed before initializing the game and starting the loop.

```
var _startLoop = function() {
    // Step A: reset frame time
    mPreviousTime = Date.now();
    mLagTime = 0.0;
    // Step B: remember that loop is now running
    mIsLoopRunning = true;
    // Step C: request _runLoop to start when loading is done
    requestAnimationFrame(function(){_runLoop.call(mMyGame);});
};
```

```

var start = function(myGame) {
    mMyGame = myGame;
    gEngine.ResourceMap.setLoadCompleteCallback(
        function() {
            mMyGame.initialize();
            _startLoop();
        });
};

```

In the previous listing, the `_startLoop()` function implements the continuous cycling of the game loop. The `start()` function registers a call with the `ResourceMap` to wait for the completion of `MyGame`-loading operations before calling the game `initialize()` function, and then it begins the game loop. In this way, the game loop will begin only after all asynchronous loading operations are completed and the game is properly initialized.

You can now run the project and experience the identical behaviors with the previous two projects. Though less than overwhelming, it is important to remember the purpose of this project and recognize that through the process of supporting asynchronous loading of external resources, the public methods and calling sequence between the game engine and the client have been defined.

Before continuing, you may notice that the `MyGame.unloadScene()` function is never called. This is because in this example the game loop never stopped cycling and `MyGame` is never unloaded. This issue will be addressed in the next two projects.

Scene Object: Client Interface to the Game Engine

At this point, in your game engine, the following is happening:

- The `EngineCore` component defines the `startScene()` function that will initiate the loading process of `MyGame` and call to start the game loop.
- The `GameLoop.start()` function registers with the `ResourceMap` to wait for the completion of all asynchronous loading operations before it calls to initialize `MyGame` and starts the actual game loop cycle.

In the previous discussion, it is interesting to recognize that any object with the appropriately defined public methods can replace the `MyGame` object. Effectively, at any point, it is possible to call the `startScene()` function to load a new scene. This section pursues this observation by introducing the `Scene` object for interfacing the game engine with its clients.

The Scene Objects Project

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 well-defined and uniform 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.SceneObjects folder.

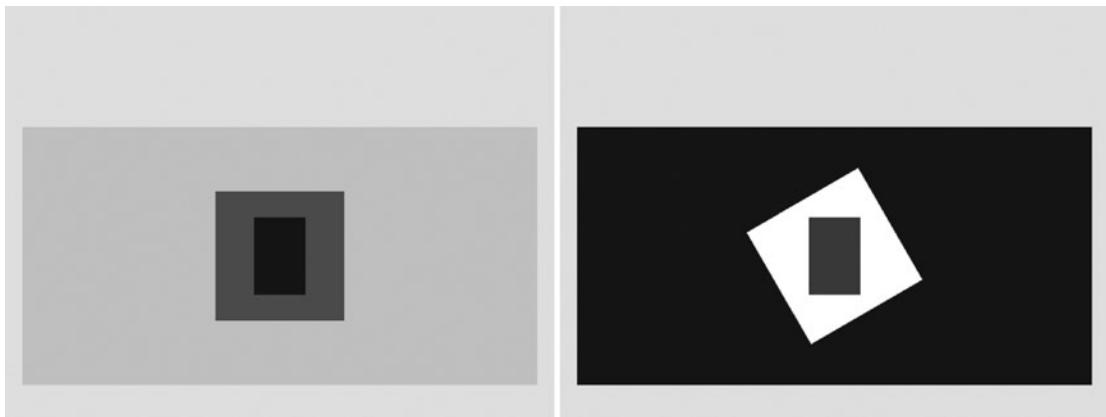


Figure 4-5. Running the Scene Objects project with both scenes

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.

- *Left/right arrow key:* Move the front rectangle left and right

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.

The goals of the project are as follows:

- To define the abstract `Scene` object to interface to the game engine
- To experience game engine support for scene transitions
- To create scene-specific loading and unloading support

The Abstract Scene Object

Based on the experience from the previous project, an abstract `Scene` object for encapsulating the interface to the game engine must define these functions: constructor, `initialize()`, `loadScene()`, `unloadScene()`, `update()`, and `draw()`.

1. Create a new JavaScript file in the `src/Engine` folder and name it `Scene.js`.
2. Implement the following functions:

```
function Scene() { // constructor }
Scene.prototype.initialize = function() {
    // Called from GameLoop, after loading is done
};
Scene.prototype.loadScene = function() {
    // called from EngineCore.startScene()
};
Scene.prototype.unloadScene = function() { };
Scene.prototype.update = function() { };
Scene.prototype.draw = function() { };
```

Notice that the Scene object is an abstract object because all the functions are empty. Together these functions present a protocol to interface with the game engine. It is expected that subclasses will override these functions to implement the actual game behaviors.

Note JavaScript does not support abstract objects. The language does not prevent a game programmer from instantiating a Scene object. However, the created instance will be completely useless.

Modify Game Engine to Support the Scene Object

The game engine core components can now be modified to support the explicitly defined Scene object.

1. To facilitate inheritance of the Scene prototype methods, the `inheritPrototype()` function is defined in the `Engine_Core.js` as follows:

```
var inheritPrototype = function(subClass, superClass) {
    var prototype = Object.create(superClass.prototype);
    prototype.constructor = subClass;
    subClass.prototype = prototype;
};
```

The `inheritPrototype()` function is a utility function that passes a reference to the prototype of one object to another. This is called *prototypal inheritance* and allows the inherited object to access the prototype functions of the original object. Note that the subclass's constructor is explicitly saved to prevent it from being overwritten.

2. In the `Engine_GameLoop.js` file, include support to stop the game loop and remember to add this function to public interface:

```
var stop = function() {
    mIsLoopRunning = false;
};
var mPublic = {
    start: start,
    stop: stop
};
```

3. To support orderly unloading of the currently running scene, the game loop must be stopped first, and then the `unload` function of the currently running scene can be called. With the previous `stop()` function, the unloading can be implemented by modifying the `_runLoop()` function in `Engine_GameLoop.js` to call the `unloadScene()` function when the game loop has stopped.

```
var _runLoop = function () {
    if(mIsLoopRunning) {
        // ... identical to previous code ...
    } else {
        // the game loops has stopped, unload current scene!
        mMyGame.unloadScene();
    }
};
```

Test the Scene Object Interface to the Game Engine

With the abstract Scene object definition and the simple modification to the game engine core components, it is now possible to stop an existing scene and load a new scene at will. This section uses the cycling between two subclasses of the Scene object, BlueLevel and MyGame, to illustrate the loading and unloading of scenes.

The BlueLevel Scene

Define a BlueLevel object that inherits from the Scene object and loads the scene from an external XML scene file.

1. Create a new XML scene file in the assets folder and name it `BlueLevel.xml`.

Add a scene definition for the BlueLevel as follows:

```
<MyGameLevel>
    <Camera CenterX="20" CenterY="60" Width="20"
        Viewport="20 40 600 300"
        BgColor="0 0 1 1.0"/>
    <Square PosX="20" PosY="60" Width="5" Height="5" Rotation="30" Color="1 1 1 1" />
    <Square PosX="20" PosY="60" Width="2" Height="3" Rotation="0"
        Color="1 0 0 1" />
</MyGameLevel>
```

Besides minor size, position, or color differences, this file defines a scene that is similar to the one defined by the `scene.xml` file from the previous project.

2. Create a new file in the `src/MyGame` folder and name it `BlueLevel.js`.

3. Define a constructor for BlueLevel as follows:

```
function BlueLevel() {
    // scene file name
    this.kSceneFile = "assets/BlueLevel.xml";
    // all squares
    this.mSqSet = []; // these are the renderable objects
    // The camera to view the scene
    this.mCamera = null;
}
gEngine.Core.inheritPrototype(BlueLevel, Scene);
```

Note the `inheritPrototype()` function is called right after the constructor. This overrides the prototype of the BlueLevel with that from the Scene and effectively defines all the behaviors of Scene on BlueLevel. It is important to call the `inheritPrototype()` function right after the constructor before any new prototype methods are defined for BlueLevel.

4. Override the Scene functions as follows:

```
BlueLevel.prototype.loadScene = function() {
    gEngine.TextFileLoader.loadTextFile(this.kSceneFile,
        gEngine.TextFileLoader.eTextFileType.eXMLFile);
};
```

```

BlueLevel.prototype.initialize = function() {
    // ... identical to MyGame.initialize from previous project ...
    // ... parse the scene from the this.kSceneFile ...
};

BlueLevel.prototype.draw = function() {
    // ... identical to MyGame.draw from previous project ...
    // ... set up the camera and draw all elements in the mSqSet array ...
};

BlueLevel.prototype.update = function() {
    // ... identical to MyGame.update from previous project ...

    if (gEngine.Input.isKeyPressed(gEngine.Input.keys.Left)) {
        xform.IncXPosBy(-deltaX);
        if (xform.getXPos() < 11) { // this is the left-boundary
            gEngine.GameLoop.stop();
        }
    }
};

BlueLevel.prototype.unloadScene = function() {
    // unload the scene file
    gEngine.TextFileLoader.unloadTextFile(this.kSceneFile);

    var nextLevel = new MyGame(); // the next level
    gEngine.Core.startScene(nextLevel);
};

```

Many of the BlueLevel functions are similar to the corresponding functions of the MyGame object from the previous project, and thus much of the details are not shown. However, notice the following:

- The `loadScene()` function initiates the asynchronous loading of the scene file and returns. It is important that the game engine wait for the completion of the load process before calling the `initialize()` function. Recall that in your game engine, the `GameLoop.start()` function implements this.
- The `update()` function is responsible for initiating the level transition by calling the `GameLoop.stop()` function when the transition condition becomes favorable, in this case when the rectangle approaches and touches the left boundary from the right. Recall that the `GameLoop.stop()` function will signal, stop the game loop, and call the `unloadScene()` function.
- The `unloadScene()` function is called when the game loop has stopped. At this point, since the game loop is not running, no update or draw function will be executed, and all resources are free from being accessed. This is the opportunity to unload and free up `ResourceMap` entries.
- At the end of the `unloadScene()` function, the next scene, `MyGame`, is created and passed to the `EngineCore.startScene()` function, which will load, initialize, and run the `MyGame` scene.

The MyGame Scene

By this point, it may become clear that the `MyGame` scene definition is going to be quite similar to that of the `BlueLevel` where the simple scene will include a camera and two `Renderable` objects. The major distinction is that `MyGame` defines its entire scene in the `initialize()` function and does not load its scene from an external file. As in the case of `BlueLevel`, because of the similarities, only important differences will be highlighted in the code listings.

1. Change the `MyGame` constructor to inherit from the abstract `Scene` object.

```
function MyGame() {
    // ... similar to previous code ...
    // ... declare a camera and two squares ...
}
gEngine.Core.inheritPrototype(MyGame, Scene);
```

The `MyGame` constructor is similar to the previous examples, with the exception of not defining the file path to a scene file and the call to the `inheritPrototype()` function to set `Scene` as the parent in inheritance.

2. Override the `Scene` functions as follows:

```
MyGame.prototype.initialize = function() {
    // ... similar to previous code ...
    // ... allocate and define the camera and two Renderables ...
};

MyGame.prototype.draw = function() {
    // ... similar to previous code ...
    // ... set up the camera and draw the two Renderables
};

MyGame.prototype.update = function() {
    // ... identical to MyGame.update from previous project ...

    if (gEngine.Input.isKeyPressed(gEngine.Input.keys.Left)) {
        xform.incXPosBy(-deltaX);
        if (xform.getXPos() < 11) { // this is the left-bound
            gEngine.GameLoop.stop();
        }
    }
};

MyGame.prototype.unloadScene = function() {
    var nextLevel = new BlueLevel(); // next level to be loaded
    gEngine.Core.startScene(nextLevel);
};
```

Similar to the case of BlueLevel, many of the functions are similar to the previous projects with the details not shown in the previous code listing. However, do take note of the following:

- a. The `loadScene()` function is not defined. Since `MyGame` does not have any external resources to load, there is no need to override this function.
- b. The `update()` function is almost identical to that from the `BlueLevel`. This should be expected because the two levels behave in almost an identical manner. It is once again important to note that when a level transition condition becomes favorable, the `GameLoop.stop()` function should be called to halt the game loop and cause the calling of the `unloadScene()` function.
- c. The `unloadScene()` function is defined even though `MyGame` does not have any particular resources to unload. This is because the `unloadScene()` function is responsible for creating and transitioning the game to the next level. In this case, when the rectangle touches the left boundary, the game will transition back to the `BlueLevel`.

You can now run the project and experience the loading and unloading of the two scenes. Your game engine now has a well-defined interface for working with its client. This interface follows the well-defined protocol of the `Scene` object.

- Constructor: For declaring variables and defining constants
- `loadScene()`: For initiating the asynchronous loading of external resources
- `initialize()`: For instantiating the variables and setting up the game scene
- `update()/draw()`: For continuously receiving player input, implementing game logic, and displaying the game state
- `unloadScene()`: For unloading external resources and initiating the loading of the next scene by calling the `EngineCore.startScene()` function

Any objects that define the previous methods can be loaded and interacted with by your game engine. You can experiment with creating other levels and even trying the loading of the `MyGame` level in the `MyGame.unloadScene()` function.

Audio

Audio is an essential element of all video games. In general, audio effects in games fall into two categories. The first category is background audio. This includes background music or ambient effects and is often used to bring atmosphere or emotion to different portions of the game. The second category is sound effects. Sound effects are useful for all sorts of purposes, from notifying users of game actions to hearing the footfalls of your hero character. Usually, sound effects represent a specific action, triggered either by the user or by the game itself. Such sound effects are often thought of as an audio cue.

One important difference between these two types of audio is how you control them. 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 be started and stopped at will. These capabilities are useful for stopping the background track completely and starting another one.

The Audio Support Project

This project is identical to the previous one where you can move the front rectangle left or right with the arrow keys, and the intersection with the left boundary triggers the loading of next scene, MyGame loads BlueLevel, and vice versa. However, in this version, each scene plays background music and triggers a brief audio cue when the left/right arrow key is pressed. The implementation of this project also reinforces the concept of loading and unloading of external resources and the audio clips themselves. You can see an example of this project running in Figure 4-6. The source code to this project is defined in the Chapter4\4.6.AudioSupport folder.



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

The goals of the project are as follows:

- To add audio support to the resource management system
- To provide an interface to play audio for games
- To optimize and facilitate resource sharing with reference counts of individual resources

You can find the following audio files in the Chapter4\4.6.AudioSupport\public_html\assets\sounds folder:

- BGClip.mp3
- BlueLevel_cue.wav
- MyGame_cue.wav

Notice that the previous audio files are in two formats, .mp3 and .wav. While both are supported, audio files of these formats should be used with care. Files in .mp3 format are compressed and are suitable for storing longer durations of audio content, for example, for background music. Files in .wav format are uncompressed and should contain only very short audio snippet, for example, for storing cue effects.

Define AudioClips Component with Web Audio API

While audio and text files are completely different, from the perspective of your game engine implementation, there are two important similarities. First, both are external resources and thus will be implemented similarly as engine components in the `src/Engine/Core/Resources` folder. Second, both involve standardized file formats with existing well-defined API utilities. The Web Audio API will be used for the actual retrieving and playing of sound files. Even though this API offers vast capabilities, in the interests of focusing on the rest of the game engine development, only basic supports for background audio and effect cues are discussed.

Note Interested readers can learn more about the Web Audio API from <http://www.w3.org/TR/webaudio/>.

1. In the `src/Engine/Core/Resources` folder, create a new file and name it `Engine_AudioClips.js`. This file will implement the `AudioClips` engine component.

```
var gEngine = gEngine || { };
gEngine.AudioClips = (function(){
    var mPublic = {};
    return mPublic;
}());
```

2. Declare variables to maintain references to the Web Audio context and background music.

```
var mAudioContext = null;
var mBgAudioNode = null;
```

3. Define a function to create a reference to the Web Audio context and store the results in `mAudioContext`.

```
var InitAudioContext = function() {
    try {
        var AudioContext = window.AudioContext || window.webkitAudioContext;
        mAudioContext = new AudioContext();
    }
    catch(e) {alert("Web Audio Is not supported.");}
};
```

4. Define the function to load an audio file asynchronously.

```
var loadAudio = function (clipName) {
    if (!(gEngine.ResourceMap.isAssetLoaded(clipName))) {
        // Update resources in load counter.
        gEngine.ResourceMap.asyncLoadRequested(clipName);

        // Asyncrounsly request the data from server.
        var req = new XMLHttpRequest();
```

```

req.onreadystatechange = function () {
    if ((req.readyState === 4) && (req.status !== 200)) {
        alert(clipName + ": loading failed!
            [Hint: you cannot double click index.html to run
            this project. " +
            "The index.html file must be loaded by a web-server.]");
    }
};

req.open('GET', clipName, true);
// Specify that the request retrieves binary data.
req.responseType = 'arraybuffer';

req.onload = function () {
    // Asynchronously decode, then call the function in parameter.
    mAudioContext.decodeAudioData(req.response,
        function(buffer) {
            gEngine.ResourceMap.asyncLoadCompleted(clipName, buffer);
        }
    );
};

req.send();
} else {
    gEngine.ResourceMap.incAssetRefCount(clipName);
}
};

```

Note the similarity between the `loadAudio()` function and the `loadTextFile()` function where the `ResourceMap` is first consulted to ensure the requested resource is not already loaded, followed by registering a new asynchronous load, and lastly, when the load is completed, the request for storage of the loaded resource.

- `ResourceMap.isAssetLoaded()`: Checks whether the requested resource is already loaded. Notice that if a resource is indeed already loaded, there is a call to a `ResourceMap.incAssetRefCount()` function. This function ensures resources are reference counted and can be properly shared across different load requests. Details to reference counting will be described in the next section.
- `ResourceMap.asyncLoadRequested()`: Registers an outstanding asynchronous loading with the `ResourceMap`.
- `ResourceMap.asyncLoadCompleted()`: Informs the `ResourceMap` that an asynchronous loading has completed and stores the loaded asset in the `ResourceMap`.

Notice the actual loading of the audio file.

- The `XMLHttpRequest` retrieves the corresponding data as an `arrayBuffer` object, which holds binary data.
- The loaded binary data is decoded asynchronously into playable audio format by the `mAudioContext.decodeAudioData()` function. Be aware that the `asyncLoadCompleted()` function is called only when the decoding has been completed.

5. Add a function to unload the audio file.

```
var unloadAudio = function(clipName) {
    gEngine.ResourceMap.unloadAsset(clipName);
};
```

6. Now add a function to play the entire duration of an audio clip.

```
var playACue = function(clipName) {
    var clipInfo = gEngine.ResourceMap.retrieveAsset(clipName);
    if (clipInfo != null) {
        // SourceNodes are one use only.
        var sourceNode = mAudioContext.createBufferSource();
        sourceNode.buffer = clipInfo;
        sourceNode.connect(mAudioContext.destination);
        sourceNode.start(0);
    }
};
```

The `playACue()` function uses the audio file path as a resource name to find the loaded asset from the `ResourceMap` and then invokes the Web Audio API to play the audio clip. Notice that no reference to the `sourceNode` 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.

7. Add the functionality to start, stop, and test for background audio.

```
var playBackgroundAudio = function(clipName) {
    var clipInfo = gEngine.ResourceMap.retrieveAsset(clipName);
    if (clipInfo != null) {
        // Stop audio if playing.
        stopBackgroundAudio();
        mBgAudioNode = mAudioContext.createBufferSource();
        mBgAudioNode.buffer = clipInfo;
        mBgAudioNode.connect(mAudioContext.destination);
        mBgAudioNode.loop = true;
        mBgAudioNode.start(0);
    }
};
var stopBackgroundAudio = function() {
    // Check if the audio is playing.
    if(mBgAudioNode !== null) {
        mBgAudioNode.stop(0);
        mBgAudioNode = null;
    }
};
var isBackgroundAudioPlaying = function() {
    return (mBgAudioNode !== null);
};
```

In this case, notice that the `mBgAudioNode` keeps a reference to the currently running audio, and thus it is possible to stop the clip.

8. Finally, remember to add the public functions to the public interface.

```
var mPublic = {
    InitAudioContext: initAudioContext,
    loadAudio: loadAudio,
    unloadAudio: unloadAudio,
    playACue: playACue,
    playBackgroundAudio: playBackgroundAudio,
    stopBackgroundAudio: stopBackgroundAudio,
    isBackgroundAudioPlaying: isBackgroundAudioPlaying
};
return mPublic;
```

Counting Resource References

It is common for a resource to be shared and reused over the course of a game, for example, an audio cue effect or a background clip that is reused in different levels. In this case, a game may inevitably issue multiple load requests on these resources, potentially each time when a particular resource is needed. As described, loading external resources is a relatively expensive endeavor and should be avoided if possible. For this reason, in both the audio and text file loading, the `ResourceMap` is first consulted with the `isAssetLoaded()` function before the actual asynchronous load command is issued. This way, if a client issues more than one load request for the same resource, the game engine will perform only one actual load operation.

Since the client of the game engine is responsible for always unloading external resources that it has loaded, the support of multiple load requests must be complemented with the corresponding support for unload requests. For example, if there are two requests to load a particular resource, then it can be assumed that this resource is used in two separate cases, and there will be two corresponding unload requests for this resource. In this case, the resource must be kept in the `ResourceMap` until the second unload request is received. To properly support this behavior, individual resources must be reference counted. An integer counter is introduced to keep track of the number of load and unload requests on a resource, and the resource will be removed only when the value of this counter is zero.

To implement reference counting, an integer counter must be added to the `MapEntry` object in the `Engine_ResourceMap.js` file.

1. Open `Engine_ResourceMap.js` and add a counter variable with an initial value of 1 to the `MapEntry` constructor.

```
var MapEntry = function(rName) {
    this.mAsset = rName;
    this.mRefCount = 1;
};
```

Since the `MapEntry` object is created when the first load request is made on a resource, the initial value of 1 represents that there is one load command issued on the corresponding resource.

2. Add a function to increase the reference counter counting the number of times load commands have been issued on the resource, and remember to add this function to the public interface.

```
var incAssetRefCount = function(rName) {
    mResourceMap[rName].mRefCount += 1;
};
```

Each resource loader (for example, `Engine.AudioClip` or `Engine.TextFileLoader`) is responsible for calling this function when it detects a target resource has already been loaded. For example, earlier you saw this function being called from the `loadAudio()` function when an audio clip to be loaded is already present in the `ResourceMap`.

3. Complete the reference counting implementation by updating the `unloadAsset()` function.

```
var unloadAsset = function(rName) {
    var c = 0;
    if (rName in mResourceMap) {
        mResourceMap[rName].mRefCount -= 1;
        c = mResourceMap[rName].mRefCount;
        if (c === 0)
            delete mResourceMap[rName];
    }
    return c;
};
```

Notice that by checking the return value of the `unloadAsset()` function, a client can verify how many more unloads are necessary to free up the associated memory.

Testing the Audio Component

To test the audio component, you must copy the necessary audio files into your game project. Create a new folder in the assets folder and name it sounds. Copy the `BG_Clip.mp3`, `BlueLevel_cue.wav`, and `MyGame_cue` files into the sounds folder. You will now need to update the `MyGame` and `BlueLevel` implementations to load and use these audio resources.

Change `MyGame.js`

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.

```
function MyGame() { // audio clips: supports both mp3 and wav formats
    this.kBgClip = "assets/sounds/BGClip.mp3";
    this.kCue = "assets/sounds/MyGame_cue.wav";
    // ... Identical to previous code ...
}
```

Recall that these file paths are used as resource names for loading, storage, and retrieval. Declaring these as constants for later references is a good software engineering practice.

2. Request the loading of audio clips in the `loadScene()` function.

```
MyGame.prototype.loadScene = function() {
    gEngine.AudioClips.loadAudio(this.kBgClip);
    gEngine.AudioClips.loadAudio(this.kCue);
}
```

3. Remember to unload external resources that are loaded.

```
MyGame.prototype.unloadScene = function() {
    // stop the background audio before unloading it
    gEngine.AudioClips.stopBackgroundAudio();

    // unload the scene resources
    // gEngine.AudioClips.unloadAudio(this.kBgClip);
    //      The above line is commented out on purpose because
    //      you know this clip will be used elsewhere in the game
    //      so you decide to not unload this clip!!
    gEngine.AudioClips.unloadAudio(this.kCue);

    // starts the next level
    var nextLevel = new BlueLevel(); // next level to be loaded
    gEngine.Core.startScene(nextLevel);
};
```

While it is important to unload resources, there can be exceptions. In this case, the background music clip is not unloaded because `BlueLevel` will be using the same resource. In addition, this is an excellent opportunity for testing resources reference counting.

4. Start the background audio at the end of the `initialize()` function.

```
MyGame.prototype.initialize = function() {
    // ... identical to previous code ...
    gEngine.AudioClips.playBackgroundAudio(this.kBgClip);
}
```

5. In the `update()` function, cue the players when the left and right arrow keys are pressed.

```
MyGame.prototype.update = function(){
    // ... Identical to previous code ...

    // Support hero movements
    if (gEngine.Input.isKeyPressed(gEngine.Input.keys.Right)) {
        gEngine.AudioClips.playACue(this.kCue);
        xform.incXPosBy(deltaX);
        // ... identical to previous code ...
    }

    if (gEngine.Input.isKeyPressed(gEngine.Input.keys.Left)) {
        gEngine.AudioClips. playACue (this.kCue);
        xform.incXPosBy(-deltaX);
        // ... identical to previous code ...
    }
};
```

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:

```
function BlueLevel() {
    // audio clips: supports both mp3 and wav formats
    this.kBgClip = "assets/sounds/BGClip.mp3";
    this.kCue = "assets/sounds/BlueLevel_cue.wav";
    // ... Identical to previous code ...
}
```

2. In addition to the scene file, request the loading of the audio clips in the `loadScene()` function.

```
BlueLevel.prototype.loadScene = function() {
    // load the scene file
    gEngine.TextFileLoader.loadTextFile(this.kSceneFile,
        gEngine.TextFileLoader.eTextFileType.eXMLFile);
    // loads the audios
    gEngine.AudioClips.loadAudio(this.kBgClip);
    gEngine.AudioClips.loadAudio(this.kCue);
};
```

3. Remember to stop background audio and unload all external resources.

```
BlueLevel.prototype.unloadScene = function() {
    // stop the background audio
    gEngine.AudioClips.stopBackgroundAudio();

    // unload the scene file and loaded Resources
    gEngine.TextFileLoader.unloadTextFile(this.kSceneFile);
    gEngine.AudioClips.unloadAudio(this.kBgClip);
    gEngine.AudioClips.unloadAudio(this.kCue);

    var nextLevel = new MyGame(); // load the next level
    gEngine.Core.startScene(nextLevel);
};
```

4. In the same manner as MyGame, start the background audio in the `initialize()` function and cue the player when the left and right keys are pressed in the `update()` function.

```
BlueLevel.prototype.initialize = function() {
    // ... identical to previous code ...
    gEngine.AudioClips.playBackgroundAudio(this.kBgClip);
};

BlueLevel.prototype.update = function(){
    // ... Identical to previous code ...
```

```

// Support hero movements
if (gEngine.Input.isKeyPressed(gEngine.Input.keys.Right)) {
    gEngine.AudioClips.playACue(this.kCue);
    xform.incXPosBy(deltaX);
    // ... identical to previous code ...
}

if (gEngine.Input.isKeyPressed(gEngine.Input.keys.Left)) {
    gEngine.AudioClips.playACue (this.kCue);
    xform.incXPosBy(-deltaX);
    // ... identical to previous code ...
}
;

```

You can now run the project and listen to the wonderful audio feedback. Take note that when transitioning from the MyGame level to the BlueLevel, the background music reloading request actually triggers an integer increment (instead of a full reload/re-decode) and the music was stopped and restarted. If you press and hold the arrow keys, there will be many cues repeatedly played. In fact, there are so many cues echoed that the sound effects are blurred into an annoying blast. This is an excellent example to illustrate the importance of using audio cues with care and ensuring each individual cue is nice and short. You can try tapping the arrow keys to listen to more distinct and pleasant sounding cues, or you can simply replace the `isKeyPressed()` function with the `isKeyClicked()` function and listen to each 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 and has everything needed in order to create basic games. 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 as well as audio cues.

These components separately have little in common but together make up the core fundamentals of nearly every game. As you implement these common 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 provided by the engine 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 or the technical foundation contributing to the apparent immediate connection between what the player does and how the game responds. If a player grabs 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 that begins 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 object and action responsiveness play 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 actions 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 chapter, for example, many of the pieces will already be in place to begin constructing game levels. You’ve provided players with the ability to manipulate two individual elements on the screen (the red and white squares), and all that remains in order to create a basic “chunk” of game play is to design a causal chain using those elements that results in a new event when completed. Imagine the Keyboard Support project is your game. How might you use what’s available to create a causal chain? You might choose to play with the relationship between the squares, perhaps requiring that the red square be contained completely within the white square in order to complete the level and move on to the next challenge; once the player met the conditions of that causal chain (that is, once the player successfully placed the red square in the white square), the level would complete. This basic mechanic isn’t quite enough on its own to create a fun experience, but by including just a few of the other eight elements of game design (systems design, setting, visual design, music and audio, and the like), it’s possible to turn this one basic interaction into an almost infinite number of engaging experiences and to begin creating that sense of presence for players. You’ll add more game design elements to these exercises as you continue through subsequent chapters.

The Resource Map and shader Loads project, the Scene File project, and the Scene Objects project are designed to help you begin thinking about architecting game designs from the ground up for maximum efficiency so that problems such as asset loading delays that detract from the player’s sense of presence are minimized. As you begin designing games with multiple stages and levels and many assets, a resource management plan becomes essential. Understanding the limits of available memory and how to smartly load and unload assets can mean the difference between a great experience and a frustrating experience.

We experience the world through our senses, and our feeling of presence in games tends to be magnified as we include additional sensory inputs. The Audio Support project adds basic audio to our simple state-changing exercise from the Scene Objects project in the form of a constant background score to provide ambient mood and includes a distinct movement sound for each of the two areas. Compare the two experiences and consider how different they feel because of the presence of sound cues. Although the visual and interaction experience is identical between the two, the Audio Support project begins to add some emotional cues because of the beat of the background score and the individual tones the rectangle makes as it moves. Audio is a powerful enhancement to interactive experiences and can dramatically increase a player’s sense of presence in game environments; as you continue through the chapters, you’ll explore how audio contributes to game design in more detail.

CHAPTER 5



Working with Textures, Sprites, and Fonts

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

Introduction

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.

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 large amount of data that must be transferred and processed, loading images can be slower than real 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 and form 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, and 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 supporting 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 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, you'll hear "an image" and "a texture" often used interchangeably.

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.

Note Just as a pixel is a color location in an image, a texel is a color location in a texture.

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.

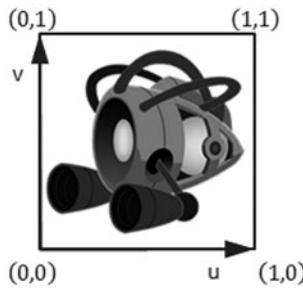


Figure 5-1. The Texture Coordinate System and the corresponding uv values defined for all images

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.

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.

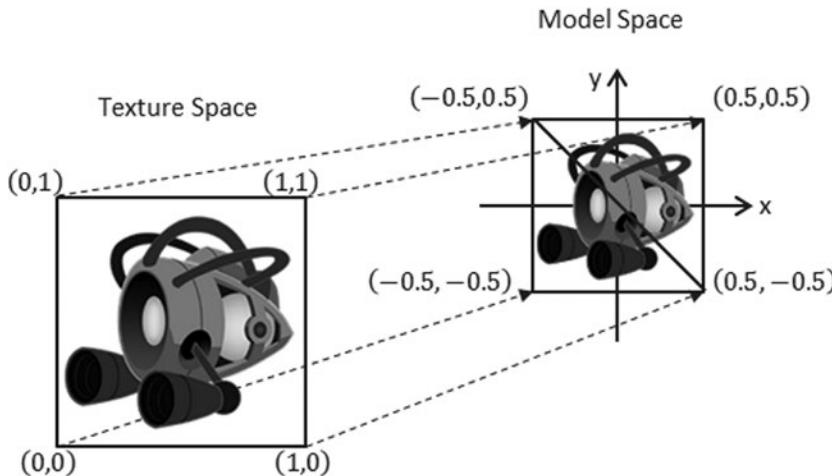


Figure 5-2. Defining texture space uv values to map the entire image onto the geometry in Model Space

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 naturally appearing 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 Chapter 5/5.1.TextureShaders folder.

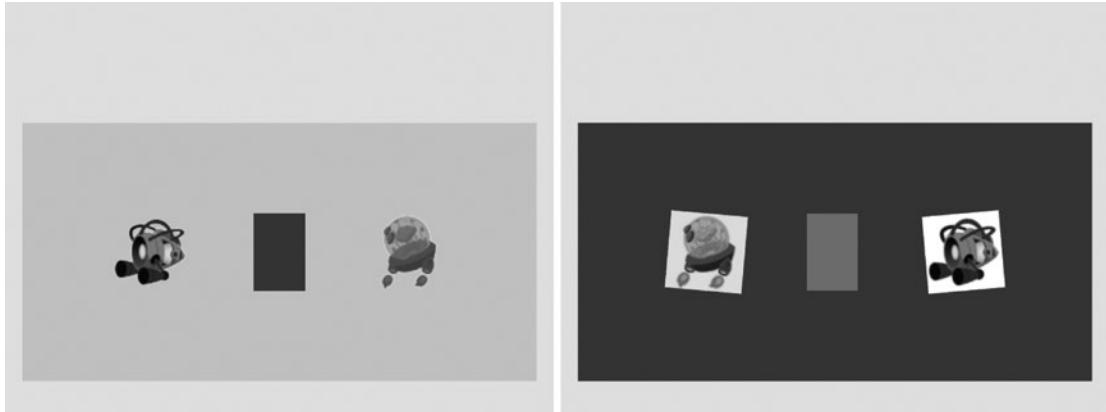


Figure 5-3. Running the Texture Shaders project with both scenes

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 (`BlueLevel.xml`) and four images (`minion_collector.jpg`, `minion_collector.png`, `minion_portal.jpg`, and `minion_portal.png`).

Overview

Creating and integrating textures involves relatively significant changes and new classes to be added to the game engine. The following overview contextualizes and describes the reasons for the changes:

- `TextureVS.gls1` and `TextureFS.gls1`: These are new files created to define GLSL shaders for supporting drawing with uv coordinates. Recall that the GLSL shaders must be loaded into WebGL and compiled during the initialization of the game engine.
- `Engine_VertexBuffer.js`: This file is modified to create a corresponding uv coordinate buffer to define the texture coordinate for the vertices of the unit square.
- `TextureShader.js`: This is a new file that defines `TextureShader` as a subclass of `SimpleShader` to interface the game engine to the corresponding GLSL shaders (`TextureVS` and `TextureFS`).
- `Engine_DefaultResources.js`: This is a new file that defines a core engine component to facilitate the sharing of systemwide resources. In this case, it's to facilitate the sharing of both `SimpleShader` and `TextureShader` by the corresponding `Renderable` objects.
- `Renderable.js`: This file is modified to facilitate `Renderable` serving as the base class to all future types of `Renderable` objects and to share the `Shader` resource provided by `gEngine_DefaultResources`.
- `TextureRenderable.js`: This is a new file that defines `RenderableTexture` as a subclass of `Renderable` to facilitate the creation, manipulation, and drawing of multiple instances of textured objects.
- `Engine_Core.js`: This file is modified to configure WebGL to support drawing with a texture map.
- `Engine_Textures.js`: This is a new file that defines the core engine component that is capable of loading, activating (for rendering), and unloading texture images.
- `MyGame.js` and `BlueLevel.js`: These game engine client files are modified to test the new texture mapping functionality.

Extension of Shader/Renderable Architecture

Recall that the `Shader/Renderable` object pair is designed to load 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 `Shader` 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 `Shader` object created in the game engine, in `Engine.DefaultResources`, and this instance is shared by all `Renderable` objects.

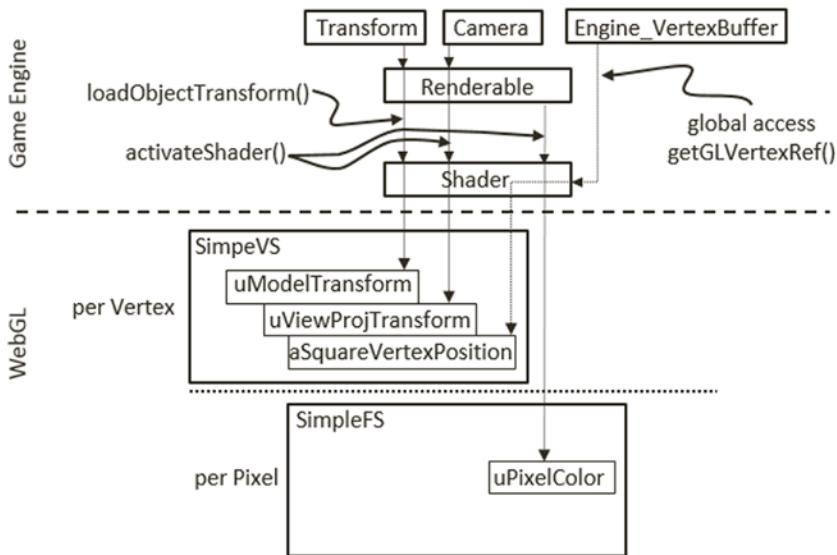


Figure 5-4. The Shader and Renderable architecture

The proper support of texture mapping demands new GLSL vertex and fragment shaders and thus requires that a corresponding shader and renderable object pair be defined in the game engine. As illustrated in Figure 5-5, both the GLSL TextureVS/FS shaders and TextureShader/TextureRenderable object pair are extensions (or subclasses) to the corresponding existing objects. The TextureShader/TextureRenderable object pair extends from the corresponding Shader/Renderable objects to forward texture coordinates to the GLSL shaders. The TextureVS/FS shaders are extensions to the corresponding SimpleVS/FS shaders to read texels from the provided texture map when computing pixel colors. Note that since GLSL does not support subclassing, the TextureVS/FS source code is copied from the SimpleVS/FS files.

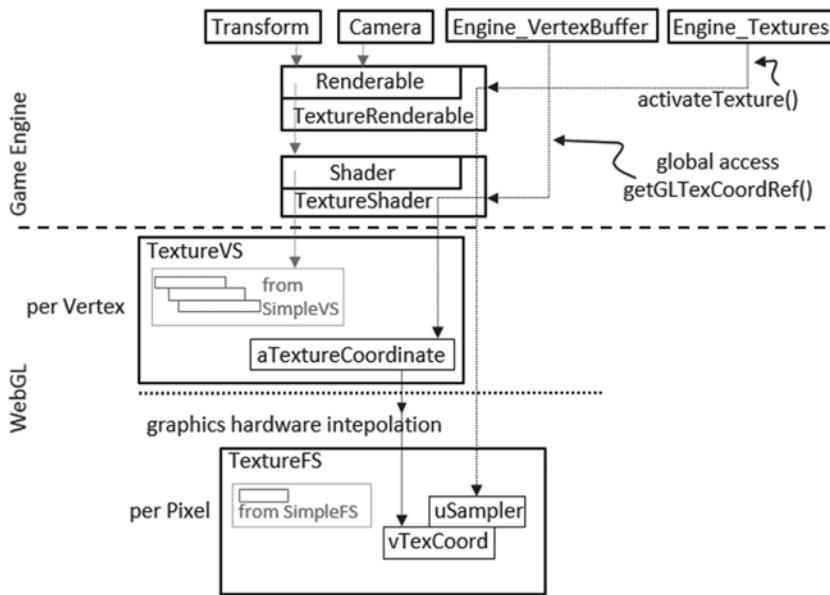


Figure 5-5. The TextureVS/FS GLSL shaders and the corresponding TextureShader/TextureRenderable object pair

GLSL Texture Shader

To support drawing with textures, you must create a shader that accepts both geometric (xy) and texture (uv) coordinates at each of the vertices. You will create new GLSL texture vertex and fragment shaders by copying and modifying the corresponding SimpleVS and SimpleFS programs. Now, create the texture vertex shader.

1. Create a new file in the `src/GLSLShaders/` folder and name it `TextureVS.gls1`.
2. Add the following code to the `TextureVS.gls1` file:

```

attribute vec3 aSquareVertexPosition; // Expects one vertex position
attribute vec2 aTextureCoordinate;

// texture coordinate that will map the entire image to the entire square
varying vec2 vTexCoord;

// to transform the vertex position
uniform mat4 uModelTransform;
uniform mat4 uViewProjTransform;

void main(void) {
    gl_Position = uViewProjTransform * uModelTransform *
        vec4(aSquareVertexPosition, 1.0);

    // pass the texture coordinate to the fragment shader
    vTexCoord = aTextureCoordinate;
}

```

You may notice that the `TextureVS` shader is similar to the `SimpleVS` shader, with only three additional lines of code.

- a. Add the `aTextureCoordinate` attribute. This defines a vertex to include a `vec3` (`aSquareVertexPosition`, the xyz position of the vertex) and a `vec2` (`aTextureCoordinate`, the uv coordinate of the vertex).
- b. Declare 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.
- c. Assign the vertex uv coordinate values to the `vTexCoord` variable for interpolation and forwarding to the fragment shader.

With the vertex shader defined, you can now create the associated fragment shader.

1. Create a new file in the `src/GLSLShaders/` folder and name it `TextureFS.gls1`.
2. Add the following code to the `TextureFS.gls1` file to declare the variables:

```
precision mediump float; // precision for floating point computation

// The object that fetches data from texture.
// Must be set outside the shader.
uniform sampler2D uSampler;

// Color of pixel
uniform vec4 uPixelColor;

// The "varying" keyword is for signifying that the texture coordinate will be
// interpolated and thus varies.
varying vec2 vTexCoord;
```

The `sampler2D` data type is a GLSL utility that is capable of reading texel values from a 2D texture. In this case, the `uSampler` object will be bounded to a GLSL texture such that texel values can be sampled for every pixel rendered. The `uPixelColor` is the same as the one from `SimpleFS`. The `vTexCoord` is the interpolated uv coordinate value for each pixel.

3. Add the following code to compute the color for each pixel:

```
void main(void) {
    // texel color look up based on interpolated UV value in vTexCoord
    vec4 c = texture2D(uSampler, vec2(vTexCoord.s, vTexCoord.t));

    // tint the textured area. Leave transparent area as defined by the texture
    vec3 r = vec3(c) * (1.0-uPixelColor.a) + vec3(uPixelColor) * uPixelColor.a;
    vec4 result = vec4(r, c.a);

    gl_FragColor = result;
}
```

4. 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 `Engine_VertexBuffer.js` file. In a similar fashion, a corresponding buffer must be defined to supply texture coordinates to the GLSL shaders.

1. Modify `Engine_VertexBuffer.js` to define both xy and uv coordinates for the unit square.

```
// reference to the vertex positions for the square in the gl context
var mSquareVertexBuffer = null;

// reference to the texture positions for the square vertices in the gl context
var mTextureCoordBuffer = null;

// First: define the vertices for a square
var verticesOfSquare = [
    0.5, 0.5, 0.0,
    -0.5, 0.5, 0.0,
    0.5, -0.5, 0.0,
    -0.5, -0.5, 0.0
];

// Second: define the corresponding texture coordinates
var textureCoordinates = [
    1.0, 1.0,
    0.0, 1.0,
    1.0, 0.0,
    0.0, 0.0
];
```

As illustrated in Figure 5-2, the `textureCoordinates` variable defines the uv values for the corresponding four xy values of the unit square defined in `verticesOfSquare`, sequentially. 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. The new `mTextureCoordBuffer` instance variable will be initialized to refer to the WebGL buffer that stores the values of `textureCoordinates`.

2. Modify the `initialize()` function to the following:

```

var initialize = function () {
    var gl = gEngine.Core.getGL();

    // Step A: Allocate and store vertex positions into the webGL context
    // Create a buffer on the gGL context for our vertex positions
    mSquareVertexBuffer = gl.createBuffer();

    // Activate vertexBuffer
    gl.bindBuffer(gl.ARRAY_BUFFER, mSquareVertexBuffer);

    // Loads verticesOfSquare into the vertexBuffer
    gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(verticesOfSquare),
        gl.STATIC_DRAW);

    // Step B: Allocate and store texture coordinates
    // Create a buffer on the gGL context for our vertex positions
    mTextureCoordBuffer = gl.createBuffer();

    // Activate vertexBuffer
    gl.bindBuffer(gl.ARRAY_BUFFER, mTextureCoordBuffer);

    // Loads verticesOfSquare into the vertexBuffer
    gl.bufferData(gl.ARRAY_BUFFER,
        new Float32Array(textureCoordinates), gl.STATIC_DRAW);
};

```

Step B of the `initialize()` function handles the initialization of the texture coordinates as a WebGL buffer and is identical to how the vertex xy coordinates are handled with the `mTextureCoordBuffer` variable.

3. Add a function to retrieve the texture coordinates.

```
var getGLTexCoordRef = function() { return mTextureCoordBuffer; };
```

4. Finally, remember to add the changes to the public interface.

```

var mPublic = {
    initialize: initialize,
    getGLVertexRef: getGLVertexRef,
    getGLTexCoordRef: getGLTexCoordRef
};

```

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. In addition, you will create a new folder to organize the growing number of different shaders.

1. Create a new folder called Shaders in src/Engine. Move the SimpleShader.js file into this folder, and do not forget to update the reference path in index.html.
2. Create a new file in the src/Engine/Shaders/ folder and name it TextureShader.js. Add the following code to construct the object:

```
// constructor
function TextureShader(vertexShaderPath, fragmentShaderPath) {
    // Call super class constructor
    SimpleShader.call(this, vertexShaderPath, fragmentShaderPath);

    // reference to aTextureCoordinate within the shader
    this.mShaderTextureCoordAttribute = null;

    // get the reference of aTextureCoordinate from the shader
    var gl = gEngine.Core.getGL();
    this.mShaderTextureCoordAttribute =
        gl.getAttribLocation(this.mCompiledShader, "aTextureCoordinate");
}
// get all the prototype functions from SimpleShader
gEngine.Core.inheritPrototype(TextureShader, SimpleShader);
```

The SimpleShader.call() syntax invokes the constructor of SimpleShader with the current TextureShader object as the caller. This line of code is simply invoking the superclass constructor. Recall that the SimpleShader constructor will load and compile the GLSL shaders defined by the vertexShaderPath and fragmentShaderPath and will locate a reference to the mShaderVertexPositionAttribute attribute defined in the shader. In the rest of the TextureShader constructor, the mShaderTextureCoordAttribute keeps a reference to the aTextureCoordinate attribute defined in the TextureVS. In this way, both of the vertex attributes defined in TextureVS.gsl are referenced by the JavaScript TextureShader object.

3. Override the activateShader() function to enable the texture coordinate data.

```
// Overriding the Activation of the shader for rendering
TextureShader.prototype.activateShader = function(pixelColor, vpMatrix) {
    // first call the super class's activate
    SimpleShader.prototype.activateShader.call(this, pixelColor, vpMatrix);

    // now our own functionality: enable texture coordinate array
    var gl = gEngine.Core.getGL();
    gl.bindBuffer(gl.ARRAY_BUFFER, gEngine.VertexBuffer.getGLTexCoordRef());
    gl.enableVertexAttribArray(this.mShaderTextureCoordAttribute);
    gl.vertexAttribPointer(this.mShaderTextureCoordAttribute, 2, gl.FLOAT,
        false, 0,0);
};
```

The superclass activateShader.call() sets up the xy vertex position and passes the pixelColor to the shader. The rest of the code binds mShaderTextureCoordAttribute to the texture coordinate buffer defined in the gEngine.VertexBuffer component, as discussed in the Define and Set up Up Texture Coordinates section previously.

In this way, after the `activateShader()` function call, both the geometry's xy coordinate (`mShaderVertexPositionAttribute`) and the texture's uv coordinate (`mShaderTextureCoordAttribute`) of each vertex are connected to the corresponding buffers in the GLSL shaders.

Facilitate Sharing with Engine_DefaultResources

In the same manner as `SimpleShader` is a reusable resource, only one instance of the `TextureShader` needs to be created, and this instance can be shared. The `DefaultResources` component should be modified to reflect this.

1. In `Engine_DefaultResources.js`, add the variables to hold a texture shader.

```
// Texture Shader
var kTextureVS = "src/GLSLShaders/TextureVS.glsl"; // Path to VertexShader
var kTextureFS = "src/GLSLShaders/TextureFS.glsl"; // Path to FragmentShader
var mTextureShader = null;
```

2. Define a function to retrieve the texture shader.

```
var getTextureShader = function() { return mTextureShader; };
```

3. Add the creation of the texture shader to the `_createShaders()` function.

```
var _createShaders = function(callBackFunction) {
    mConstColorShader = new SimpleShader(kSimpleVS, kSimpleFS);
    mTextureShader = new TextureShader(kTextureVS, kTextureFS);
    callBackFunction();
};
```

4. Modify the `initialize()` function to load the source files for the texture shader.

```
var initialize = function(callBackFunction) {
    // constant color shader: SimpleVS, and SimpleFS
    gEngine.TextFileLoader.loadTextFile(kSimpleVS,
        gEngine.TextFileLoader.eTextFileType.eTextFile);
    gEngine.TextFileLoader.loadTextFile(kSimpleFS,
        gEngine.TextFileLoader.eTextFileType.eTextFile);

    // texture shader:
gEngine.TextFileLoader.loadTextFile(kTextureVS,
    gEngine.TextFileLoader.eTextFileType.eTextFile);
gEngine.TextFileLoader.loadTextFile(kTextureFS,
    gEngine.TextFileLoader.eTextFileType.eTextFile);

    gEngine.ResourceMap.setLoadCompleteCallback(
        function(){ _createShaders(callBackFunction); }
    );
};
```

- Lastly, remember to add the changes to the public interface.

```
var mPublic =
{
    initialize: initialize,
    getConstColorShader: getConstColorShader,
getTextureShader: getTextureShader
};
```

Renderable Texture Object

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.

Changes to the Renderable Object

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.

- Create the src/Engine/Renderables folder and move Renderable.js into this folder. Remember to update index.html to reflect the change.
- Modify the Renderable constructor to refer to SimpleShader, the constant color shader, by default.

```
function Renderable() {
    this.mShader = gEngine.DefaultResources.getConstColorShader();
    this.mXform = new Transform(); // transform that moves this object around
    this.mColor = [1, 1, 1, 1];
}
```

- Define a function that sets the shader for the Renderable.

```
Renderable.prototype._setShader = function(s) { this.mShader = s; };
```

This is a protected function designed for subclasses to modify the mShader variable to refer to the appropriate shaders for each corresponding subclass.

The TextureRenderable Object

You are now ready to create the TextureRenderable object. As noted, TextureRenderable is derived from and extends the Renderable object functionality to render the object with a texture mapped to it.

1. Create a new file in the `src/Engine/Renderables/` folder and name it `TextureRenderable.js`. Add the constructor.

```
// Constructor and object definition
function TextureRenderable(myTexture) {
    Renderable.call(this);
    Renderable.prototype.setColor.call(this, [1, 1, 1, 0]);
    // Alpha 0: switch off tinting
    Renderable.prototype._setShader.call(this, gEngine.DefaultResources.
        getTextureShader());
    this.mTexture = myTexture; // the object's texture, cannot be null.
}
gEngine.Core.inheritPrototype(TextureRenderable, Renderable);
```

`Renderable.call(this)` is a call to the superclass (`Renderable`) constructor. Likewise, the `setColor()` and `_setShader()` functions are invoked with the `TextureRenderable` context to set the corresponding variables in the superclass. As will be discussed, the `myTexture` parameter is the path to the file that contains the texture image.

2. Define a `draw()` function to overwrite the function defined in the `Renderable` object to support textures.

```
TextureRenderable.prototype.draw = function(vpMatrix) {
    // activate the texture
    gEngine.Textures.activateTexture(this.mTexture);
    Renderable.prototype.draw.call(this, vpMatrix);
};
```

The `activateTexture()` function activates and allows drawing with the specific texture. The details of this function will be discussed in the following section.

3. Finally, define a getter and setter for the texture reference.

```
TextureRenderable.prototype.getTexture = function() { return this.mTexture; };
TextureRenderable.prototype.setTexture = function(t) { this.mTexture = t; };
```

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 `Engine_Core.js`, update `_initializeWebGL()` according to the following:

```
// initialize the WebGL, the vertex buffer and compile the shaders
var _initializeWebGL = function(htmlCanvasID) {
    var canvas = document.getElementById(htmlCanvasID);
```

```

// Get standard webgl, or experimental
// binds webgl to the Canvas area on the web-page to the variable mGL
mGL = canvas.getContext("webgl", {alpha: false}) ||
      canvas.getContext("experimental-webgl", {alpha: false});

// Allows transparency with textures.
mGL.blendFunc(mGL.SRC_ALPHA, mGL.ONE_MINUS_SRC_ALPHA);
mGL.enable( mGL.BLEND ) ;

// Set images to flip the y axis to match the texture coordinate space.
mGL.pixelStorei(mGL.UNPACK_FLIP_Y_WEBGL, true);

if (mGL === null) {
    document.write("<br><b>WebGL is not supported!</b>");
}
};


```

The parameter passed to `canvas.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.

Create the Texture Management Engine Component

Like audio and text files, a new engine component must be defined to support the corresponding operations, including reading from the server file system, loading to the WebGL context, activating the WebGL texture buffer for drawing, and unloading from WebGL.

1. Create a new file in the `src/Engine/Core/` folder and name it `Engine_Textures.js`. This file will implement the Textures engine component.
2. Add the following object definition to represent a texture in the game engine:

```

function TextureInfo(name, w, h, id) {
    this.mName = name;
    this.mWidth = w;
    this.mHeight = h;
    this.mGLTexID = id;
};

```

`mWidth` and `mHeight` are the pixel resolution of the texture image, `mName` stores the path to the image file, and `mGLTexID` is a reference to the WebGL texture storage.

Note For an efficient implementation, many graphics hardware only supports texture with image resolutions in powers of 2, such as 2x4 ($2^1 \times 2^2$), or 4x16 ($2^2 \times 2^4$), or 64x256 ($2^6 \times 2^8$), and so on. This is also the case for your configuration of WebGL. All examples in this book work only with textures with resolutions that are powers of 2.

3. Now, define a Textures component similar to the other engine components.

```
var gEngine = gEngine || { };

gEngine.Textures = (function(){
    var mPublic = { };
    return mPublic;
}());
```

4. Define a function to load an image asynchronously.

```
// Loads an texture so that it can be drawn.
// If already in the map, will do nothing.
var loadTexture = function(textureName) {
    if (!(gEngine.ResourceMap.isAssetLoaded(textureName)))  {
        // Create new Texture object.
        var img = new Image();

        // Update resources in loading counter.
        gEngine.ResourceMap.asyncLoadRequested(textureName);

        // When the texture loads, convert it to the WebGL format then put
        // it back into the mTextureMap.
        img.onload = function () {
            _processLoadedImage(textureName, img);
        };
        img.src = textureName;
    } else {
        gEngine.ResourceMap.incAssetRefCount(textureName);
    }
};
```

Note the similarity between the loading of textures and the loading of audio or text files. In this case, once an image is loaded, it is passed to the `_processLoadedImage()` function with its file path as the name.

5. Add an `unloadTexture()` function to clean up the engine and release WebGL resources.

```
// Remove the reference to allow associated memory
// be available for subsequent garbage collection
var unloadTexture = function(textureName) {
    var gl = gEngine.Core.getGL();
    var texInfo = gEngine.ResourceMap.retrieveAsset(textureName);
    gl.deleteTexture(texInfo.mGLTexID);
    gEngine.ResourceMap.unloadAsset(textureName);
};
```

6. Now define a function to convert the format of an image and store it to the WebGL context.

```

var _processLoadedImage = function(textureName, image) {
    var gl = gEngine.Core.getGL();

    // Creates a WebGL texture object
    var textureID = gl.createTexture();

    // bind texture with the current texture functionality in webGL
    gl.bindTexture(gl.TEXTURE_2D, textureID);

    // Load the texture into the texture data structure with descriptive info.
    // Parameters:
    // 1: Which "binding point" or target the texture is being loaded to.
    // 2: Level of detail. Used for mipmaping. 0 is base texture level.
    // 3: Internal format. The composition of each element, i.e. pixels.
    // 4: Format of texel data. Must match internal format.
    // 5: The data type of the texel data.
    // 6: Texture Data.
    gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE, image);

    // Creates a mipmap for this texture.
    gl.generateMipmap(gl.TEXTURE_2D);

    // Tells WebGL we are done manipulating data at the mGL.TEXTURE_2D target.
    gl.bindTexture(gl.TEXTURE_2D, null);

    var texInfo = new TextureInfo(textureName,
        image.naturalWidth, image.naturalHeight, textureID);
    gEngine.ResourceMap.asyncLoadCompleted(textureName, texInfo);
};

```

The `createTexture()` function creates a WebGL texture buffer and returns a unique ID. The `texImage2D()` function stores the image into the WebGL texture buffer, and `generateMipmap()` computes mipmap for the texture. Lastly, a `TextureInfo` object is instantiated to refer to the WebGL texture and stored into `ResourceMap` according to the file path to the texture image file.

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.

```
var activateTexture = function (textureName) {
    var gl = gEngine.Core.getGL();
    var texInfo = gEngine.ResourceMap.retrieveAsset(textureName);

    // Binds our texture reference to the current webGL texture functionality
    gl.bindTexture(gl.TEXTURE_2D, texInfo.mGLTexID);

    // To prevent texture wrappings
    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.
    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 pixel-graphics where you want the texture to look "sharp"
    // do the following:
    // gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
    // gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
};
```

- a. The `retrieveAsset()` function locates the `TextureInfo` object from the `ResourceMap` based on the `textureName`. The located `mGLTexID` is used in the `bindTexture()` function to activate the corresponding WebGL texture buffer for rendering.
- b. The `texParameteri()` function defines the rendering behavior for the texture. The `TEXTURE_WRAP_S/T` parameters ensure that the texel values will not wrap around at the texture boundaries. The `TEXTURE_MAG_FILTER` parameter defines how to magnify a texture, in other words, when a low-resolution texture is rendered to many pixels in the game window. The `TEXTURE_MIN_FILTER` parameter defines how to minimize a texture, in other words, when a high-resolution texture is rendered to a small number of pixels. The `LINEAR` and `LINEAR_MIPMAP_LINEAR` configurations generate smooth textures by blurring the details of the original images, while the commented-out `NEAREST` option will result in sharp textures but color boundaries of the texture image may appear jagged.

Note In general, it is best to use texture images with similar resolution as the number of pixels occupied by the objects in the game.

8. Define a function to deactivate a texture as follows:

```
var deactivateTexture = function() {
    var gl = gEngine.Core.getGL();
    gl.bindTexture(gl.TEXTURE_2D, null);
};
```

This function sets the WebGL context to a state of not working with a texture.

9. Add a getter function to retrieve the texture information by its name (a file path).

```
var getTextureInfo = function(textureName) {
    return gEngine.ResourceMap.retrieveAsset(textureName);
}
```

10. Finally, remember to add the public functions to the public interface.

```
// Public interface for this object. Anything not in here will
// not be accessible.
var mPublic = {
    loadTexture: loadTexture,
    unloadTexture: unloadTexture,
    activateTexture: activateTexture,
    deactivateTexture: deactivateTexture,
    getTextureInfo: getTextureInfo
};
return mPublic;
```

Testing of Texture Mapping Functionality

With the previous 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 highlights transparent texture maps with alpha channel using PNG images, and the BlueScene scene shows corresponding textures with images in JPEG format.

As in all cases of building a game, it is essential to ensure that all external resources are located at proper locations. Recall that the assets folder is created specifically for this purpose. Take note of the four new texture files located in the assets folder: `minion_collector.jpg`, `minion_collector.png`, `minion_portal.jpg`, and `minion_portal.png`.

Modify the BlueLevel Scene File to Support Textures

The `BlueLevel.xml` scene file is modified from the previous example to support texture mapping.

```
<MyGameLevel>
    <!-- cameras -->
        <!-- Viewport: x, y, w, h -->
    <Camera CenterX="20" CenterY="60" Width="20"
        Viewport="20 40 600 300"
        BgColor="0 0 1 1.0"/>

    <!-- The red rectangle -->
    <Square PosX="20" PosY="60" Width="2" Height="3" Rotation="0" Color="1 0 0 1" />
```

```

<!-- Textures Square -->
<TextureSquare PosX="15" PosY="60" Width="3" Height="3" Rotation="-5"
    Color="1 0 0 0.3"
    Texture="assets/minion_portal.jpg" />

<TextureSquare PosX="25" PosY="60" Width="3" Height="3" Rotation="5"
    Color="0 0 0 0"
    Texture="assets/minion_collector.jpg"/>
    <!-- without tinting, alpha should be 0 -->
</MyGameLevel>

```

The TextureSquare element is similar to Square with the addition of a Texture attribute that specifies which image file should be used as a texture map for the square. Note that as implemented in TextureFS.gls, the alpha value of the Color element is used for tinting the texture map. The previous code shows 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, there are the white areas outside the portal and collector minions in the right image of Figure 5-3.

Modify SceneFileParser

The scene file parser, `SceneFileParser.js`, is modified to support the parsing of the updated `BlueScene.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/MyGame/Util` folder.

Test BlueLevel with JPEGs

The modifications to `BlueLevel.js` are in the `loadScene()`, `unloadScene()`, and `initialize()` functions where the texture images are loaded and unloaded and new TextureRenderable objects are parsed.

1. Modify the constructor to define constants to represent the texture images that will be used.

```

function BlueLevel() {
    // scene file name
    this.kSceneFile = "assets/BlueLevel.xml";

    // textures: ( Note: jpg does not support transparency )
    this.kPortal = "assets/minion_portal.jpg";
    this.kCollector = "assets/minion_collector.jpg";

    // all square
    this.mSqSet = [];           // these are the renderable objects

    // The camera to view the rectangles
    this.mCamera = null;
};

gEngine.Core.inheritPrototype(BlueLevel, Scene);

```

2. Initiate loading of the textures in the `loadScene()` function.

```
BlueLevel.prototype.loadScene = function() {
    // load the scene file
    gEngine.TextFileLoader.loadTextFile(this.kSceneFile,
        gEngine.TextFileLoader.eTextFileType.eXMLFile);

    // load the textures
    gEngine.Textures.loadTexture(this.kPortal);
    gEngine.Textures.loadTexture(this.kCollector);
};
```

3. Likewise, add code to clean up by unloading the textures in the `unloadScene()` function.

```
BlueLevel.prototype.unloadScene = function() {
    // unload the scene file and loaded resources
    gEngine.TextFileLoader.unloadTextFile(this.kSceneFile);
    gEngine.Textures.unloadTexture(this.kPortal);
    gEngine.Textures.unloadTexture(this.kCollector);

    var nextLevel = new MyGame(); // load the next level
    gEngine.Core.startScene(nextLevel);
};
```

4. Parse the textured squares in the `initialize()` function.

```
BlueLevel.prototype.initialize = function() {
    var sceneParser = new SceneFileParser(this.kSceneFile);

    // Step A: Read in the camera
    this.mCamera = sceneParser.parseCamera();

    // Step B: Read all the squares and textureSquares
    sceneParser.parseSquares(this.mSqSet);
    sceneParser.parseTextureSquares(this.mSqSet);
};
```

5. Include extra code in the `update()` function to continuously change the tinting of the portal TextureRenderable, as follows:

```
BlueLevel.prototype.update = function() {
    // ... Identical to previous code ...

    // continuously change texture tinting
    var c = this.mSqSet[1].getColor();
    var 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 previous code continuously increases and wraps the alpha value of the `mColor` variable in the `TextureRenderable` object. Recall this variable is passed to `TextureShader` and then loaded into the `uPixelColor` of `TextureFS` for tinting the texture map results.
- c. As defined in the first `TextureSquare` element in the `BlueScene.xml` file, the color defined for the portal object is red. For this reason, when running the example for this project, in the blue level the portal object appears to be blinking in red.

Test MyGame with PNGs

Similar to the `BlueLevel` scene, `MyGame` is a straightforward modification of the previous example with changes to load and unload texture images and to create `TextureRenderable` objects.

1. Modify the `MyGame` constructor to define texture image files that will be used and the variables for referencing the `TextureRenderable` objects that will be instantiated.

```
function MyGame() {
    // textures: ( Note: supports png with transparency )
    this.kPortal = "assets/minion_portal.png";
    this.kCollector = "assets/minion_collector.png";

    // The camera to view the rectangles
    this.mCamera = null;

    // the hero and the support objects
    this.mHero = null;
    this.mPortal = null;
    this.mCollector = null;
};

gEngine.Core.inheritPrototype(MyGame, Scene);
```

2. Load the textures in `loadScene()`.

```
MyGame.prototype.loadScene = function() {
    // loads the textures
    gEngine.Textures.loadTexture(this.kPortal);
    gEngine.Textures.loadTexture(this.kCollector);
};
```

3. Make sure you remember to unload the textures in `unloadScene()`.

```
MyGame.prototype.unloadScene = function() {
    // Game loop not running, unload all assets
    gEngine.Textures.unloadTexture(this.kPortal);
    gEngine.Textures.unloadTexture(this.kCollector);
```

```

    // starts the next level
    var nextLevel = new BlueLevel(); // next level to be loaded
    gEngine.Core.startScene(nextLevel);
}

```

4. Create and initialize the TextureRenderables objects in the initialize() function.

```

MyGame.prototype.initialize = function() {
    // Step A: set up the cameras
    this.mCamera = new Camera(
        vec2.fromValues(20, 60), // position of the camera
        20, // width of camera
        [20, 40, 600, 300] // viewport (orgX, orgY, 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 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 TextureRenderable(this.kCollector);
    this.mCollector.setColor([0, 0, 0, 0]); // No tinting
    this.mCollector.getXform().setPosition(15, 60);
    this.mCollector.getXform().setSize(3, 3);

    // Step C: Create the hero object in blue
    this.mHero = new Renderable();
    this.mHero.setColor([0, 0, 1, 1]);
    this.mHero.getXform().setPosition(20, 60);
    this.mHero.getXform().setSize(2, 3);
}

```

Remember that the texture file path is used as the unique identifier in the ResourceMap. For this reason, it is essential for file texture loading and unloading and for the creation of TextureRenderable objects to refer to the same file path. In the previous code, all three functions refer to the same constants defined in the constructor.

5. The modification to the draw() function draws the two new TextureRenderable objects by calling their corresponding draw() functions, while the modification to the update() function is similar to that of the BlueLevel discussed earlier.
- Please refer to the MyGame.js source code file in the src/MyGame folder for details.

When running the example for this project in the Chapter5/5.1.TextureShaders folder, once again take note of the results of continuously changing the texture tinting—the blinking of the portal minion in red. In addition, notice the differences between the PNG-based textures in the MyGame level and the corresponding JPEG ones with white borders in the BlueLevel. It is visually more pleasing and accurate to represent objects using textures with the alpha (or transparency) channel. PNG is one of the most popular image formats supporting the alpha channel.

Observations

This project has been the longest and most complicated one that you have worked with. This is because working with texture mapping requires you to understand texture coordinates, the implementation cuts across many of the files in the engine, and the fact that actual images must be loaded, converted into textures, and stored/accessed from WebGL. To help summarize the changes, Figure 5-6 shows the game engine states in relation to the states of an image used for texture mapping and some of the main game 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 scene initialization, the `loadScene()` function will invoke the `gEngine_Texture.loadTexture()` function to load the image and process it with the `gEngine_Texture._processLoadedImage()` function into a corresponding WebGL texture and store it in a WebGL context. During the game loop cycles, the `TextureRenderable.draw()` function activates the appropriate WebGL texture via the `gEngine_Texture.activateTexture()` function. This enables the corresponding GLSL fragment shader sample from the correct texture during rendering. Finally, when a texture is no longer needed by the game engine, the `gEngineTexture_unloadTexture()` call will remove its existence from the system.

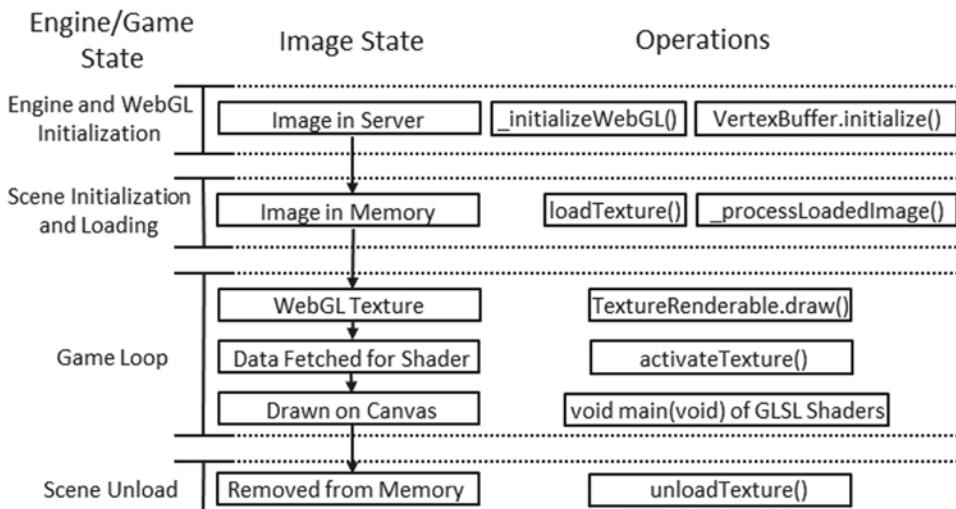


Figure 5-6. Overview of the states of an image file and the corresponding WebGL texture

Drawing with Sprite Sheets

As described earlier, a sprite sheet is an image that is composed of multiple lower-resolution images that individually represent different objects. Each of these individual images is referred to as a *sprite sheet element*. For example, Figure 5-7 shows a sprite sheet with 13 elements showing four different objects. Each of the top two rows contains five elements of the same object in different animated positions, and in the last row there are three elements of different objects: the character Dye, the portal minion, and the collector minion. The artist or software program that created the sprite sheet must communicate the pixel locations of each sprite element to the game developer, in much the same way as illustrated in Figure 5-7.

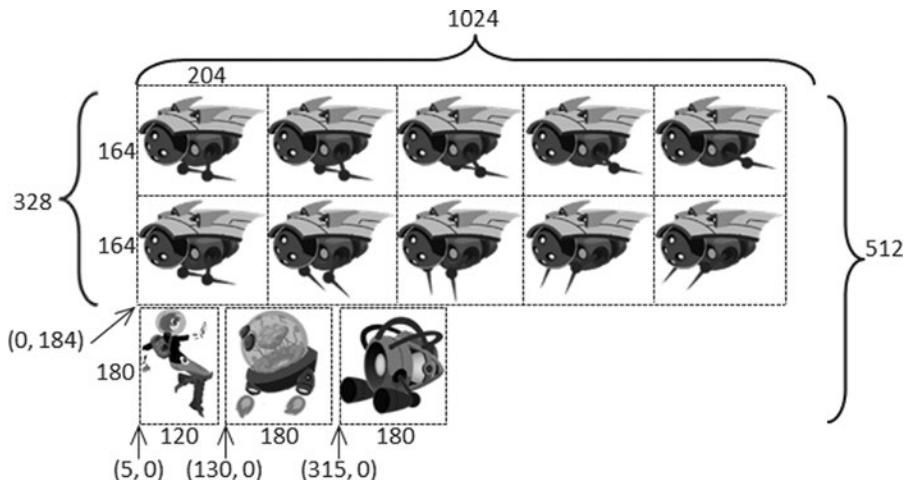


Figure 5-7. Example sprite sheet: *minion_sprite.png* composed of lower-resolution images of different objects

Sprite sheets are defined to optimize both memory and processing requirements. For example, recall that WebGL supports only textures that are defined by images with $2^x \times 2^y$ resolutions; this means a 256x128 ($2^8 \times 2^7$) image would be required in order to create a WebGL texture for the Dye character. In addition, if the 13 elements of Figure 5-7 were stored as separate images, 13 slow file system accesses would be required to load all the images, instead of one single system access to load the sprite sheet.

The key to working with a sprite sheet and the associated elements is to remember that the texture coordinate uv values are defined over the 0 to 1 normalized range 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 third row's rightmost element. 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).
- **UV values:** The lower-left corner is (0.308, 0.0), and the upper-right corner is (0.483, 0.352).
- **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.

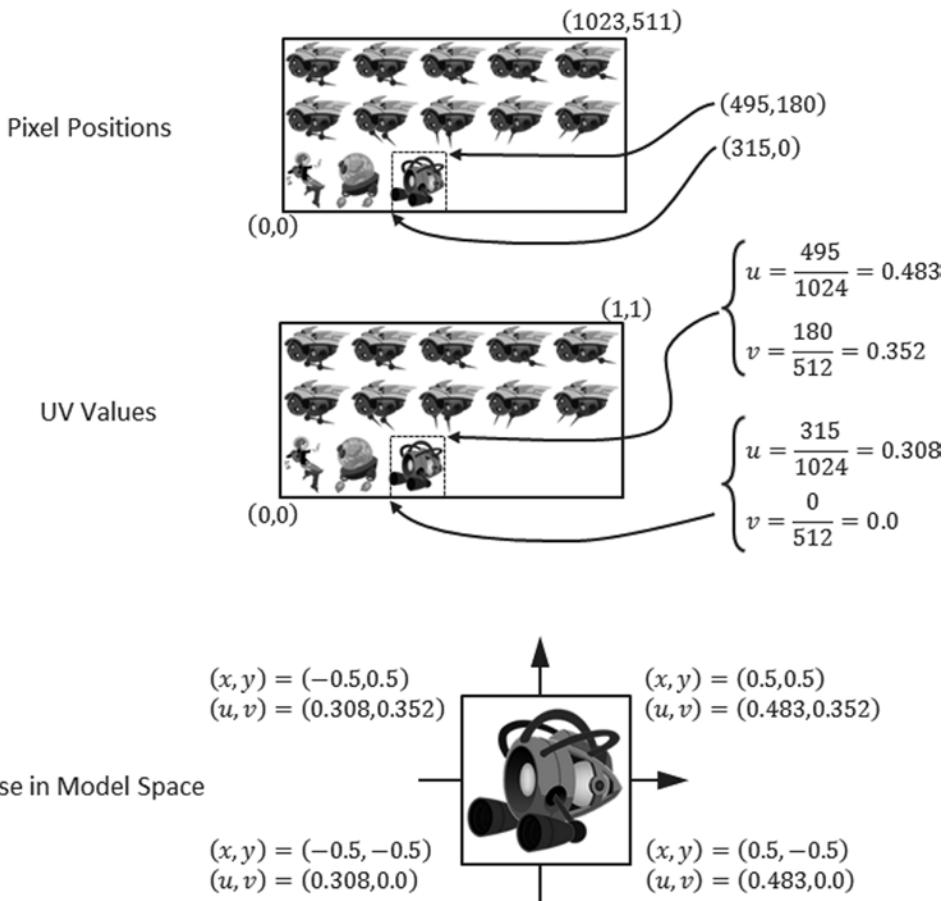


Figure 5-8. A conversion of coordinates from pixel position to uv values and used for mapping on geometry

The Sprite Shaders Project

This project demonstrates how to draw objects with sprite sheet elements by defining appropriate abstractions and classes. You can see an example of this project running in Figure 5-9. The source code to this project is defined in the Chapter5/5.2.SpriteShaders folder.



Figure 5-9. Running the *Sprite Shaders* project

The controls of the project are as follows:

- *Right arrow key*: Moves the Dye character (the hero) right and loops to the left boundary when the right boundary is reached
- *Left arrow key*: Moves the hero left and resets the position to the 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
- To experience defining subregions within an image for texture mapping
- To draw squares by mapping from sprite sheet elements
- To prepare for working with sprite animation and bitmap fonts

You can find the following external resource files in the assets folder: `Consolas-72.png` and `minion_sprite.png`. Notice that `minion_sprite.png` is the image shown in Figure 5-7.

As depicted in Figure 5-5, one of the main advantages and shortcomings of the texture support defined in the previous section is that the texture coordinate accessed via the `getGLTexCoordRef()` function is statically defined in the `Engine_VertexBuffer.js` file. This is an advantage because in those cases where

an entire image is mapped onto a square, all instances of `TextureShader` objects can share the same default uv values. This is also a shortcoming because the static texture coordinate buffer does not allow support for working with different subregions of an image. In other words, the static texture coordinate values do not allow the support for working with sprite sheet elements. As illustrated in Figure 5-10, the example from this section overcomes this shortcoming by defining per-object texture coordinates in the `SpriteShader` and `SpriteRenderable` objects. Notice that there are no new GLSL shaders defined since their functionality remains the same.

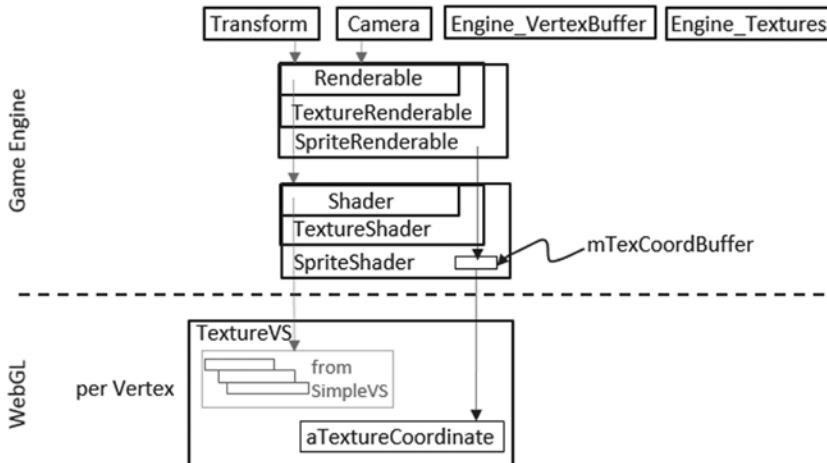


Figure 5-10. Defining a texture coordinate buffer in the `SpriteShader`

Interface GLSL Texture Shaders to the Engine with `SpriteShader`

Shaders supporting texture mapping with sprite sheet elements must be able to identify different unique subregions of an image. To support this functionality, you will implement the `SpriteShader` to define its own texture coordinates. Since this new shader extends the functionality of `TextureShader`, it is convenient to implement it as a subclass.

1. Create a new file in the `src/Engine/Shaders` folder and name it `SpriteShader.js`.
2. Define the `SpriteShader` constructor to derive it from `TextureShader`.

```

function SpriteShader(vertexShaderPath, fragmentShaderPath) {
    // Call super class constructor
    TextureShader.call(this, vertexShaderPath, fragmentShaderPath);

    this.mTexCoordBuffer = null; // gl buffer containing texture coordinate
    var initTexCoord = [
        1.0, 1.0,
        0.0, 1.0,
        1.0, 0.0,
        0.0, 0.0
    ];
}

```

```

var gl = gEngine.Core.getGL();
this.mTexCoordBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, this.mTexCoordBuffer);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(initTexCoord),
    gl.DYNAMIC_DRAW);
}
// get all the prototype functions from SimpleShader
gEngine.Core.inheritPrototype(SpriteShader, TextureShader);

```

SpriteShader defines its own texture coordinate buffer in WebGL, and the reference to this buffer is kept by `mTexCoordBuffer`. Notice that in the previous code when creating this buffer in the `gl.bufferData()` function, the `DYNAMIC_DRAW` option is specified. This is compared with the `STATIC_DRAW` option used in `Engine_VertexBuffer.js` when defining the system default texture coordinate buffer. In this case, the `dynamic` option informs the WebGL graphics system that the content to this buffer will be subject to changes.

3. Define a function to set the WebGL texture coordinate buffer.

```

SpriteShader.prototype.setTextureCoordinate = function(texCoord) {
    var gl = gEngine.Core.getGL();
    gl.bindBuffer(gl.ARRAY_BUFFER, this.mTexCoordBuffer);
    gl.bufferSubData(gl.ARRAY_BUFFER, 0, new Float32Array(texCoord));
};

```

Note that `texCoord` 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. `texCoord` must be a float array with eight floating-point numbers that identify four corners of a subregion in a Texture Space: 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 shader activation function to enable the custom texture coordinate buffer for rendering.

```

SpriteShader.prototype.activateShader = function(pixelColor, vpMatrix) {
    // first call the super class's activate
    SimpleShader.prototype.activateShader.call(this, pixelColor, vpMatrix);

    // now binds the proper texture coordinate buffer
    var gl = gEngine.Core.getGL();
    gl.bindBuffer(gl.ARRAY_BUFFER, this.mTexCoordBuffer);
    gl.vertexAttribPointer(this.mShaderTextureCoordAttribute, 2, gl.FLOAT,
        false, 0, 0);
    gl.enableVertexAttribArray(this.mShaderTextureCoordAttribute);
};

```

Notice that the superclass `activateShader()` function is called to `SimpleShader` instead of `TextureShader`. This is to avoid `TextureShader` activating the system default texture coordinate buffer for rendering.

SpriteRenderable Object

Similar to Renderable objects (which are shaded with SimpleShader) and TextureRenderable objects (which are shaded with TextureShader), a corresponding SpriteRenderable object should be defined to represent objects that will be shaded with SpriteShader.

1. Create a new file in the `src/Engine/Renderables` folder and name it `SpriteRenderable.js`.
2. Define the `SpriteRenderable` constructor to derive it from `TextureRenderable`.

```
function SpriteRenderable(myTexture) {
    TextureRenderable.call(this, myTexture);
    Renderable.prototype._setShader.call(this, gEngine.DefaultResources.
        getSpriteShader());

    this.mTexLeft = 0.0;    // bounds of texture coord (0 is left, 1 is right)
    this.mTexRight = 1.0;   //
    this.mTexTop = 1.0;    // 1 is top and 0 is bottom of image
    this.mTexBottom = 0.0; //
}
gEngine.Core.inheritPrototype(SpriteRenderable, TextureRenderable);
```

A `SpriteRenderable` object extends the `TextureRenderable` object by defining the set of variables that identifies the texture coordinate bounds of a subregion within the Texture Space, the bounds of a sprite sheet element.

3. Define an enumerated data type with values that identify corresponding offset positions of a WebGL texture coordinate specification array.

```
// the expected texture coordinate array is an array of 8 floats where:
// [0] [1]: is u/v coordinate of Top-Right
// [2] [3]: is u/v coordinate of Top-Left
// [4] [5]: is u/v coordinate of Bottom-Right
// [6] [7]: is u/v coordinate of Bottom-Left
SpriteRenderable.eTexCoordArray = Object.freeze({
    eLeft: 2,
    eRight: 0,
    eTop: 1,
    eBottom: 5
});
```

Note `eName` is an enumerated data type.

- Define functions to allow the specification of a sprite sheet element's uv values in both texture coordinate space (normalized between 0 to 1) and with pixel positions (which will be converted to uv values).

```
SpriteRenderable.prototype.setElementUVCoordinate = function(left, right,
bottom, top) {
    this.mTexLeft = left;
    this.mTexRight = right;
    this.mTexBottom = bottom;
    this.mTexTop = top;
};

SpriteRenderable.prototype.setElementPixelPositions = function(left, right,
bottom, top) {
    var texInfo = gEngine.ResourceMap.retrieveAsset(this.mTexture);

    // entire image width, height
    var imageW = texInfo.mWidth;
    var imageH = texInfo.mHeight;

    this.mTexLeft = left / imageW;
    this.mTexRight = right / imageW;
    this.mTexBottom = bottom / imageH;
    this.mTexTop = top / imageH;
};
```

Note that the `setElementPixelPositions()` function converts from pixel to texture coordinates before storing the results with the corresponding instance variables.

- Add a function to construct the texture coordinate specification array that is appropriate for passing the corresponding values to the WebGL context.

```
SpriteRenderable.prototype.getElementUVCoordinateArray = function() {
    return [
        this.mTexRight, this.mTexTop,           // x,y of top-right
        this.mTexLeft, this.mTexTop,
        this.mTexRight, this.mTexBottom,
        this.mTexLeft, this.mTexBottom
    ];
};
```

- Override the `draw()` function to load the specific texture coordinates values into WebGL context before the actual drawing.

```
SpriteRenderable.prototype.draw = function(pixelColor, vpMatrix) {
    // set the current texture coordinate
    this.mShader.setTextureCoordinate(this.getElementUVCoordinateArray());
    TextureRenderable.prototype.draw.call(this, pixelColor, vpMatrix);
};
```

SpriteShader as a Default Resource

Similar to SimpleShader and TextureShader, the SpriteShader is a resource that can be shared. Thus, it should be added to the engine's DefaultResources.

1. In the Engine_DefaultResources.js file, add a variable for storing a SpriteShader.

```
var mSpriteShader = null;
```

2. Define a getter function for the SpriteShader.

```
var getSpriteShader = function() { return mSpriteShader; };
```

3. Modify the _createShaders function to also create the SpriteShader.

```
var _createShaders = function(callBackFunction) {
    gEngine.ResourceMap.setLoadCompleteCallback(null);
    mConstColorShader = new SimpleShader(kSimpleVS, kSimpleFS);
    mTextureShader = new TextureShader(kTextureVS, kTextureFS);
    mSpriteShader = new SpriteShader(kTextureVS, kTextureFS);
    callBackFunction();
};
```

Notice that the SpriteShader actually wraps over the existing GLSL shaders defined in the TextureVS.glsL and TextureFS.glsL 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.

4. Remember to update the public interface.

```
var mPublic = {
    initialize: initialize,
    getConstColorShader: getConstColorShader,
    getTextureShader: getTextureShader,
    getSpriteShader: getSpriteShader
};
```

Testing the SpriteRenderable

There are two important functionalities of working with sprite elements and texture coordinates that should be tested: the proper extraction, drawing, and controlling of a sprite sheet element as an object; and the changing and controlling of uv coordinate on an object. For proper testing of the added functionality, you must modify the MyGame.js file.

1. The constructing, loading, unloading, and drawing of MyGame are similar to previous examples, so the details will not be repeated here. Please refer to the source code in the src/MyGame folder for details.

2. In the `initialize()` function, do the following:

```
MyGame.prototype.initialize = function() {
    // Step A: set up the cameras
    this.mCamera = new Camera(
        vec2.fromValues(20, 60),    // position of the camera
        20,                         // width of camera
        [20, 40, 600, 300]          // viewport (orgX, orgY, width, height)
    );
    this.mCamera.setBackgroundColor([0.8, 0.8, 0.8, 1]);
    // sets the background to gray

    // Step B: Create the support objects
    this.mPortal = new SpriteRenderable(this.kMinionSprite);
    this.mPortal.setColor([1, 0, 0, 0.2]); // tints red
    this.mPortal.getXform().setPosition(25, 60);
    this.mPortal.getXform().setSize(3, 3);
    this.mPortal.setElementPixelPositions(130, 310, 0, 180);

    this.mCollector = new SpriteRenderable(this.kMinionSprite);
    this.mCollector.setColor([0, 0, 0, 0]); // No tinting
    this.mCollector.getXform().setPosition(15, 60);
    this.mCollector.getXform().setSize(3, 3);
    this.mCollector.setElementPixelPositions(315, 495, 0, 180);

    // Step C: Create the font and minion images using sprite
    this.mFontImage = new SpriteRenderable(this.kFontImage);
    this.mFontImage.setColor([1, 1, 1, 0]);
    this.mFontImage.getXform().setPosition(13, 62);
    this.mFontImage.getXform().setSize(4, 4);

    this.mMinion= new SpriteRenderable(this.kMinionSprite);
    this.mMinion.setColor([1, 1, 1, 0]);
    this.mMinion.getXform().setPosition(26, 56);
    this.mMinion.getXform().setSize(5, 2.5);

    // Step D: Create the hero object with texture from lower-left corner
    this.mHero = new SpriteRenderable(this.kMinionSprite);
    this.mHero.setColor([1, 1, 1, 0]);
    this.mHero.getXform().setPosition(20, 60);
    this.mHero.getXform().setSize(2, 3);
    this.mHero.setElementPixelPositions(0, 120, 0, 180);
};
```

- a. After the camera is set up, in step B, notice that both `mPortal` and `mCollector` are created based on the same image, `kMinionSprite`, with the respective `setElementPixelPositions()` calls to specify the actual sprite element to use for rendering.

- b. Step C creates two additional `SpriteRenderable` objects: `mFontImage` and `mMinion`. The sprite element uv coordinate settings are left to the defaults where the texture image will cover the entire geometry.
- c. Similar to step B, step C creates the hero character as a `SpriteRenderable` object based on the same `kMinionSprite` image. The sprite sheet element that corresponds to the hero is identified with the `setElementPixelPositions()` call.

Notice that in this example, four of the five `SpriteRenderable` objects created are based on the same `kMinionSprite` image.

3. The `update()` function is modified to support the controlling of the hero object and changes to the uv values.

```
MyGame.prototype.update = function() {
    // let's only allow the movement of hero,
    // and if hero moves too far off, this level ends, we will
    // load the next level
    var deltaX = 0.05;
    var xform = this.mHero.getXform();

    // Support hero movements
    if (gEngine.Input.isKeyPressed(gEngine.Input.keys.Right)) {
        xform.incXPosBy(deltaX);
        if (xform.getXPos() > 30) // this is the right-bound of the window
            xform.setPosition(12, 60);
    }

    if (gEngine.Input.isKeyPressed(gEngine.Input.keys.Left)) {
        xform.incXPosBy(-deltaX);
        if (xform.getXPos() < 11) { // this is the left-bound of the window
            xform.setPosition(20, 60);
        }
    }

    // continuously change texture tinting
    var c = this.mPortal.getColor();
    var ca = c[3] + deltaX;
    if (ca > 1) ca = 0;
    c[3] = ca;

    // New update code for changing the sub-texture regions being shown
    var deltaT = 0.001;

    // The font image:
    // zoom into the texture by updating texture coordinate
    // For font: zoom to the upper left corner by changing bottom right
    var texCoord = this.mFontImage.getElementUVCoordinateArray();
        // The 8 elements:
        //     mTexRight, mTexTop,           // x,y of top-right
        //     mTexLeft,   mTexTop,
```

```

        //      mTexRight,  mTexBottom,
        //      mTexLeft,   mTexBottom
var b = texCoord[SpriteRenderable.eTexCoordArray.eBottom] + deltaT;
var r = texCoord[SpriteRenderable.eTexCoordArray.eRight] - deltaT;

if (b > 1.0) b = 0;
if (r < 0) r = 1.0;
this.mFontImage.setElementUVCoordinate(
    texCoord[SpriteRenderable.eTexCoordArray.eLeft], r,
    b, texCoord[SpriteRenderable.eTexCoordArray.eTop]);

// The minion image:
// For minion: zoom to the bottom right corner by changing top left
var texCoord = this.mMinion.getElementUVCoordinateArray();
        // The 8 elements:
        //      mTexRight,  mTexTop,           // x,y of top-right
        //      mTexLeft,   mTexTop,
        //      mTexRight,  mTexBottom,
        //      mTexLeft,   mTexBottom
var t = texCoord[SpriteRenderable.eTexCoordArray.eTop] - deltaT;
var l = texCoord[SpriteRenderable.eTexCoordArray.eLeft] + deltaT;

if (l > 0.5) l = 0;
if (t < 0.5) t = 1.0;

this.mMinion.setElementUVCoordinate(
    l, texCoord[SpriteRenderable.eTexCoordArray.eRight],
    texCoord[SpriteRenderable.eTexCoordArray.eBottom], t);
};

```

- Observe that the keyboard control and the drawing of the hero object are identical to previous projects.
- Notice the calls to `setElementUVCoordinate()` for `mFontImage` and `mMinion`. These calls continuously decrease and reset the V values that correspond to the bottom, the U values that correspond to the right for `mFontImage`, the V values that correspond to the top, and the U values that correspond to the left for `mMinion`. The end results are the continuous changing of texture and the appearance of a 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 and 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 separate rows; however, this is not always the case. 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

Figure 5-12 shows that to achieve the animated effect of an object retracting its spikes toward the right, as depicted by the top row of Figure 5-11, you map the elements from the left to the right in the sequence 1, 2, 3, 4, 5. When these images are mapped onto the same geometry, sequenced, and looped in an appropriate rate, it conveys the sense that the object is indeed repeating the action of retracting its spikes. Alternatively, if the sequence is reversed where the elements are mapped in the right-to-left sequence, it would create the animation that corresponds to the object extending the spikes toward the left. Last, it is also possible to map the sequence in a swing loop from left to right and then back from right to left. In this case, the animation would correspond to the object going through the motion of retracting and extending its spikes continuously.



Figure 5-12. A sprite animation sequence that loops

With a firm background in sprite sheet animation and understanding the different ways of generating sprite animation, you can now tackle the implementation.

The Sprite Animation Project

This project demonstrates how to work with animated sprite sheet and generate continuous sprite animations. You can see an example of this project running in Figure 5-13. The project scene contains the objects from the previous scene plus two animated objects. The source code to this project is defined in the [Chapter5/5.3.SpriteAnimation](#) folder.



Figure 5-13. Running the Sprite Animation 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
- *Left arrow key:* Opposite of the right arrow key
- *Number 1 key:* Animates by showing sprite elements continuously from right to left
- *Number 2 key:* Animates by showing sprite elements moving back and forth continuously from left to right and right to left
- *Number 3 key:* Animates by showing sprite elements continuously from left to right
- *Number 4 key:* Increases the animation speed
- *Number 5 key:* Decreases the animation speed

The goals of the project are as follows:

- To gain a deeper understanding of animated sprite sheets
- To experience the creation of sprite animations
- To define abstractions for implementing sprite animations

You can find the same files as in the previous project in the assets folder.

SpriteAnimateRenderable Object

Sprite animation can be implemented by strategically controlling the uv values of a `SpriteRenderable` to display the appropriate sprite element at desired time periods. For this reason, only a single class, `SpriteAnimateRenderable`, needs to be defined to support sprite animations.

For simplicity and ease of understanding, the following implementation assumes that all sprite elements associated with an animation are always organized along the same row. Animated sprite elements organized along a column are not supported. For example, in Figure 5-11, the rightward retraction and leftward extension movements of the spikes are each organized along a row; neither spans more than one single row, and neither is organized along a column.

1. Create a new file in the `src/Engine/Renderables` folder and name it `SpriteAnimateRenderable.js`.
2. Define an enumerated data type that describes the different ways to animate with a sprite sheet.

```
// Assumption: first sprite in an animation is always the left-most element.
SpriteAnimateRenderable.eAnimationType = Object.freeze({
    eAnimateRight: 0, // Animate from left to right, then restart to left
    eAnimateLeft: 1, // Animate from right to left, then restart to right
    eAnimateSwing: 2 // Animate first left to right, then animates backwards
});
```

`eAnimationType` defines three modes for animation.

- a. `eAnimateRight` 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. `eAnimateLeft` is the reverse of `eAnimateRight`; it starts from the right, animates toward the left, and continues by starting from the rightmost element after reaching the leftmost element.
- c. `eAnimateSwing` is a continuous loop from left to right and then from right to left.
3. Define the `SpriteAnimateRenderable` constructor to derive it from `SpriteRenderable`.

```
function SpriteAnimateRenderable(myTexture) {
    SpriteRenderable.call(this, myTexture);
    Renderable.prototype._setShader.call(this, gEngine.DefaultResources.
        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; // 1.0 is top corner of image
this.mElmWidth = 1.0; // default sprite element size is the entire image
this.mElmHeight = 1.0;
this.mWidthPadding = 0.0;
this.mNumElems = 1; // number of elements in an animation

// per animation settings
this.mAnimationType = SpriteAnimateRenderable.eAnimationType.eAnimateRight;
this.mUpdateInterval = 1; // how often to advance

// current animation state
this.mCurrentAnimAdvance = -1;
this.mCurrentElm = 0;

this._initAnimation();
}
gEngine.Core.inheritPrototype(SpriteAnimateRenderable, SpriteRenderable);

```

The `SpriteAnimateRenderable` constructor defines three sets of variables:

- The first set, including `mFirstElmLeft`, `mElmTop`, and so on, defines the location and dimensions of each sprite element and the number of elements in the animation. This information can be used to accurately compute the texture coordinates for each sprite element when the elements are ordered by rows and columns. Note that all coordinates are in Texture Space (0 to 1).
- The second set stores information on how to animate: the `mAnimationType` of left, right, or swing; and how many `mUpdateInterval` time to wait before advancing to the next sprite element to control the speed of the animation. This information can be changed during runtime to reverse a character's movement, loop the character's movement, or speed up or slow down the movement.
- The third set, `mCurrentAnimAdvance` and `mCurrentElm`, describes the current animation state, which frame, and the direction of the animation. Both of these variables are in units of element counts, are not accessible by the game programmer, and are used internally to compute the next sprite element for display.

The `_initAnimation()` function computes the values of `mCurrentAnimAdvance` and `mCurrentElm` to initialize an animation sequence.

- Define a function to set the animation type.

```

SpriteAnimateRenderable.prototype.setAnimationType = function(animationType) {
    this.mAnimationType = animationType;
    this.mCurrentAnimAdvance = -1;
    this.mCurrentElm = 0;
    this._initAnimation();
};

```

Note that the animation is always reset to start from the beginning when the animation type (left, right, or swing) is changed.

5. Define the `_initAnimation()` function to compute the proper values for `mCurrentAnimAdvance` and `mCurrentElm` according to the current animation type.

```
SpriteAnimateRenderable.prototype._initAnimation = function() {
    // Currently running animation
    this.mCurrentTick = 0;
    switch (this.mAnimationType) {

        case SpriteAnimateRenderable.eAnimationType.eAnimateRight:
            this.mCurrentElm = 0;
            this.mCurrentAnimAdvance = 1; // either 1 or -1
            break;

        case SpriteAnimateRenderable.eAnimationType.eAnimateSwing:
            this.mCurrentAnimAdvance = -1 * this.mCurrentAnimAdvance;
            this.mCurrentElm += 2*this.mCurrentAnimAdvance;
            break;

        case SpriteAnimateRenderable.eAnimationType.eAnimateLeft:
            this.mCurrentElm = this.mNumElems - 1;
            this.mCurrentAnimAdvance = -1; // either 1 or -1
            break;
    }
    this._setSpriteElement();
};
```

The previous code shows that `mCurrentElm` is the number of elements offset from the leftmost element, and `mCurrentAnimAdvance` records whether the `mCurrentElm` offset should be incremented (for rightward animation) or decremented (for leftward animation) during each update.

The `_setSpriteElement()` is called to set the uv values that correspond to the currently identified sprite element for displaying on the geometry (by the superclass, `SpriteRenderable`).

6. Define the `_setSpriteElement()` function to compute and load the uv values of the currently identified sprite element for rendering.

```
SpriteAnimateRenderable.prototype._setSpriteElement = function() {
    var left = this.mFirstElmLeft + (this.mCurrentElm * (this.mElmWidth +
        this.mWidthPadding));

    SpriteRenderable.prototype.setElementUVCoordinate.call(this, left,
        left+this.mElmWidth,
        this.mElmTop-this.mElmHeight,
        this.mElmTop);
};
```

The variable `left` is the left u value of `mCurrentElm`. Based on this value, the right, bottom, and top uv values can be derived and set to `SpriteRenderable`.

7. Define a function to allow game programmers to specify a sprite animation.

```
// Always set the right-most element to be the first
SpriteAnimateRenderable.prototype.setSpriteSequence = function(
    topPixel,           // offset from top-left
    rightPixel,         // offset from top-left
    elmWidthInPixel,
    elmHeightInPixel,
    numElements,        // number of elements in sequence
    wPaddingInPixel    // left/right padding
)
{
    var texInfo = gEngine.ResourceMap.retrieveAsset(this.mTexture);
    // entire image width, height
    var imageW = texInfo.mWidth;
    var imageH = texInfo.mHeight;

    this.mNumElems = numElements; // number of elements in animation
    this.mFirstElmLeft = rightPixel / imageW;
    this.mElmTop = topPixel / imageH;
    this.mElmWidth = elmWidthInPixel / imageW;
    this.mElmHeight = elmHeightInPixel / imageH;
    this.mWidthPadding = wPaddingInPixel / imageW;
    this._initAnimation();
};
```

The inputs of the `setSpriteSequence()` function are in pixels and are converted to texture coordinates by dividing by the width and height of the image.

8. Implement functions to change animation speed, either directly or by an offset.

```
SpriteAnimateRenderable.prototype.setAnimationSpeed = function(tickInterval) {
    // number of update calls before advancing animation
    this.mUpdateInterval = tickInterval; // how often to advance
};

SpriteAnimateRenderable.prototype.incAnimationSpeed = function(deltaInterval) {
    // number of update calls before advancing animation
    this.mUpdateInterval += deltaInterval; // how often to advance
};
```

9. Finally, define a function to advance the animation for each game loop update.

```
SpriteAnimateRenderable.prototype.updateAnimation = function() {
    this.mCurrentTick++;
    if (this.mCurrentTick >= this.mUpdateInterval) {
        this.mCurrentTick = 0;
        this.mCurrentElm += this.mCurrentAnimAdvance;
        if ((this.mCurrentElm >= 0) && (this.mCurrentElm < this.mNumElems))
            this._setSpriteElement();
```

```

        else
            this._initAnimation();
    }
};

```

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 `GameLoop _runLoop()` function ensures systemwide updates to occur at kMPF intervals even when frame rate lags. The game engine architecture ensures the `updateAnimation()` function calls are kMPF millisecond apart.

Testing Sprite Animation

The test cases for the `SpriteAnimateRenderable` object must demonstrate how the forward, reverse, and swing modes of animation and the animation speed are under the programmer's control. The `MyGame` object is modified to accomplish these purposes.

1. The constructing, loading, unloading, and drawing of `MyGame` are similar to the previous example, so the details will not be repeated here. Please refer to the source code in the `src/MyGame` folder for details.
2. In the `initialize()` function, add code to create and initialize the `SpriteAnimateRenderable` objects.

```

MyGame.prototype.initialize = function() {
    // ... Identical to previous code ...

    // The right minion
    this.mRightMinion= new 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 position: top-right, 512 is top, 0 is right
        204,164,     // width x height in pixels
        5,           // number of elements in this sequence
        0);          // horizontal padding in between
    this.mRightMinion.setAnimationType(SpriteAnimateRenderable.eAnimationType.
        eAnimateRight);
    this.mRightMinion.setAnimationSpeed(50);

    // the left minion
    this.mLeftMinion= new 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: top-right, 164 from 512 is top, 0 is right
    204,164,     // widthxheight in pixels
    5,           // number of elements in this sequence
    0);          // horizontal padding in between
this.mLeftMinion.setAnimationType(SpriteAnimateRenderable.eAnimationType.
    eAnimateRight);
this.mLeftMinion.setAnimationSpeed(50);

// ... Identical to previous code ...
}

```

The `SpriteAnimateRenderable` objects are created in similar ways as `SpriteRenderable` objects with a sprite sheet as the texture parameter. Additionally, in this case, it is essential to call the `setSpriteSequence()` function to identify the elements involved in the animation including the location, dimension, and total number of elements.

3. The `update()` function must invoke the `SpriteAnimateRenderable` object's `updateAnimation()` function to advance the sprite animation.

```

MyGame.prototype.update = function() {
    // ... Identical to previous code ...

    // remember to update the minion's animation
    this.mRightMinion.updateAnimation();
    this.mLeftMinion.updateAnimation();

    // Animate left on the sprite sheet
    if (gEngine.Input.isKeyClicked(gEngine.Input.keys.One)) {
        this.mRightMinion.setAnimationType(
            SpriteAnimateRenderable.eAnimationType.eAnimateLeft);
        this.mLeftMinion.setAnimationType(
            SpriteAnimateRenderable.eAnimationType.eAnimateLeft);
    }

    // swing animation
    if (gEngine.Input.isKeyClicked(gEngine.Input.keys.Two)) {
        this.mRightMinion.setAnimationType(
            SpriteAnimateRenderable.eAnimationType.eAnimateSwing);
        this.mLeftMinion.setAnimationType(
            SpriteAnimateRenderable.eAnimationType.eAnimateSwing);
    }

    // Animate right on the sprite sheet
    if (gEngine.Input.isKeyClicked(gEngine.Input.keys.Three)) {
        this.mRightMinion.setAnimationType(
            SpriteAnimateRenderable.eAnimationType.eAnimateRight);
        this.mLeftMinion.setAnimationType(
            SpriteAnimateRenderable.eAnimationType.eAnimateRight);
    }
}

```

```

// decrease the duration of showing sprite elements,
// speeding up the animation
if (gEngine.Input.isKeyClicked(gEngine.Input.keys.Four)) {
    this.mRightMinion.incAnimationSpeed(-2);
    this.mLeftMinion.incAnimationSpeed(-2);
}

// increase the duration of showing sprite elements,
// slowing down the animation
if (gEngine.Input.isKeyClicked(gEngine.Input.keys.Five)) {
    this.mRightMinion.incAnimationSpeed(2);
    this.mLeftMinion.incAnimationSpeed(2);
}
};

```

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" 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" />
  </chars>
</font>
```

Figure 5-15. A snippet of the XML file with the decoding information for the bitmap font image shown in Figure 5-14

Notice that the decoding information as shown in Figure 5-15 uniquely defines the uv coordinate positions for each character in the image, as shown in Figure 5-14. In this way, the texture mapping of individual characters from a bitmap font sprite image can be performed in a straightforward manner by the `SpriteRenderable` objects.

Note There are many bitmap font file formats. The format used in this book is the AngleCode BMFont-compatible font in XML form. BMFont is an open source software that converts vector fonts, such as TrueType and OpenType, into bitmap fonts. See <http://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. This project consists of two scenes. The first is a simple extension from the previous project with sample text output from different bitmap fonts. You can control the hero's position with the arrow keys. If the hero exits the window from the left boundary, the second scene will be invoked. The second scene simply shows the "Game Over!" message and stops the game loop. The source code to this project is defined in the Chapter5/5.4.FontSupport folder.



Figure 5-16. Running the Font Support project

The controls of the project are as follows:

- *Number keys 0, 1, 2, and 3*: Selects 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 arrow key*: Moves the hero left; if the hero exits the screen from the left boundary, the GameOver level is invoked and the game ends
- *Right arrow key*: Moves the hero right; when crossing the right boundary, the hero is wrapped back to the left boundary

The goals of the project are as follows:

- To gain a basic understanding of drawing text strings in a game
- To understand what bitmap fonts are
- To implement text drawing support in your game engine

You can find the following external resource files in the assets folder: Consolas-72.png and minion_sprite.png. In the assets/fonts folder are the bitmap font sprite image files and the associated XML files that contain the decoding information: Consolas-16.fnt, Consolas-16.png, Consolas-24.fnt, Consolas-24.png, Consolas-32.fnt, Consolas-32.png, Consolas-72.fnt, Consolas-72.png, Segment7-96.fnt, Segment7-96.png, system-default-font.fnt, and system-default-font.png.

Notice that the .fnt and .png files are paired. The former contains decoding information for the latter. These file pairs must be included in the same folder for the engine to load the font properly. system-default-font is the default font for the game engine, and it is assumed that this font is always present in the asset/fonts folder.

Note The actions of parsing, decoding, and extracting of character information from the .fnt files are independent from the foundational operations of a game engine. For this reason, the details of these operations are not presented. If you are interested, you should consult the source code.

Loading and Storing Fonts in the Engine

Loading font files is special because fonts are defined in pairs: the .fnt file that contains decoding information and the corresponding .png sprite image file. However, since the .fnt file is an XML file and the .png file is a simple texture image, the actual loading of these two files is already supported by the existing engine functionality. The details of loading and storing fonts in the engine will be hidden by a new engine component.

1. Create a new file in the src/Engine/Resources folder and name it Engine_Fonts.js.
2. Before implementing the gEngine_Fonts component, first define an object for storing pixel location and display information associated with the characters.

```
function CharacterInfo() {
    // in texture coordinate (0 to 1) maps to the entire image
    this.mTexCoordLeft = 0;
    this.mTexCoordRight = 1;
    this.mTexCoordBottom = 0;
    this.mTexCoordTop = 0;
```

```

// nominal character 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 ration
this.mCharAspectRatio = 1;
}

```

The uv coordinate and character appearance information can be computed based on the contents from the .fnt file.

3. Following the pattern of previous engine components, implement the Fonts engine component.

```

var gEngine = gEngine || { };

gEngine.Fonts = (function() {
    var mPublic = { };
    return mPublic;
}());

```

4. Define a function to load font files from a given path.

```

var loadFont = function(fontName) {
    if (!(gEngine.ResourceMap.isAssetLoaded(fontName))) {
        var fontInfoSourceString = fontName + ".fnt";
        var textureSourceString = fontName + ".png";

        // register an entry in the map
        gEngine.ResourceMap.asyncLoadRequested(fontName);

        gEngine.Textures.loadTexture(textureSourceString);
        gEngine.TextFileLoader.loadTextFile(fontInfoSourceString,
            gEngine.TextFileLoader.eTextFileType.eXMLFile,
            _storeLoadedFont);
    } else {
        gEngine.ResourceMap.incAssetRefCount(fontName);
    }
};

```

- a. `fontName` is a path to the font files but without any file extensions. For example, `assets/fonts/system-default-font` is the string that identifies the two associated `.fnt` and `.png` files.
- b. Two file load operations are actually invoked: one to load the `.fnt` as a text file and the second to load the `.png` as a texture image file.
- c. The last parameter of the `loadTextFile()` function specifies that the `_storeLoadedFont()` function should be invoked when the load operation is completed.

5. Define the `_storeLoadedFont()` function to store the loaded results in the `ResourceMap`.

```
var _storeLoadedFont = function(fontInfoSourceString) {
    var fontName = fontInfoSourceString.slice(0, -4); // trims .fnt extension
    var fontInfo = gEngine.ResourceMap.retrieveAsset(fontInfoSourceString);
    fontInfo.FontImage = fontName + ".png";
    gEngine.ResourceMap.asyncLoadCompleted(fontName, fontInfo);
};
```

Recall that the `ResourceMap.retrieveAsset()` function returns an `MapEntry` object that contains the reference to the loaded asset, the XML file content in this case, and a reference count. The previous code creates an additional property, `FontImage`, on the returned `MapEntry` to record the associated sprite image for the font.

6. Define a function to unload a font and release the associated memory.

```
var unloadFont = function(fontName) {
    gEngine.ResourceMap.unloadAsset(fontName);
    if (!(gEngine.ResourceMap.isAssetLoaded(fontName))) {
        var fontInfoSourceString = fontName + ".fnt";
        var textureSourceString = fontName + ".png";

        gEngine.Textures.unloadTexture(textureSourceString);
        gEngine.TextFileLoader.unloadTextFile(fontInfoSourceString);
    };
};
```

It is important to remember to unload both the text `.fnt` file and the `.png` texture image file.

7. Define a function to compute `CharacterInfo` based on the information presented in the `.fnt` file.

```
var getCharInfo = function(fontName, aChar) {
    // ... details omitted for lack of relevancy

    returnInfo = new CharacterInfo();

    // computes and fills in the contents of CharacterInfo
    // ... details omitted for lack of relevancy

    return returnInfo;
};
```

Details of decoding and extracting information for the given character are omitted because they are unrelated to the rest of the game engine implementation.

Note For details of the `.fnt` format information, please refer to http://www.angelcode.com/products/bmfont/doc/file_format.html.

8. Finally, remember to add the public functions to the public interface.

```
var mPublic = {
    loadFont: loadFont,
    unloadFont: unloadFont,
    getCharInfo: getCharInfo
};
return mPublic;
```

Defining a FontRenderable Object to Draw Texts

The defined gEngine_Fonts component is capable of loading font files and extracting per-character uv coordinate and appearance information. With this functionality, the drawing of a text string can be accomplished by identifying each character in the string, extracting the corresponding texture mapping information from the gEngine_Fonts component, and rendering the character using the 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 `FontRenderable.js`.
2. Implement a constructor for `FontRenderable` that accepts a string as its parameter.

```
function FontRenderable(aString) {
    this.mFont = gEngine.DefaultResources.getDefaultFont();
    this.mOneChar = new SpriteRenderable(this.mFont + ".png");
    this.mXform = new Transform(); // transform that moves this object around
    this.mText = aString;
}
```

- a. The `aString` variable is the message to be drawn.
- 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.
3. Define the `draw()` function to parse and draw each character in the string using the `mOneChar` variable.

```
FontRenderable.prototype.draw = function(vpMatrix) {
    var widthOfOneChar = this.mXform.getWidth() / this.mText.length;
    var heightOfOneChar = this.mXform.getHeight();
    var yPos = this.mXform.getYPos();

    // center position of the first char
    var xPos = this.mXform.getXPos() - (widthOfOneChar / 2) +
    (widthOfOneChar * 0.5);
    var charIndex, aChar, charInfo, xSize, ySize, xOffset, yOffset;
    for (charIndex = 0; charIndex < this.mText.length; charIndex++) {
        aChar = this.mText.charCodeAt(charIndex);
        charInfo = gEngine.Fonts.getCharInfo(this.mFont, aChar);
```

```

        // set the texture coordinate
        this.mOneChar.setElementUVCoordinate(charInfo.mTexCoordLeft,
            charInfo.mTexCoordRight,
            charInfo.mTexCoordBottom, charInfo.mTexCoordTop);

        // now the size of the char
        xSize = widthOfOneChar * charInfo.mCharWidth;
        ySize = heightOfOneChar * charInfo.mCharHeight;
        this.mOneChar.getXform().setSize(xSize, ySize);

        // how much to offset from the center
        xOffset = widthOfOneChar * charInfo.mCharWidthOffset * 0.5;
        yOffset = heightOfOneChar * charInfo.mCharHeightOffset * 0.5;

        this.mOneChar.getXform().setPosition(xPos - xOffset, yPos - yOffset);

        this.mOneChar.draw(vpMatrix);

        xPos += widthOfOneChar;
    }
};


```

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 does the following:

- Extracts each character in the string
 - Calls the `getCharInfo()` function to receive the character's uv values and appearance information in `charInfo`
 - 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)
 - Uses the appearance 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
4. Implement the getters and setters for the transform, the text message to be drawn, the font to use for drawing, and the color.

```

FontRenderable.prototype.getXform = function() { return this.mXform; };
FontRenderable.prototype.getText = function() { return this.mText; };
FontRenderable.prototype.setText = function(t) {
    this.mText = t;
    this.setTextHeight(this.getXform().getHeight());
};

FontRenderable.prototype.getFont = function() { return this.mFont; };
FontRenderable.prototype.setFont = function(f) {
    this.mFont = f;
    this.mOneChar.setTexture(this.mFont + ".png");
};

```

```
FontRenderable.prototype.setColor = function(c){ this.mOneChar.setColor(c); };
FontRenderable.prototype.getColor = function() {
    return this.mOneChar.getColor();
};
```

5. Define the `setTextHeight()` function to define the height of the message to be output.

```
FontRenderable.prototype.setTextHeight = function(h) {
    // this is for "A"
    var charInfo = gEngine.Fnts.getCharInfo(this.mFont, "A".charCodeAt(0));
    var w = h * charInfo.mCharAspectRatio;
    this.getXform().setSize(w * this.mText.length, h);
};
```

Notice that the width of the entire message to be drawn is automatically computed based on the message string length and maintaining the character width to height aspect ratio.

Note `FontRenderable` does not support the rotation of the entire message. Text messages are always drawn horizontally from left to right.

Adding a Default Font to the Engine

A default system font should be provided by the game engine for the convenience of the game programmer. This can be accomplished with simple modifications to the `gEngine_DefaultResources` component.

1. Edit the `Engine_DefaultResources.js` file to define the default system font name and to define a getter function to retrieve it.

```
// Default font
var kDefaultFont = "assets/fonts/system-default-font";
var getDefaultFont = function() { return kDefaultFont; };
```

2. Modify the `initialize()` function to also load the default font.

```
var initialize = function(callBackFunction) {
    // ... Identical to previous code ...

    // load default font
    gEngine.Fnts.loadFont(kDefaultFont);

    // ... Identical to previous code ...
};
```

After Game Cleanup

In all the examples, the WebGL resource allocation and cleanup have been carefully handled during scene transitions where loaded external resources are always unloaded. However, when the entire game is ready to terminate, there is no current support to release the allocated shared resources. In the game engine, there are two components that allocated WebGL resources for sharing: `gEngine_VertexBuffer` for sharing unit square vertex positions and static texture coordinate buffers, and `gEngine_DefaultResources` for the allocated shared resources.

The following files must be modified to support proper cleanup at the end of a game:

1. In the `Engine_Core.js` file, add the following function to instruct `VertexBuffer` and `DefaultResources` to release their allocated resources, and remember to add the `cleanup()` function to the public interface:

```
var cleanUp = function() {
    gEngine.VertexBuffer.cleanUp();
    gEngine.DefaultResources.cleanUp();
};

var mPublic = {
    // ... Identical to previous code ...
    cleanUp: cleanUp
};
```

2. In the `Engine_DefaultResources.js` file, add the following function to release all default resources, and remember to add the `cleanup()` function to the public interface so that this function is accessible:

```
var cleanUp = function() {
    mConstColorShader.cleanUp();
    mTextureShader.cleanUp();
    mSpriteShader.cleanUp();

    gEngine.TextFileLoader.unloadTextFile(kSimpleVS);
    gEngine.TextFileLoader.unloadTextFile(kSimpleFS);

    // texture shader:
    gEngine.TextFileLoader.unloadTextFile(kTextureVS);
    gEngine.TextFileLoader.unloadTextFile(kTextureFS);

    // default font
    gEngine.FONTS.unloadFont(kDefaultFont);
};
```

3. In the `Engine_VertexBuffer.js` file, add the following function to release the allocated WebGL buffers, and remember to add the `cleanup()` function to the public interface so that this function is accessible:

```
var cleanUp = function() {
    var gl = gEngine.Core.getGL();
    gl.deleteBuffer(mSquareVertexBuffer);
    gl.deleteBuffer(mTextureCoordBuffer);
};
```

4. In the `SpriteShader.js` file, add the following function to release the allocated texture coordinate buffer:

```
SpriteShader.prototype.cleanUp = function() {
    var gl = gEngine.Core.getGL();
    gl.deleteBuffer(this.mTexCoordBuffer);

    // now call super class's clean up ...
    SimpleShader.prototype.cleanUp.call(this);
};
```

5. In the `SimpleShader.js` file, the WebGL memory allocated for vertex shader, fragment shader, and compiled GLSL shader must all be released. To accomplish this, instance variables must be defined to refer to the corresponding shaders.

```
function SimpleShader(vertexShaderPath, fragmentShaderPath) {

    // ... Identical to previous code ...

    // Step A: load and compile vertex and fragment shaders
    this.mVertexShader = this._CompileShader(vertexShaderPath,
                                              gl.VERTEX_SHADER);
    this.mFragmentShader = this._CompileShader(fragmentShaderPath,
                                                gl.FRAGMENT_SHADER);

    // Step B: Create and link the shaders into a program.
    this.mCompiledShader = gl.createProgram();
    gl.attachShader(this.mCompiledShader, this.mVertexShader);
    gl.attachShader(this.mCompiledShader, this.mFragmentShader);
    gl.linkProgram(this.mCompiledShader);

    // Step C, D, E, F, G:
    // ... Identical to previous code ...
}

SimpleShader.prototype.cleanUp = function() {
    var gl = gEngine.Core.getGL();
    gl.detachShader(this.mCompiledShader, this.mVertexShader);
    gl.detachShader(this.mCompiledShader, this.mFragmentShader);
    gl.deleteShader(this.mVertexShader);
    gl.deleteShader(this.mFragmentShader);
};
```

- a. The constructor is modified to reference the vertex and fragment shaders with the `mVertexShader` and `mFragmentShader` variables.
- b. The new `cleanUp()` function releases the memory associated with the shaders.

Testing Fonts

You are now ready to test the system font support for drawing text messages and to actually end the game by stopping the game loop. The first step is to define the GameOver scene.

The GameOver Scene

The goal of this scene is to output a “Game Over” message for the player and end the game gracefully. The scene will set up a proper WC system, draw the “Game Over” text, and stop the game loop.

1. Create a new file in the `src/Engine/MyGame` folder and name it `GameOver.js`.
2. Implement the constructor, with the `mCamera` to set up a convenient World Coordinate (WC) system and the `mMsg` to contain the “Game Over” message.

```
function GameOver() {
    this.mCamera = null;
    this.mMsg = null;
};

gEngine.Core.inheritPrototype(GameOver, Scene);

3. In the initialize() function, define a convenient WC and initialize the
mMsg object.

GameOver.prototype.initialize = function() {
    // Step A: set up the cameras
    this.mCamera = new Camera(
        vec2.fromValues(50, 33), // position of the camera
        100, // width of camera
        [0, 0, 600, 400] // viewport (orgX, orgY, width, height)
    );
    this.mCamera.setBackgroundColor([0.9, 0.9, 0.9, 1]);

    this.mMsg = new FontRenderable("Game Over!");
    this.mMsg.setColor([0, 0, 0, 1]);
    this.mMsg.getXform().setPosition(22, 32);
    this.mMsg.setTextHeight(10);
};
```

The `mMsg` is instantiated as a `FontRenderable` object with “Game Over” as the default message. Notice that only the height of the text is set, `setTextHeight()`; the width of the entire message is computed based on preserving the aspect ratio of the font.

4. Draw the text in the same manner as all Renderable objects.

```
GameOver.prototype.draw = function() {
    // Step A: clear the canvas
    gEngine.Core.clearCanvas([0.9, 0.9, 0.9, 1.0]); // clear to light gray

    // Step B: Activate the drawing Camera
    this.mCamera.setupViewProjection();
    this.mMsg.draw(this.mCamera.getVPMMatrix());
};
```

5. In the `update()` function, stop the `GameLoop` to trigger the call to the `unloadScene()` function.

```
GameOver.prototype.update = function() {
    gEngine.GameLoop.stop();
};
```

6. In the `unloadScene()` function, call `gEngine_Core` to clean up all system default resources.

```
GameOver.prototype.unloadScene = function() {
    gEngine.Core.cleanUp(); // release gl resources
};
```

The `GameOver` scene is designed to print the “Game Over” message and terminate the entire game.

The MyGame Scene

You can now finally modify the `MyGame` scene to print messages with the various fonts found in the `assets` folder.

1. In the `MyGame.js` file, modify the constructor to contain corresponding variables for printing the messages, and modify the `draw()` function to draw all objects accordingly. Please refer to the `src/MyGame/MyGame.js` file for the details of the code.
2. Modify the `loadScene()` function to load the textures and fonts.

```
MyGame.prototype.loadScene = function() {
    // Step A: loads the textures
    gEngine.Textures.loadTexture(this.kFontImage);
    gEngine.Textures.loadTexture(this.kMinionSprite);

    // Step B: loads all the fonts
    gEngine.FONTS.loadFont(this.kFontCon16);
    gEngine.FONTS.loadFont(this.kFontCon24);
    gEngine.FONTS.loadFont(this.kFontCon32);
    gEngine.FONTS.loadFont(this.kFontCon72);
    gEngine.FONTS.loadFont(this.kFontSeg96);
};
```

3. Modify the `unloadScene()` function to unload the textures and fonts and to start the `GameOver` scene.

```
MyGame.prototype.unloadScene = function() {
    gEngine.Textures.unloadTexture(this.kFontImage);
    gEngine.Textures.unloadTexture(this.kMinionSprite);

    // unload the fonts
    gEngine.FONTS.unloadFont(this.kFontCon16);
    gEngine.FONTS.unloadFont(this.kFontCon24);
```

```

gEngine.Fnts.unloadFont(this.kFontCon32);
gEngine.Fnts.unloadFont(this.kFontCon72);
gEngine.Fnts.unloadFont(this.kFontSeg96);

// Step B: starts the next level
var nextLevel = new GameOver(); // next level to be loaded
gEngine.Core.startScene(nextLevel);
};

```

4. Define a private `_initText()` function to set the color, location, and height of a `FontRenderable` object. Modify the `initialize()` function to set up the proper WC system and initialize the fonts.

```

MyGame.prototype._InitText = function(font, posX, posY, color, textH) {
    font.setColor(color);
    font.getXform().setPosition(posX, posY);
    font.setTextHeight(textH);
};

MyGame.prototype.initialize = function() {
    // Step A: set up the cameras
    // ... Identical to previous code ...

    // Step B: Create the font and minion images using sprite
    // ... Identical to previous code ...

    // Create the fonts!
    this.mTextSysFont = new FontRenderable("System Font: in Red");
    this._InitText(this.mTextSysFont, 50, 60, [1, 0, 0, 1], 3);

    this.mTextCon16 = new FontRenderable("Consolas 16: in black");
    this.mTextCon16.setFont(this.kFontCon16);
    this._InitText(this.mTextCon16, 50, 55, [0, 0, 0, 1], 2);

    this.mTextCon24 = new FontRenderable("Consolas 24: in black");
    this.mTextCon24.setFont(this.kFontCon24);
    this._InitText(this.mTextCon24, 50, 50, [0, 0, 0, 1], 3);

    this.mTextCon32 = new FontRenderable("Consolas 32: in white");
    this.mTextCon32.setFont(this.kFontCon32);
    this._InitText(this.mTextCon32, 40, 40, [1, 1, 1, 1], 4);

    this.mTextCon72 = new FontRenderable("Consolas 72: in blue");
    this.mTextCon72.setFont(this.kFontCon72);
    this._InitText(this.mTextCon72, 30, 30, [0, 0, 1, 1], 6);

    this.mTextSeg96 = new FontRenderable("Segment7-92");
    this.mTextSeg96.setFont(this.kFontSeg96);
    this._InitText(this.mTextSeg96, 30, 15, [1, 1, 0, 1], 7);

    this.mTextToWork = this.mTextCon16;
};

```

Notice the calls to `setFont()` function to change the font type for each message.

5. Modify the `update()` function with the following:

```
MyGame.prototype.update = function() {
    // Controlling of hero and zooming of font image
    // ... Identical to previous code ...

    // choose which text to work on
    if (gEngine.Input.isKeyClicked(gEngine.Input.keys.Zero))
        this.mTextToWork = this.mTextCon16;
    if (gEngine.Input.isKeyClicked(gEngine.Input.keys.One))
        this.mTextToWork = this.mTextCon24;
    if (gEngine.Input.isKeyClicked(gEngine.Input.keys.Three))
        this.mTextToWork = this.mTextCon32;
    if (gEngine.Input.isKeyClicked(gEngine.Input.keys.Four))
        this.mTextToWork = this.mTextCon72;

    var deltaF = 0.005;
    if (gEngine.Input.isKeyPressed(gEngine.Input.keys.Up)) {
        if (gEngine.Input.isKeyPressed(gEngine.Input.keys.X)) {
            this.mTextToWork.getXform().incWidthBy(deltaF);
        }
        if (gEngine.Input.isKeyPressed(gEngine.Input.keys.Y)) {
            this.mTextToWork.getXform().incHeightBy(deltaF);
        }
        this.mTextSysFont.setText(
            this.mTextToWork.getXform().getWidth().toFixed(2) +
            "x" + this.mTextToWork.getXform().getHeight().toFixed(2));
    }

    if (gEngine.Input.isKeyPressed(gEngine.Input.keys.Down)) {
        if (gEngine.Input.isKeyPressed(gEngine.Input.keys.X)) {
            this.mTextToWork.getXform().incWidthBy(-deltaF);
        }
        if (gEngine.Input.isKeyPressed(gEngine.Input.keys.Y)) {
            this.mTextToWork.getXform().incHeightBy(-deltaF);
        }
        this.mTextSysFont.setText(
            this.mTextToWork.getXform().getWidth().toFixed(2) + "x"
            + this.mTextToWork.getXform().getHeight().toFixed(2));
    }
};
```

The previous code shows that you can do the following:

- 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 FontSupport project to modify each of the displayed font message sizes and to move the hero to trigger the end of the game.

Summary

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 and texture map a selected subregion of an image to the unit square based on the normalize-ranged texture coordinate system. The chapter then explained how sprite sheets can conserve the time required for loading texture images and facilitate the creation of animations. This knowledge was then generalized and applied to the drawing of bitmap fonts.

The implementation of texture mapping and sprite sheet rendering take advantage of an important aspect of game engine architecture: the `Shader/Renderable` object pair where JavaScript `Shader` 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 pairing with `SpriteAnimateRenderable` shows that, when appropriate, the same `Shader` object can support multiple renderable object types in the game engine. This `Shader/Renderable` pair implementation pattern will appear again in Chapter 8, when you learn to create 3D illumination effects. Lastly, you have learned about the importance of cleaning up system resources when the game terminates.

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 supporting intrinsic properties for these objects including pseudo autonomous behaviors such as chasing and collision detections.

Game Design Considerations

In Chapter 4 you learned how responsive game feedback is essential to 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 (referred to as the *hero* moving forward) must maneuver through a 2D platformer-style game world; the player's goal might be to use the mouse and keyboard to jump the hero between individual surfaces rendered in the game without falling through gaps that exist between those surfaces. The visual representation of the hero and other objects in the environment determine how the player identifies with the game setting, which in turn determines how effectively the game creates presence. Is the hero represented as a living creature or just an abstract shape like a square or circle? Are the surfaces represented as building rooftops, as floating rocks on an alien planet, or simply as abstract rectangles? There is no right or wrong answer when it comes to selecting a visual representation or game setting, but it is important to design a visual style for all game elements that feels unified and integrated into whatever game setting you choose (for example, abstract rectangle platforms may negatively impact presence if your game setting is a tropical rainforest).

The Texture Shaders project demonstrated how .png images with transparency more effectively integrate game elements into the game environment than formats like JPEG that don't support transparency. If you move the hero (represented here as simply a rectangle) to the right, nothing on the screen changes, but if you move the hero to the left, you'll eventually trigger a state change that alters the displayed visual elements as you did in the Scene Objects project from Chapter 4. Notice how much more effectively the robot sprites are integrated into the game scene when they're .png files with transparency on the gray background compared to when they're .jpg images without transparency on the blue background. The Sprite Shaders project introduces

a hero that more closely matches other elements in the game setting. You've replaced the hero rectangle from the Texture Shaders project with a humanoid figure stylistically matched to the flying robots on the screen, and the area of the rectangular hero image not occupied by the humanoid figure is transparent. If you were to combine the hero from the Sprite Shaders project with the screen-altering action in the Texture Shaders project, imagine that as the hero moves toward the robot on the right side of the screen, the robot might turn red when the hero gets too close. The coded events are still simple at this point, but you can see how the visual design and a few simple triggered actions can already begin to convey a game setting and enhance presence.

Note that as game designers, we often become enamored with highly detailed and elaborate visual designs, and we begin to believe that higher fidelity and more elaborate visual elements are required to make the best games. This drive for ever-more powerful 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. Good 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 hyper-real 3D environments, and every combination in between.

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; you 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 engaging 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. And again, imagine combining the Sprite Animation project with the earlier projects in this chapter. As the hero moves closer to the robot, it might turn first turn red; closer still might trigger the robot's animations and move it either toward or away from the player. Animations often come fairly late in the game design process because it's helpful to have the game mechanic and other systems well defined so that animations, which tend to be time-consuming to change once complete, can be produced after level designs have been finalized and tested with placeholder assets.

As was the case with visual design, the animation approach need not be especially complex to be effective. Animation needs to be intentional and unified, and it 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.

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. Note how the yellow font conveys a digital feeling that's matched to the science fiction-inspired visual style of the hero and robots, while the Consolas font family with its round letterforms feels a bit out of place with this game setting (sparse though the game setting may still be). As a more extreme example, imagine how disconnected a flowing calligraphic script font (the type typically used in high-fantasy games) would appear in a futuristic game that takes place on a spaceship.

There are as many visual style possibilities for games as there are people and ideas, and great games can feature extremely simple graphics. Remember that excellent game design is a combination of the nine contributing elements (return to the introduction if you need to refresh your memory), and the most important thing to keep in mind as a game designer is maintaining focus on how each of those elements harmonizes with and elevates the others to create something greater than the sum of its parts.

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 and 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 result of this shortcoming in the `initialize()` and `update()` functions of the `MyGame` objects in all the previous projects: the `initialize()` 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 abstract and hide the specifics of initialization and behaviors. There are two main advantages to this approach. First, the `initialize()` and `update()` functions of a game level can focus on managing individual game object 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 `Shader` object 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 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 relevant objects in the game. In general, resolving relative positions of all objects in a two-dimensional world is nontrivial. Fortunately, typical video games require the knowledge only of objects that are in close proximity to each other or of detecting objects that are about to collide or have collided. An efficient but somewhat crude approximation to detect collision is to compute, bound, and collide objects with bounding boxes. In the simplest cases, bounding boxes are rectangular boxes with edges that are aligned with the x/y-axes. Because of the axes alignments, it is computationally efficient to detect when two bounding boxes overlap or when

collision is about to occur. Most 2D game engines 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 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 target-locked 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 `initialize()` and `update()` functions of a game level and to facilitate reuse. This section introduces the simple `GameObject` class to illustrate how the cleaner and unclustered `initialize()` 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.

The Game Objects Project

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.GameObjects` folder.



Figure 6-1. Running the Game Objects project

The controls of the project are as follows:

- WASD keys: Move the hero up, left, down, and right

The goals of the project are as follows:

- To improve the engine drawing infrastructure to receive the Camera object instead of just the View-Projection matrix of a camera. Although this goal is minimally related to game object behaviors, this is a necessary step to include in order to support the growing game engine sophistication.
- To begin defining the GameObject class to encapsulate object behaviors in games.
- To demonstrate the creation of subclasses to the GameObject class to maintain the simplicity of the MyGame level update() function.

You can find the following external resource file in the assets folder: `minion_sprite.png`; you'll also find the fonts folder that contains the default system fonts. Note that, as shown in Figure 6-2, the `minion_sprite.png` image file has been updated from the previous project to include two extra sprite elements: the DyePack and the Brain minion.

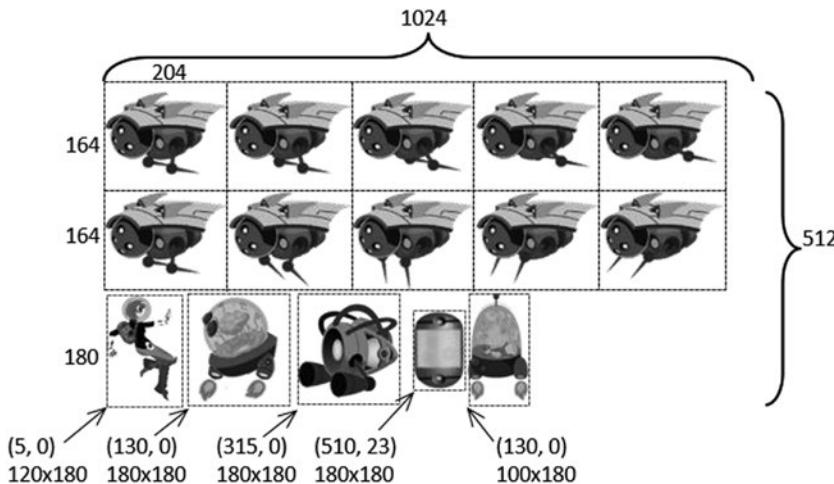


Figure 6-2. The new sprite elements of the `minion_sprite.png` image

Modify Renderable and Shader Objects to Draw with a Camera

As the game engine improves to a higher level of abstraction, it is important to upgrade its drawing capability from receiving and working with a simple View-Projection transformation to a `Camera` object. This change prepares the engine to support the more sophisticated shaders to be covered in later chapters. This change involves all the `Shader` and `Renderable` objects. Though many files are involved, the actual changes are mundane; they consist of a simple replacement of the matrix with a camera. For this reason, only changes to the base objects, `Renderable` and `SimpleShader`, are shown here. You can refer to the source code files of `TextureShader`, `SpriteShader`, `TextureRenderable`, `SpriteRenderable`, `SpriteAnimateRenderable`, and `FontRenderable` for details.

1. Modify the `Renderable` object's `draw()` function to replace the `vpMatrix` parameter with `aCamera` and forward this camera to the shader.

```
Renderable.prototype.draw = function (aCamera) {
    var gl = gEngine.Core.getGL();
    this.mShader.activateShader(this.mColor, aCamera);
    this.mShader.loadObjectTransform(this.mXform.getXform());
    gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);
};
```

2. Modify the `SimpleShader` object's `activateShader()` function to work with a camera.

```
SimpleShader.prototype.activateShader = function (pixelColor, aCamera) {
    var gl = gEngine.Core.getGL();
    gl.useProgram(this.mCompiledShader);
    gl.uniformMatrix4fv(this.mViewProjTransform, false, aCamera.getVPMATRIX());
    gl.enableVertexAttribArray(this.mShaderVertexPositionAttribute);
    gl.uniform4fv(this.mPixelColor, pixelColor);
};
```

Notice that these changes are trivial replacements of the parameter. Please refer to the source code files for all similar changes to all the objects derived from the `Renderable` and `SimpleShader` objects.

Define the GameObject

The goal is to define a logical abstraction to integrate the behavioral characteristics of a game object including the ability to control positions, drawing, and so on. As in the case for the `Scene` objects in the earlier chapter, the main 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. This example demonstrates the potentials of only the `GameObject` class with minimal behaviors defined.

1. Add a new folder `src/Engine/GameObjects` for storing `GameObject`-related files.
2. Create a new file in this folder, name it `GameObject.js`, and add the following code:

```
function GameObject(renderableObj) {
    this.mRenderComponent = renderableObj;
}

GameObject.prototype.getXform = function () {
    return this.mRenderComponent.getXform();
};

GameObject.prototype.update = function () {};

GameObject.prototype.getRenderable = function () {
    return this.mRenderComponent;
};

GameObject.prototype.draw = function (aCamera) {
    this.mRenderComponent.draw(aCamera);
};
```

With the assessors to the `Renderable` and `Transform` objects defined, all `GameObject` instances can be drawn and have defined locations and sizes. Note that the `update()` function is designed for subclasses to override with per-object specific behaviors, and thus it is left empty.

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/GameObjects` folder and name it `GameObjectSet.js`. Define the constructor to create an array for holding `GameObject` instances.

```
function GameObjectSet() {
    this.mSet = [];
}
```

2. Define functions for managing the set membership.

```
GameObjectSet.prototype.size = function() { return this.mSet.length; };

GameObjectSet.prototype.getObjectAt = function(index) {
    return this.mSet[index];
};

GameObjectSet.prototype.addToSet = function(obj) {
    this.mSet.push(obj);
};
```

3. Define functions to update and draw each of the `GameObject` instances in the set.

```
GameObjectSet.prototype.update = function() {
    var i;
    for (i = 0; i < this.mSet.length; i++) {
        this.mSet[i].update();
    }
};

GameObjectSet.prototype.draw = function(aCamera) {
    var i;
    for (i = 0; i < this.mSet.length; i++) {
        this.mSet[i].draw(aCamera);
    }
};
```

Test the `GameObject` and `GameObjectSet`

The goals are to ensure proper functioning of the new `GameObject` class, to demonstrate customization of behaviors by individual object types, and to observe a cleaner `MyGame` implementation clearly reflecting the in-game logic. To accomplish these goals, three object types are defined: `DyePack`, `Hero`, and `Minion`. Before you begin to examine the detail implementation of these objects, following good source code organization practice, create the new folder `src/MyGame/Objects` for storing the new object types.

The `DyePack` `GameObject`

The `DyePack` object derives from the `GameObject` class to demonstrate the simplest example of a `GameObject` where the object can be drawn only without any specific behaviors.

- Create a new file in the `src/MyGame/Objects/` folder and name it `DyePack.js`. Define this as a subclass of `GameObject` and implement the constructor as follows:

```
function DyePack(spriteTexture) {
    this.kRefWidth = 80;
    this.kRefHeight = 130;

    this.mDyePack = new SpriteRenderable(spriteTexture);
    this.mDyePack.setColor([1, 1, 1, 0.1]);
    this.mDyePack.getXform().setPosition(50, 33);
```

```

        this.mDyePack.getXform().setSize(this.kRefWidth / 50, this.kRefHeight / 50);
        this.mDyePack.setElementPixelPositions(510, 595, 23, 153);
        GameObject.call(this, this.mDyePack);
    }
gEngine.Core.inheritPrototype(DyePack, GameObject);

```

Notice that even without specific behaviors, the DyePack is implementing code that used to be found in the `initialize()` function of the MyGame level. In this way, the DyePack object hides specific geometric information and simplifies the MyGame level.

The Hero GameObject

The Hero object supports direct user keyboard control. This object demonstrates hiding of game object control logic from the MyGame level's `update()` function.

1. Create a new file in the `src/MyGame/Objects/` folder and name it `Hero.js`. Define this as a subclass of `GameObject` and implement the constructor to initialize the sprite UV values, size, and position as follows:

```

function Hero(spriteTexture) {
    this.kDelta = 0.3;
    this.mDye = new SpriteRenderable(spriteTexture);
    this.mDye.setColor([1, 1, 1, 0]);
    this.mDye.getXform().setPosition(35, 50);
    this.mDye.getXform().setSize(9, 12);
    this.mDye.setElementPixelPositions(0, 120, 0, 180);
    GameObject.call(this, this.mDye);
}
gEngine.Core.inheritPrototype(Hero, GameObject);

```

2. Add a function to support the update of this object by user keyboard control:

```

Hero.prototype.update = function () {
    // control by WASD
    var xform = this.getXform();
    if (gEngine.Input.isKeyPressed(gEngine.Input.keys.W))
        xform.incYPosBy(this.kDelta);
    if (gEngine.Input.isKeyPressed(gEngine.Input.keys.S))
        xform.incYPosBy(-this.kDelta);
    if (gEngine.Input.isKeyPressed(gEngine.Input.keys.A))
        xform.incXPosBy(-this.kDelta);
    if (gEngine.Input.isKeyPressed(gEngine.Input.keys.D))
        xform.incXPosBy(this.kDelta);
};

```

The Hero object moves at a `kDelta` rate based on WASD input from the keyboard.

The Minion GameObject

The Minion object demonstrates that simple autonomous behavior can also be hidden.

1. Create a new file in the `src/MyGame/Objects/` folder and name it `Minion.js`.

Define this as a subclass of `GameObject` and implement the constructor to initialize the sprite UV values, sprite animation parameters, size, and position as follows:

```
function Minion(spriteTexture, atY) {
    this.kDelta = 0.2;
    this.mMinion = new SpriteAnimateRenderable(spriteTexture);
    this.mMinion.setColor([1, 1, 1, 0]);
    this.mMinion.getXform().setPosition(Math.random() * 100, atY);
    this.mMinion.getXform().setSize(12, 9.6);
    this.mMinion.setSpriteSequence(512, 0, // first element pixel
        204, 164, // widthxheight in pixels
        5, // number of elements
        0); // horizontal padding in between
    this.mMinion.setAnimationType(SpriteAnimateRenderable.eAnimationType.eAnimateSwing);
    this.mMinion.setAnimationSpeed(15); // show each element for mAnimSpeed updates
    GameObject.call(this, this.mMinion);
}
gEngine.Core.inheritPrototype(Minion, GameObject);
```

2. Add a function to update the sprite animation, support the simple right-to-left movements, and provide the wrapping functionality.

```
Minion.prototype.update = function () {
    // remember to update this.mMinion's animation
    this.mMinion.updateAnimation();

    // move toward the left and wraps
    var xform = this.getXform();
    xform.incXPosBy(-this.kDelta);

    // if fly off to the left, re-appear at the right
    if (xform.getXPos() < 0) {
        xform.setXPos(100);
        xform.setYPos(65 * Math.random());
    }
};
```

MyGame Scene

As in all cases, the `MyGame` level is implemented in the `MyGame.js` file. In this case, with the three specific `GameObject` subclasses defined, follow these steps:

1. The constructor and the `loadScene()`, `unloadScene()`, and `draw()` functions are similar as in previous projects, so the details are not shown here. Please refer to the `MyGame.js` file for the actual implementations.

2. Edit the `initialize()` function and add the following code:

```
MyGame.prototype.initialize = function () {
    // ... identical code to previous project ...

    // Step B: The dye pack: simply another GameObject
    this.mDyePack = new DyePack(this.kMinionSprite);

    // Step C: A set of Minions
    this.mMinionset = new GameObjectSet();
    var i = 0, randomY, aMinion;
    // create 5 minions at random Y values
    for (i = 0; i < 5; i++) {
        randomY = Math.random() * 65;
        aMinion = new Minion(this.kMinionSprite, randomY);
        this.mMinionset.addToSet(aMinion);
    }

    // Step D: Create the hero object
    this.mHero = new Hero(this.kMinionSprite);

    // Step E: Create and initialize message output
    this.mMsg = new FontRenderable("Status Message");
    this.mMsg.setColor([0, 0, 0, 1]);
    this.mMsg.getXform().setPosition(1, 2);
    this.mMsg.setTextHeight(3);
};
```

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`. Notice that the initialization function is free from the clustering of setting each object's textures, geometries, and so on.

3. Edit the `update()` function to update the game state.

```
MyGame.prototype.update = function () {
    this.mHero.update();
    this.mMinionset.update();
    this.mDyePack.update();
};
```

With the well-defined behaviors for each object type abstracted, the clean `update()` function clearly shows that the game consists of three noninteracting objects.

You can now run the project and notice that the slightly more complex movements of six minions are accomplished with much cleaner `initialize()` and `update()` functions. The `initialize()` function simply consists of only logic and controls of 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 `MyGame` level's `update()` function can focus on object-level controls without being clustered by the details. Note that the structure of this function clearly shows the three objects are updated independently and do not interact with each other.

Chasing of a GameObject

A closer examination of the previous project reveals that though there are quite a few minions moving on the screen, their motions are simple and boring. Even though there are variations in speed and direction, the motions are without purpose or awareness of other game objects in the scene. To support more sophisticated or interesting movements, a `GameObject` needs to be aware of the locations of other objects and determine motion based on that information.

Chasing behavior is one such example. The goal of a chasing object is usually to catch the game object that it is targeting. This requires programmatic manipulation of the chaser's front direction and speed so it can hone in on its target. However, it is generally important to avoid implementing a chaser that has perfect aim and always hits its target—because if the projectile always hits its target and characters are unable to avoid being hit, the challenge will essentially be removed from the game. Nonetheless, this does not mean you should not implement a perfect chaser if your game design requires it. You will implement a chaser in the next project.

Vectors and the associated operations are the foundation for implementing object movements and behaviors. Before programming with vectors, a quick review is provided. As in the case of matrices and transform operators, the following discussion is not meant to be a comprehensive coverage of vectors. Instead, the focus is on the application of a small collection of concepts that are relevant to the implementation of the game engine. This is not a study of the theories behind the mathematics. If you are interested in the specifics of vectors and how they relate to computer graphics, please refer to the discussion in Chapter 1 where you can learn more about these topics in depth by delving into relevant books on linear algebra and computer graphics.

Vectors Review

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.

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 an object's velocity or acceleration. 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.

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 (\bar{V}_{ab}) as $P_b - P_a$. You can see this represented in the following equations and Figure 6-3:

- $P_a = (x_a, y_a)$
- $P_b = (x_b, y_b)$
- $\bar{V}_{ab} = P_b - P_a = (x_b - x_a, y_b - y_a)$

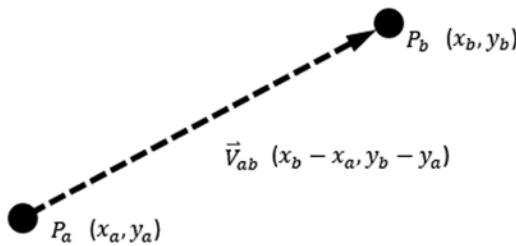


Figure 6-3. A vector defined by two points

Now that you have a vector \bar{V}_{ab} , you can easily ascertain its length (or size) and its direction. A vector's length is equal to the distance between the two points that created it. In this example, the length of \bar{V}_{ab} is equal to the distance between P_a and P_b , while the direction of \bar{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 two-dimensional (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 \bar{V}_{ab} can all be implemented as instances of the `vec2` object; however, \bar{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 and Figure 6-4. Notice that the mathematic symbol for a regular vector is \bar{V}_a and for a normalized vector is \hat{V}_a :

- `vec2.normalized(` \bar{V}_a `)`: Normalizes vector \bar{V}_a and stores the results to the `vec2` object

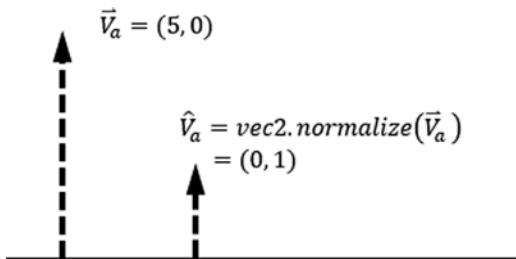


Figure 6-4. A vector being normalized

Vectors to a position can also be rotated. If, for example, the vector $\bar{V} = (x_v, y_v)$ represents the direction from the origin to the position (x_v, y_v) and you want to rotate it by θ , then you can use the following equations shown to derive x_r and y_r . Figure 6-5 shows the rotation of a vector to a position from the origin being applied.

- $x_r = x_v \cos \theta - y_v \sin \theta$
- $y_r = x_v \sin \theta + y_v \cos \theta$

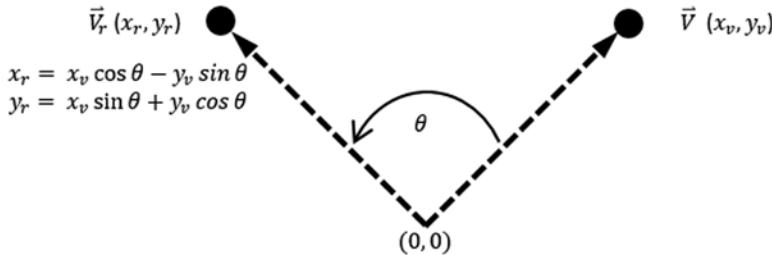


Figure 6-5. A vector from the origin to a position being rotated by the angle theta

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.

Furthermore, it is 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. Because their direction and magnitude are the same, these vectors 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.

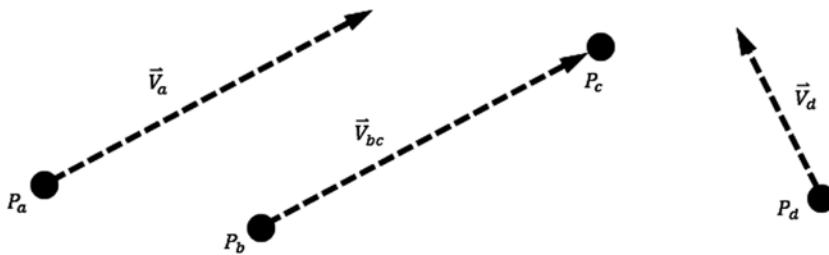


Figure 6-6. Three vectors represented in 2D space with two vectors equal to each other

The Dot Product

The dot product of two normalized vectors provides you with the means to find the angle between those vectors. For example, given the following:

- $\vec{V}_1 = (x_1, y_1)$
- $\vec{V}_2 = (x_2, y_2)$

then the following is true:

- $\vec{V}_1 \cdot \vec{V}_2 = \vec{V}_2 \cdot \vec{V}_1 = x_1 x_2 + y_1 y_2$.

Additionally, if both vectors \vec{V}_1 and \vec{V}_2 are normalized, then:

- $\hat{V}_1 \cdot \hat{V}_2 = \cos \theta$.

You can see this concept reinforced in Figure 6-7. Moreover, it is also important to recognize that if $\vec{V}_1 \cdot \vec{V}_2 = 0$, then the two vectors are perpendicular.

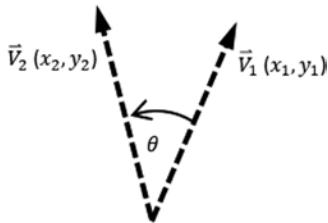


Figure 6-7. The angle between two vectors, which can be found through the dot product

Note If you need to review or refresh the concept of a dot product, please refer to <http://www.mathsisfun.com/algebra/vectors-dot-product.html>.

The Cross Product

The cross product of two vectors produces a vector that is *orthogonal*, or perpendicular, to both of the original vectors. In 2D games, where the 2D dimensions lie flat on the screen, the cross product results in a vector that points either inward (toward the screen) or outward (away from the screen). This may seem odd because it is not intuitive that crossing two vectors in 2D or the XY plane results in a vector that lies in the third dimension, or along the z-axis. However, this vector is essential for the sole purpose of determining 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 .

Also, you know that the cross product of two vectors on the XY 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.

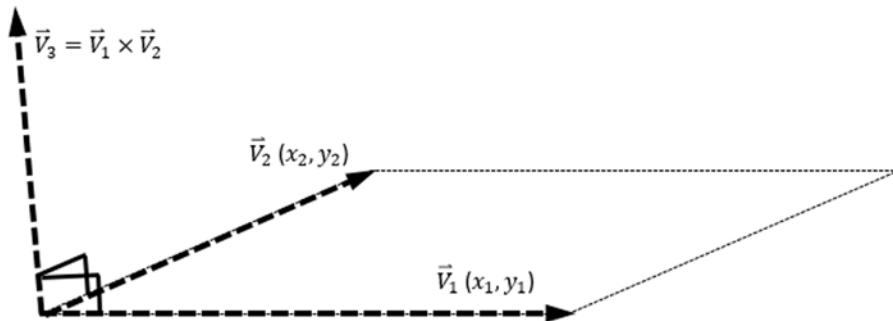


Figure 6-8. The cross product on two vectors

Note If you need to review or refresh the concept of a cross product, please refer to <http://www.mathsisfun.com/algebra/vectors-cross-product.html>.

The Front and Chase Project

This project implements more interesting and sophisticated behaviors based on the vector concepts reviewed. Instead of constant and aimless motions, you will experience defining and varying the front direction of an object and guiding an object to chase after another object in the scene. You can see an example of this project running in Figure 6-9. The source code to this project is defined in the Chapter6/6.2.FrontAndChase folder.



Brain modes [H:keys, J:immediate, K:gradual]: H

Figure 6-9. Running the *Front and Chase* project

The controls of the project are as follows:

- *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 experience working with speed and direction
- To practice traveling along a predefined direction

- To implement algorithms with vector dot and cross products
- To examine and implement chasing behavior

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 the projects from this point forward.

Modify GameObject to Support Interesting Behaviors

The `GameObject` class abstracts and implements the desired new object behaviors:

1. Edit the `GameObject.js` file and modify the `GameObject` constructor to define visibility, front direction, and speed.

```
function GameObject(renderableObj) {
  this.mRenderComponent = renderableObj;
  this.mVisible = true;
  this.mCurrentFrontDir = vec2.fromValues(0, 1); // current front direction
  this.mSpeed = 0;
}
```

2. Add assessor and setter functions for the instance variables.

```
GameObject.prototype.getXform = function () { return this.mRenderComponent.getXform(); };
GameObject.prototype.setVisibility = function (f) { this.mVisible = f; };
GameObject.prototype.isVisible = function () { return this.mVisible; };

GameObject.prototype.setSpeed = function (s) { this.mSpeed = s; };
GameObject.prototype.getSpeed = function () { return this.mSpeed; };
GameObject.prototype.incSpeedBy = function (delta) { this.mSpeed += delta; };
```

```
GameObject.prototype.setCurrentFrontDir = function (f) { vec2.normalize(this.mCurrentFrontDir, f); };
GameObject.prototype.getCurrentFrontDir = function () { return this.mCurrentFrontDir; };

GameObject.prototype.getRenderable = function () { return this.mRenderComponent; };
```

3. Implement a function to rotate the front direction toward a position, p.

```
GameObject.prototype.rotateObjPointTo = function (p, rate) {
    // Step A: determine if reach the destination position p
    var dir = [];
    vec2.sub(dir, p, this.getXform().getPosition());
    var len = vec2.length(dir);
    if (len < Number.MIN_VALUE)
        return; // we are there.
    vec2.scale(dir, dir, 1 / len);

    // Step B: compute the angle to rotate
    var fdir = this.getCurrentFrontDir();
    var cosTheta = vec2.dot(dir, fdir);
    if (cosTheta > 0.999999) // almost exactly the same direction
        return;

    // Step C: clamp the cosTheta to -1 to 1
    // in a perfect world, this would never happen! BUT ...
    if (cosTheta > 1)
        cosTheta = 1;
    else
        if (cosTheta < -1)
            cosTheta = -1;

    // Step D: compute whether to rotate clockwise, or counterclockwise
    var dir3d = vec3.fromValues(dir[0], dir[1], 0);
    var f3d = vec3.fromValues(fdir[0], fdir[1], 0);
    var r3d = [];
    vec3.cross(r3d, f3d, dir3d);

    var rad = Math.acos(cosTheta); // radian to roate
    if (r3d[2] < 0)
        rad = -rad;

    // Step E: rotate the facing direction with the angle and rate
    rad *= rate; // actual angle need to rotate from Obj's front
    vec2.rotate(this.getCurrentFrontDir(), this.getCurrentFrontDir(), rad);
    this.getXform().incRotationByRad(rad);
};
```

The `rotateObjPointTo()` function rotates the `mCurrentFrontDir` to point to destination position `p` at the rate specified by the parameter. Here are the detailed operations:

- a. Step A computes the distance between the current object and the destination position `p`. If this value is small, it means current object and the target position are close. The function returns without further processing.
- b. Step B computes the dot product to determine the angle between the current front direction of the object (`fdir`) and the direction toward the destination position `p` (`dir`). If these two vectors are pointing in the same direction, the function returns. This computation is illustrated in Figure 6-10.

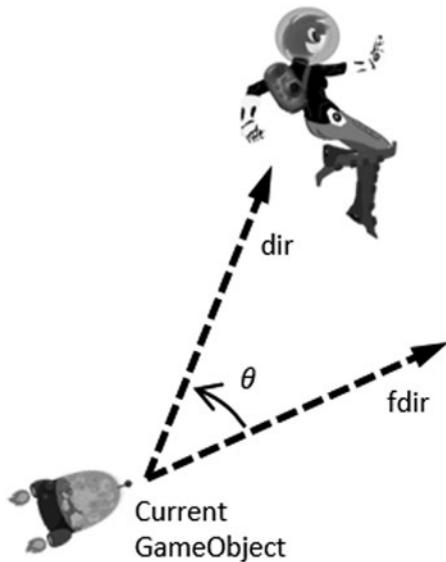


Figure 6-10. A `GameObject` (Brain) chasing a target (Hero)

- c. Step C checks for the range of `cosTheta`. This is a step that must be performed because of the inaccuracy of floating-point operations in JavaScript.
- d. Step D computes 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.
4. Add a function to update the object's position with its direction and speed.

```
GameObject.prototype.update = function () {
    // simple default behavior
    var pos = this.getXform().getPosition();
    vec2.scaleAndAdd(pos, pos, this.getCurrentFrontDir(), this.getSpeed());
};
```

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.

5. Add a function to draw the object based on the visibility setting.

```
GameObject.prototype.draw = function (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 for testing traveling along a predefined front direction and direct the `Brain` object to chase after the `Hero` object to test 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/MyGame/Objects` folder and name it `Brain.js`. Define this as a subclass of `GameObject` and implement the constructor to initialize the appearance and initial behavior of the object.

```
function Brain(spriteTexture) {
    this.kDeltaDegree = 1;
    this.kDeltaRad = Math.PI * this.kDeltaDegree / 180;
    this.kDeltaSpeed = 0.01;
    this.mBrain = new SpriteRenderable(spriteTexture);
    this.mBrain.setColor([1, 1, 1, 0]);
    this.mBrain.getXform().setPosition(50, 10);
    this.mBrain.getXform().setSize(3, 5.4);
    this.mBrain.setElementPixelPositions(600, 700, 0, 180);
    GameObject.call(this, this.mBrain);
    this.setSpeed(0.05);
}
gEngine.Core.inheritPrototype(Brain, GameObject);
```

2. Override the `update()` function to support user steering and controlling the speed. Notice that the default `update()` function in the `GameObject` must be called to support the basic traveling of the object along the front direction according to its speed.

```
Brain.prototype.update = function () {
    GameObject.prototype.update.call(this); // default moving forward
    var xf = this.getXform();
    var fdir = this.getCurrentFrontDir();
    if (gEngine.Input.isKeyPressed(gEngine.Input.keys.Left)) {
        xf.incRotationByDegree(this.kDeltaDegree);
        vec2.rotate(fdir, fdir, this.kDeltaRad);
    }
}
```

```

if (gEngine.Input.isKeyPressed(gEngine.Input.keys.Right)) {
    xf.incRotationByRad(-this.kDeltaRad);
    vec2.rotate(fdir, fdir, -this.kDeltaRad);
}
if (gEngine.Input.isKeyClicked(gEngine.Input.keys.Up))
    this.incSpeedBy(this.kDeltaSpeed);
if (gEngine.Input.isKeyClicked(gEngine.Input.keys.Down))
    this.incSpeedBy(-this.kDeltaSpeed);
};

```

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 MyGame.js is similar to previous projects. For this reason, only the details of the update() function are shown. Please refer to the source code in the MyGame.js file for the rest of the implementation details.

```

MyGame.prototype.update = function () {
    var msg = "Brain modes [H:keys, J:immediate, K:gradual]: ";
    var rate = 1;

    this.mHero.update();

    switch (this.mMode) {
        case 'H':
            this.mBrain.update(); // player steers with arrow keys
            break;
        case 'K':
            rate = 0.02; // graduate rate
            // When "K" is typed, the following should also be executed.
        case 'J':
            this.mBrain.rotateObjPointTo(this.mHero.getXform().getPosition(), rate);
            GameObject.prototype.update.call(this.mBrain);
            break;
    }

    if (gEngine.Input.isKeyClicked(gEngine.Input.keys.H))
        this.mMode = 'H';
    if (gEngine.Input.isKeyClicked(gEngine.Input.keys.J))
        this.mMode = 'J';
    if (gEngine.Input.isKeyClicked(gEngine.Input.keys.K))
        this.mMode = 'K';

    this.mMsg.setText(msg + this.mMode);
};

```

In the update() function, the switch statement uses mMode to determine how to update the Brain object. In the cases of J and K modes, the Brain object turns toward the Hero object position with the rotateObjPointTo() function call. While in the H mode, the Brain object update() function is called for the user to steer the object with the arrow keys. The final three if statements simply set the mMode variable according to user input.

You can now try running the project. Initially the Brain object is under the user control. You can use the Left and Right Arrow keys to change the front direction of the Brain object to experience with steering of the object. Pressing the J key causes the Brain object to immediately point and move toward the Hero object. This is a result of the default turn rate value of 1.0. The K key causes a more natural behavior, where the Brain object continues to move forward and turns gradually to moves toward the Hero object. Feel free to change the values of the rate variable or modify the controls value of the Brain object; for example, change the kDeltaRad or kDeltaSpeed to experience with different behavior settings.

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 misses the crucial ability to detect that it has collided with the Hero object, and thus it can stop moving. This section describes the axes-aligned bounding boxes, one of the most straightforward tools for approximating object collisions, and demonstrates the implementation of collision detection based on bounding boxes.

Bounding Box

A bounding box is an x/y-axes aligned rectangular box that bounds a given object. The term *x/y-axes aligned* refers to the fact that the four sides of a bounding box 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 Brain object by the lower-left corner (`mLL`), width, and height. This is a fairly common way to represent a bounding box because it uses only one position and two floating-point numbers to represent the dimensions.

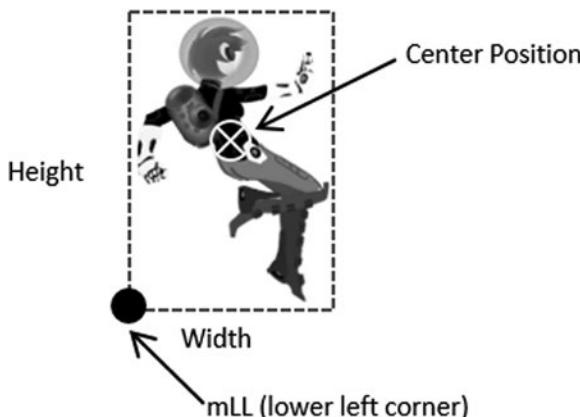


Figure 6-11. The lower-left corner and size of the bounds for an object

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. A bounding box 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.

The Bounding Box and Collisions Project

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.BBoxAndCollisions folder.



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 a 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 Object

Define a `BoundingBox` object to represent the bounds of a rectangular area.

1. Create a new file in the `src/Engine` folder, name it `BoundingBox.js`, and define its constructor with instance variables to represent a bound, as illustrated in Figure 6-11.

```
function BoundingBox(centerPos, w, h) {
    this.mLL = vec2.fromValues(0, 0); // the lower-left corner position
    this.setBounds(centerPos, w, h);
}
```

2. The `setBounds()` function computes and sets the instance variables of the bounding box.

```
BoundingBox.prototype.setBounds = function (centerPos, w, h) {
    this.mWidth = w;
    this.mHeight = h;
    this.mLL[0] = centerPos[0] - (w / 2);
    this.mLL[1] = centerPos[1] - (h / 2);
};
```

3. Define an enumerated data type with values that identify the collision sides of a bounding box.

```
BoundingBox.eboundCollideStatus = Object.freeze({
    eCollideLeft: 1,
    eCollideRight: 2,
    eCollideTop: 4,
    eCollideBottom: 8,
    eInside : 16,
    eOutside: 0
});
```

Notice that each enumerated value is different and has only one nonzero bit. This allows the enumerated values to be combined with the bitwise-or operator to represent a multisided collision. For example, if an object collides with both the top and left sides of a bounding box, the collision status will be `eCollideLeft | eCollideTop = 4 | 1 = 5`.

4. Define a function to determine whether a given position, (x, y) , is within the bounds of the box.

```
BoundingBox.prototype.containsPoint = function (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.

```
BoundingBox.prototype.intersectsBound = function (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.

```
BoundingBox.prototype.boundCollideStatus = function (otherBound) {
    var status = BoundingBox.eboundCollideStatus.eOutside;
    if (this.intersectsBound(otherBound)) {
        if (otherBound.minX() < this.minX())
            status |= BoundingBox.eboundCollideStatus.eCollideLeft;
        if (otherBound.maxX() > this.maxX())
            status |= BoundingBox.eboundCollideStatus.eCollideRight;
        if (otherBound.minY() < this.minY())
            status |= BoundingBox.eboundCollideStatus.eCollideBottom;
        if (otherBound.maxY() > this.maxY())
            status |= BoundingBox.eboundCollideStatus.eCollideTop;
        // if the bounds intersect and yet none of the sides overlaps
        // otherBound is completely inside thisBound
        if (status === BoundingBox.eboundCollideStatus.eOutside)
            status = BoundingBox.eboundCollideStatus.eInside;
    }
    return status;
};
```

Notice the subtle yet important difference between the `intersectsBound()` and `boundCollideStatus()` functions where the former is capable only of returning a true or false condition while the latter function encodes the colliding sides in the returned `status`.

7. Implement the functions that returns the X/Y values to the min and max bounds of the bounding box.

```
BoundingBox.prototype minX = function () { return this.mLL[0]; };
BoundingBox.prototype maxX = function () { return this.mLL[0] + this.mWidth; };
BoundingBox.prototype minY = function () { return this.mLL[1]; };
BoundingBox.prototype maxY = function () { return this.mLL[1] + this.mHeight; };
```

Use the BoundingBox in the Engine

Follow these steps:

1. Modify the `GameObject` class to return the bounding box of the unrotated Renderable object.

```
GameObject.prototype.getBBox = function () {
    var xform = this.getXform();
    var b = new BoundingBox(xform.getPosition(), xform.getWidth(), xform.getHeight());
    return b;
};
```

2. Modify the `Camera` object to compute the collision status between the bounds to a `Transform` object (typically defined in a `Renderable` object) and that of the WC window.

```
Camera.prototype.collideWCBound = function (aXform, zone) {
    var bbox = new BoundingBox(aXform.getPosition(), aXform.getWidth(), aXform.getHeight());
    var w = zone * this.getWCWidth();
    var h = zone * this.getWCHeight();
    var cameraBound = new BoundingBox(this.getWCCenter(), w, h);
    return cameraBound.boundCollideStatus(bbox);
};
```

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.

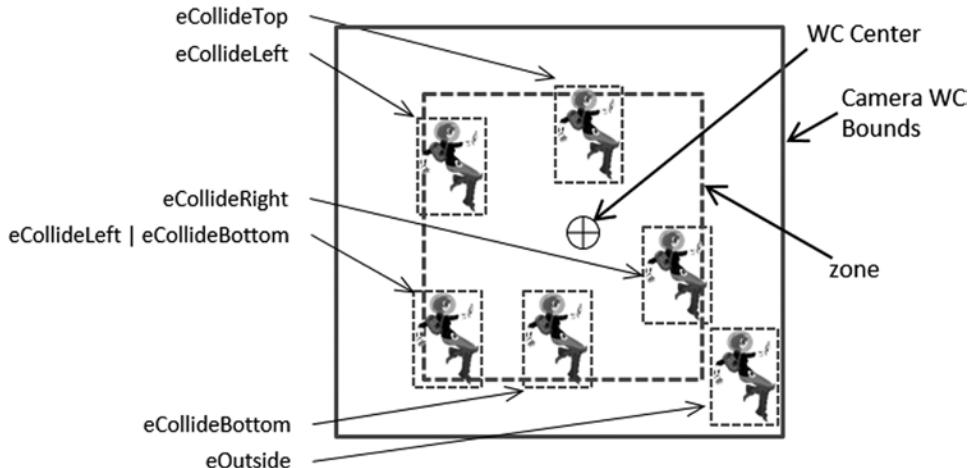


Figure 6-13. Camera WC Bounds colliding with the bounds defined a Transform object

Test Bounding Boxes with MyGame

The goal of this test case is to verify the correctness of the bounding box implementation in detecting object-object and object-camera intersections. Once again, with the exception of the `update()` function, the majority of the code in the `MyGame.js` file is similar to the previous projects and is not repeated here. The `update()` function is modified from the previous project to test for bounding box intersections.

```
MyGame.prototype.update = function () {
    // ... identical code to previous project ...

    switch (this.mMode) {
        case 'H':
            this.mBrain.update(); // player steers with arrow keys
            break;
        case 'K':
            rate = 0.02; // graduate rate
            // no break here on purpose
        case 'J':
            if (!hBbox.intersectsBound(bBbox)) { // stop brain when touches hero
                this.mBrain.rotateObjPointTo(
                    this.mHero.getXform().getPosition(), rate);
                GameObject.prototype.update.call(this.mBrain);
            }
            break;
    }

    // Check for hero going outside 80% of the WC Window bound
    var status = this.mCamera.collideWCBound(this.mHero.getXform(), 0.8);

    // ... identical code to previous project ...
    this.mMsg.setText(msg + this.mMode + " [Hero bound=" + status + "]");
};
```

The previous modification tests for bounding box collision between the `Brain` and `Hero` objects before invoking the `Brain.rotateObjPointTo()` and `update()` functions to cause the chasing behavior. In this way, the `Brain` object will stop moving as soon as it touches the bounds of the `Hero` object. In addition, the collision results between the `Hero` object and 80 percent of the camera WC window are computed and echoed.

You can now run the project and observe the `Brain` object stop moving as soon as it touches the `Hero` object. When you move the `Hero` object around, observe that the output `Hero Bound` message echoes 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.

Per-Pixel Collisions

In the previous example, you saw the results of bounding box collision approximation. Namely, the Brain object's motion stops as soon as its bounds overlap that of the Hero object. This is much improved over the original situation where the Brain object never stops moving. However, as illustrated in Figure 6-14, there are two serious limitations to the bounding box-based collisions.

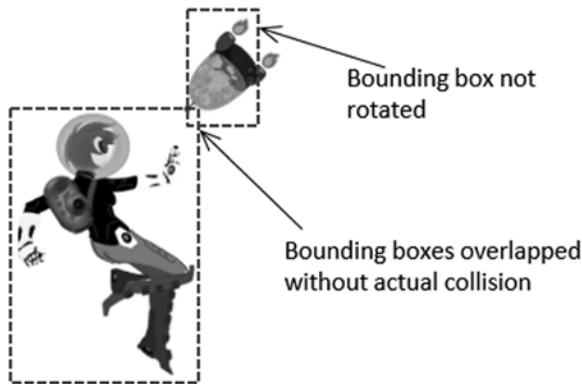


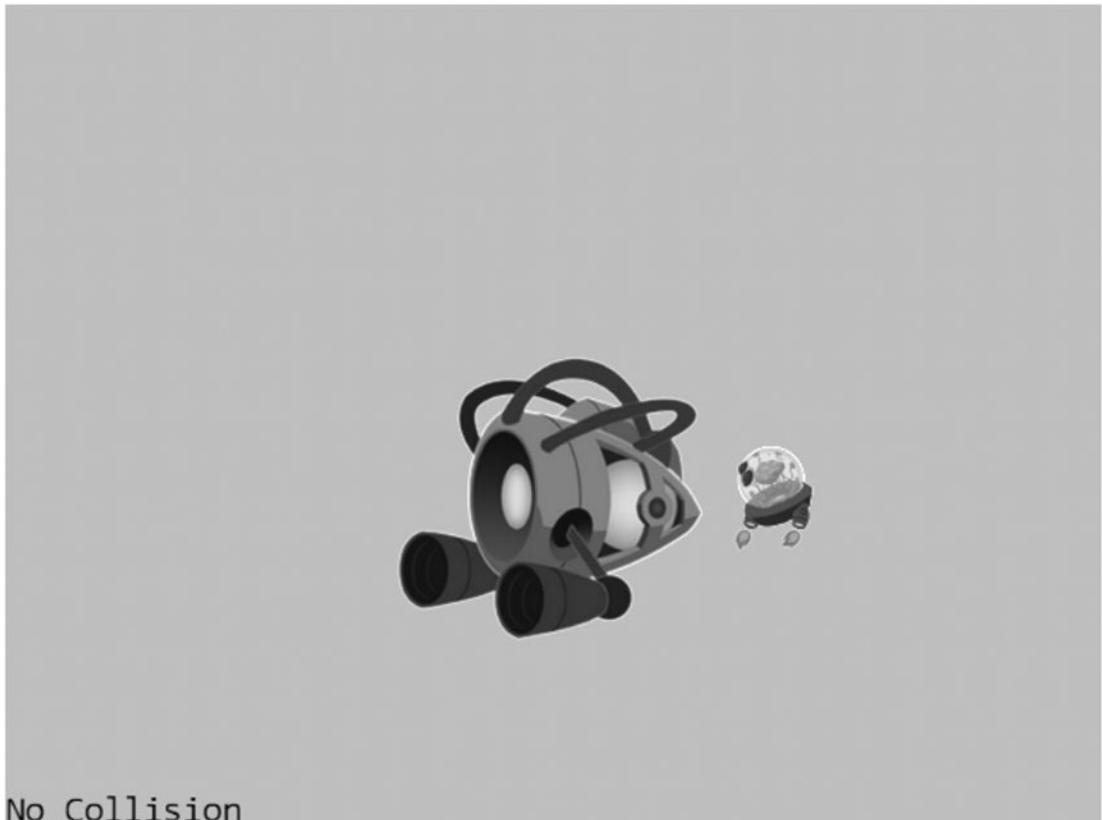
Figure 6-14. Limitation with bounding box-based collision

1. The `BoundingBox` object introduced in the previous example does not account for rotation. This is a well-known limitation for axes-aligned bounding boxes: although the approach is computationally efficient, it does not support rotation.
2. The two objects do not actually collide. The fact that the bounds of two objects overlap does not automatically equate to the two objects colliding.

In this project, you will see how to achieve per-pixel-accurate collision detection, pixel-accurate collision, or per-pixel collision, which identifies the overlapping of nontransparent pixels of two colliding objects. However, keep in mind that this is not an end-all solution. While the collision detection is precise, the trade-off is potential performance costs. As an image becomes larger and more complex, it also has more pixels that need to be checked for collisions. This is in contrast to the constant computation cost required for bounding box collision detection from the previous example.

The Per-Pixel Collisions Project

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 in Figure 6-15. The source code to this project is defined in the Chapter6/6.4.PerPixelCollisions folder.



No Collision

Figure 6-15. Running the Per-Pixel Collisions project

The controls of the project are as follows:

- *Arrow keys*: Move the small textured object, the Portal minion
- *WASD keys*: Move the large textured object, the Collector minion

The goals of the project are as follows:

- To demonstrate how to detect nontransparent pixel overlap
- To understand the pros and cons of using per-pixel-accurate collision detection

You can find the following external resources in the assets folder: the fonts folder that contains the default system fonts, `minion_collector.png`, `minion_portal.png`, and `minion_sprite.png`. Note that `minion_collector.png` is a large, 1024x1024 image, while `minion_portal.png` is a small, 64x64 image; `minion_sprite.png` defines the DyePack sprite element.

Overview of Per-Pixel Collision Algorithm

Before moving forward, it is important to identify the requirements for detecting a collision between two textured objects. Foremost is that the texture itself needs to contain an area of transparency in order for this type of collision detection to provide any increase in accuracy. Without transparency in the texture, you can and should use a simple bounding box collision detection. If one or both of the textures contain transparent areas, then you'll need to handle two cases of collision. The first case is to check whether the bounds of the two objects collide. You can see this reflected in Figure 6-16. Notice how the bounds of the objects overlap yet none of the nontransparent colored pixels are touching.

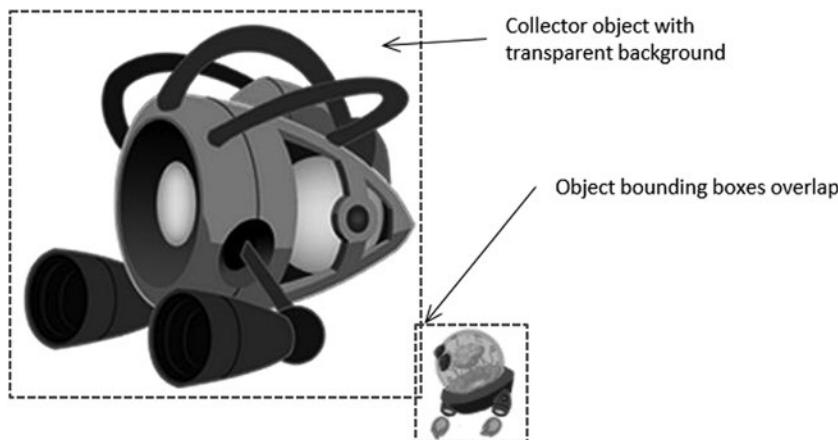


Figure 6-16. Overlapping bounding boxes without actual collision

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.

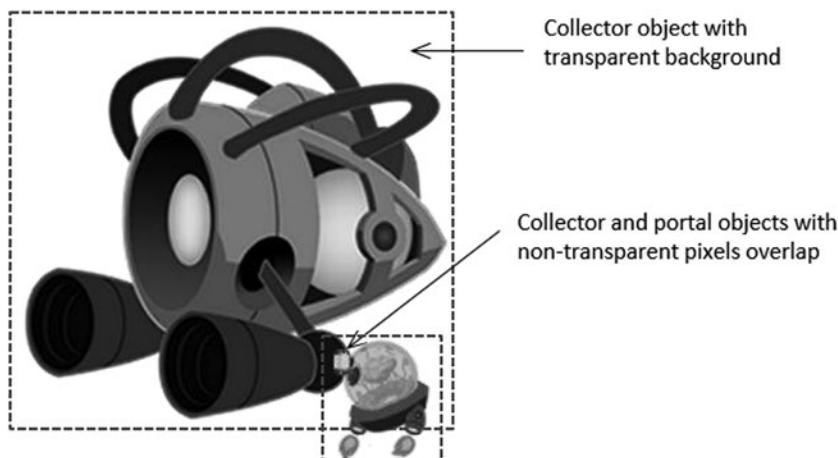


Figure 6-17. Pixel collision occurring between the large texture and the small texture

Now that the problem is clearly defined, here is the logic or pseudocode for per-pixel-accurate collision detection:

```
Given two images, Image-A and Image-B
If the bounds of the object of Image-A and Image-B collide then
    For each Pixel-A in Image-A
        pixelCameraSpace = Pixel-A position in camera space
        Transform pixelCameraSpace to Image-B space
        Read Pixel-B from Image-B
        If Pixel-A and Pixel-B are not both completely transparent then
            A collision has occurred
```

The per-pixel transformation to Image-B space from `pixelCameraSpace` is required because collision checking must be carried out within the same coordinate space. Additionally, pay attention to the runtime of this algorithm. Each pixel within Image-A must be checked, so the runtime is $O(N)$, where N is equal to the number of pixels in Image-A, or Image-A's resolution. To mitigate this performance hit, you should use the smaller of the two images (the portal minion in this case) as Image-A. However, at this point, you can probably see why the performance of pixel-accurate collision detection is concerning. Checking for these collisions during every update with many high-resolution textures onscreen can quickly bog down performance. You are now ready to examine the implementation of per-pixel accurate collision.

Modify Engine_Textures to Load a Texture as an Array of Color

Recall that the `Engine_Texture` component reads image files from the server file system, loads the images to the WebGL context, and processes the images into WebGL textures. In this way, there is no actual storage of the file texture in the game engine. To support per-pixel collision detection, the color information must be retrieved from the WebGL context. The `Engine_Texture` component can be modified to support this requirement.

1. In the `Engine_Textures.js` file, expand the `TextureInfo` object to include a new variable for storing the color array of a file texture.

```
function TextureInfo(name, w, h, id) {
    this.mName = name;
    this.mWidth = w;
    this.mHeight = h;
    this.mGLTexID = id;
    this.mColorArray = null;
}
```

2. Define a function to retrieve the color array from the WebGL context and remember to add this new function to the public interface of `Engine_Texture`.

```
var getColorArray = function (textureName) {
    var texInfo = getTextureInfo(textureName);
    if (texInfo.mColorArray === null) {
        // create a framebuffer, bind it to the texture, and read the color content
        var gl = gEngine.Core.getGL();
        var fb = gl.createFramebuffer();
        gl.bindFramebuffer(gl.FRAMEBUFFER, fb);
        gl.framebufferTexture2D(gl.FRAMEBUFFER, gl.COLOR_ATTACHMENT0, gl.TEXTURE_2D,
            texInfo.mGLTexID, 0);
```

```

if (gl.checkFramebufferStatus(gl.FRAMEBUFFER) === gl.FRAMEBUFFER_COMPLETE) {
    var pixels = new Uint8Array(texInfo.mWidth * texInfo.mHeight * 4);
    gl.readPixels(0, 0, texInfo.mWidth, texInfo.mHeight, gl.RGBA,
        gl.UNSIGNED_BYTE, pixels);
    texInfo.mColorArray = pixels;
} else {
    alert("WARNING: Engine.Textures.GetColorArray() failed!");
}

gl.bindFramebuffer(gl.FRAMEBUFFER, null);
gl.deleteFramebuffer(fb);
}
return texInfo.mColorArray;
};

var mPublic = {
    loadTexture: loadTexture,
    unloadTexture: unloadTexture,
    activateTexture: activateTexture,
    deactivateTexture: deactivateTexture,
    getTextureInfo: getTextureInfo,
    getColorArray: getColorArray
};

```

The previous code creates a WebGL FRAMEBUFFER, fills the buffer with the desired texture, and retrieves the buffer content into the associated `texInfo.mColorArray`.

Modify TextureRenderable to Support Per-Pixel Collision

The `TextureRenderable` object is the most appropriate for implementing the per-pixel collision functionality. This is because the `TextureRenderable` object is the base class for all classes that render textures. Implementation in this base class means all subclasses can inherit the functionality with minimal additional changes.

1. Edit the `TextureRenderable.js` file and modify the constructor to add instance variables to hold texture information for supporting per-pixel collision detection and for later subclass overrides:

```

function TextureRenderable(myTexture) {
    Renderable.call(this);
    Renderable.prototype.setColor.call(this, [1, 1, 1, 0]);
    Renderable.prototype._setShader.call(this, gEngine.DefaultResources.getTextureShader());
    this.mTexture = null;
    // these two instance variables are to cache texture information
    // for supporting per-pixel accurate collision
    this.mTextureInfo = null;
    this.mColorArray = null;
    // defined for subclass to override
    this.mTexWidth = 0;
    this.mTexHeight = 0;
}

```

```

this.mTexLeftIndex = 0;
this.mTexBottomIndex = 0;
this.setTexture(myTexture);
}

```

2. Modify the `setTexture()` function to initialize the instance variables accordingly.

```

TextureRenderable.prototype.setTexture = function (newTexture) {
    this.mTexture = newTexture;
    this.mTextureInfo = gEngine.Textures.getTextureInfo(newTexture);
    this.mColorArray = null;
    this.mTexWidth = this.mTextureInfo.mWidth;
    this.mTexHeight = this.mTextureInfo.mHeight;
    this.mTexLeftIndex = 0;
    this.mTexBottomIndex = 0;
};

```

Note that the `mTexWidth/Height` variables are defined for later subclass overrides such that the algorithm can support the collision of sprite elements.

Create TextureRenderable_PixelCollision.js File to Implement Per-Pixel Collision

To maintain the readability of the source code, create a new file in the `src/Engine/Renderables/` folder and name it `TextureRenderable_PixelCollision.js`. All per-pixel collision functionality for the `TextureRenderable` object will be implemented in the `TextureRenderable_PixelCollision.js` file. Remember to include commands to load this new file in `index.html`.

Note Source files with names in the form of `object_functionality.js` implement functionality for `object`. By separating complex implementations of related functionality into separate files, the readability of each source code file can be maintained.

1. Edit the `TextureRenderable_PixelCollision.js` file and define a function to set the `mColorArray`.

```

TextureRenderable.prototype.setColorArray = function () {
    if (this.mColorArray === null)
        this.mColorArray = gEngine.Textures.getColorArray(this.mTexture);
};

```

2. Define a function to return the alpha value, or the transparency, of any given pixel (x, y) .

```

TextureRenderable.prototype._pixelAlphaValue = function (x, y) {
    x = x * 4;
    y = y * 4;
    return this.mColorArray[(y * this.mTextureInfo.mWidth) + x + 3];
};

```

Notice that the `mColorArray` is a one-dimensional array where colors of pixels are stored as four floats and organized by rows.

3. Define a function to compute the WC position (`returnWCPos`) of a given pixel (i, j).

```
TextureRenderable.prototype._indexToWCPosition = function (returnWCPos, i, j) {
    var x = i * this.mXform.getWidth() / (this.mTexWidth - 1);
    var y = j * this.mXform.getHeight() / (this.mTexHeight - 1);
    returnWCPos[0] = this.mXform.getXPos() + (x - (this.mXform.getWidth() * 0.5));
    returnWCPos[1] = this.mXform.getYPos() + (y - (this.mXform.getHeight() * 0.5));
};
```

4. Now, implement the inverse of the previous function and use a WC position (`wcPos`) to compute the texture pixel indices (`returnIndex`).

```
TextureRenderable.prototype._wcPositionToIndex = function (returnIndex, wcPos) {
    // use wcPos to compute the corresponding returnIndex[0 and 1]
    var delta = [];
    vec2.sub(delta, wcPos, this.mXform.getPosition());
    returnIndex[0] = this.mTexWidth * (delta[0] / this.mXform.getWidth());
    returnIndex[1] = this.mTexHeight * (delta[1] / this.mXform.getHeight());

    // recall that xForm.getPosition() returns center, yet Texture origin is at lower-left corner!
    returnIndex[0] += this.mTexWidth / 2;
    returnIndex[1] += this.mTexHeight / 2;

    returnIndex[0] = Math.floor(returnIndex[0]);
    returnIndex[1] = Math.floor(returnIndex[1]);
};
```

5. Now, it is possible to implement the outlined per-pixel collision algorithm.

```
TextureRenderable.prototype.pixelTouches = function(other, wcTouchPos) {
    var pixelTouch = false;
    var xIndex = 0, yIndex;
    var otherIndex = [0, 0];

    while ((!pixelTouch) && (xIndex < this.mTexWidth)) {
        yIndex = 0;
        while ((!pixelTouch) && (yIndex < this.mTexHeight)) {
            if (this._pixelAlphaValue(xIndex, yIndex) > 0) {
                this._indexToWCPosition(wcTouchPos, xIndex, yIndex);
                other._wcPositionToIndex(otherIndex, wcTouchPos);
                if ((otherIndex[0] > 0) && (otherIndex[0] < other.mTexWidth) &&
                    (otherIndex[1] > 0) && (otherIndex[1] < other.mTexHeight)) {
                    pixelTouch = other._pixelAlphaValue(otherIndex[0], otherIndex[1]) > 0;
                }
            }
            yIndex++;
        }
        xIndex++;
    }
    return pixelTouch;
};
```

The parameter `other` is a reference to the other `TextureRenderable` object that is being tested for collision. If pixels do overlap between the objects, the returned value of `wcTouchPos` is the first detected colliding position in the WC space. Notice that the nested loops terminate as soon as one pixel overlap is detected or when `pixelTouch` becomes true. This is an important feature for efficiency concerns. However, this also means that the returned `wcTouchPos` is simply one of many potentially colliding points.

Support GameObject Per-Pixel Collision in `GameObject_PixelCollision.js`

The `GameObject` class must be modified to support per-pixel collision. Create a new file in the `src/Engine/GameObjects/` folder and name it `GameObject_PixelCollision.js`. As always, remember to include the loading of this new file in `index.html`. This new file will implement the support for per-pixel collision of the `GameObject` class.

```
GameObject.prototype.pixelTouches = function (otherObj, wcTouchPos) {
    // only continue if both objects have GetColorArray defined
    var pixelTouch = false;
    var myRen = this.getRenderable();
    var otherRen = otherObj.getRenderable();

    if ((typeof myRen.pixelTouches === "function") &&
        (typeof otherRen.pixelTouches === "function")) {
        var otherBbox = otherObj.getBBox();
        if (otherBbox.intersectsBound(this.getBBox())) {
            myRen.setColorArray();
            otherRen.setColorArray();
            pixelTouch = myRen.pixelTouches(otherRen, wcTouchPos);
        }
    }
    return pixelTouch;
};
```

This function checks to ensure that the objects are colliding and delegates the actual per-pixel collision to `TextureRenderable` objects. Notice the `intersectsBound()` function for a bounding box intersection check before invoking the potentially expensive `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` instances: the large Collector minion, the small Portal minion, and the DyePack. The Collector and Portal minions are controlled by arrow and WASD keys, respectively. The detail of the implementation of `MyGame` is similar to previous projects and is not shown. The only noteworthy code fragment is the collision testing in the `update()` function, as shown here:

```
MyGame.prototype.update = function () {
    var msg = "No Collision";

    this.mPortal.update(gEngine.Input.keys.W, gEngine.Input.keys.S,
                       gEngine.Input.keys.A, gEngine.Input.keys.D);
    this.mCollector.update(gEngine.Input.keys.Up, gEngine.Input.keys.Down,
                          gEngine.Input.keys.Left, gEngine.Input.keys.Right);
```

```

var h = [];
// Portal's resolution is 1/16 that of Collector!
// if (this.mCollector.pixelTouches(this.mPortal, h)) { // VERY EXPENSIVE!!
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);
};

```

You can now test the collision accuracy by moving the two minions and intersecting them at different locations (for example, 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 position. 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 $3\hat{j}$.

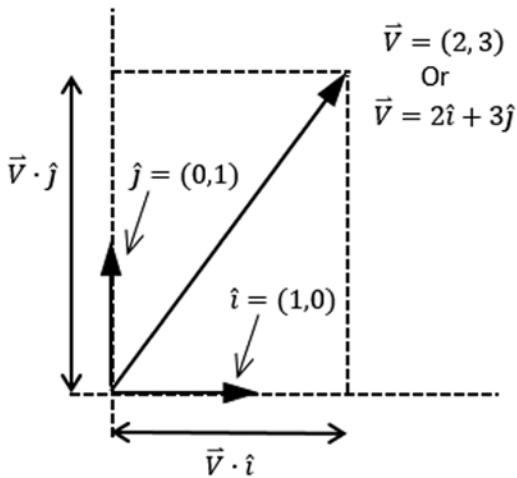


Figure 6-18. The decomposition of vector \vec{V}

With that in mind, given the normalized perpendicular component vectors \hat{L} and \hat{M} and any vector \vec{V} , the following formulas will always be true. You can see a representation of this principle in Figure 6-19.

$$\vec{V} = (\vec{V} \cdot \hat{i}) \hat{i} + (\vec{V} \cdot \hat{j}) \hat{j}$$

$$\vec{V} = (\vec{V} \cdot \hat{L}) \hat{L} + (\vec{V} \cdot \hat{M}) \cdot \hat{M}$$

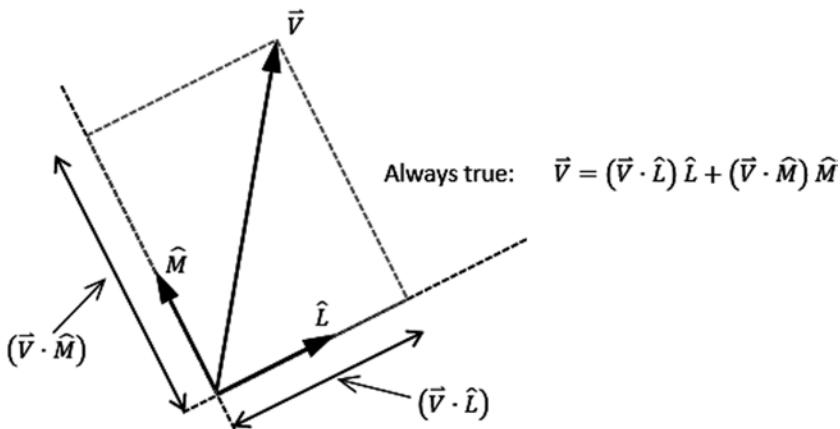


Figure 6-19. Decomposing a vector by two normalized component vectors

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.

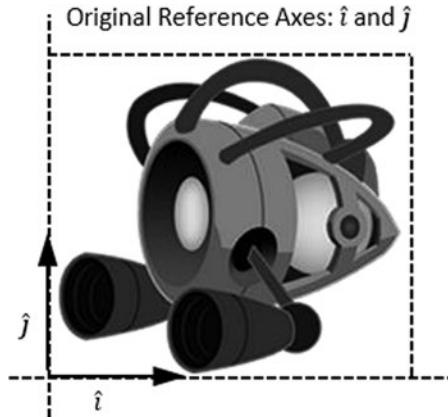


Figure 6-20. An axes-aligned texture

However, after the image has been rotated, the reference vector set no longer resides along the x/y-axes. Therefore, the collision computation must take into account the newly rotated axes \hat{L} and \hat{M} , as shown in Figure 6-21.

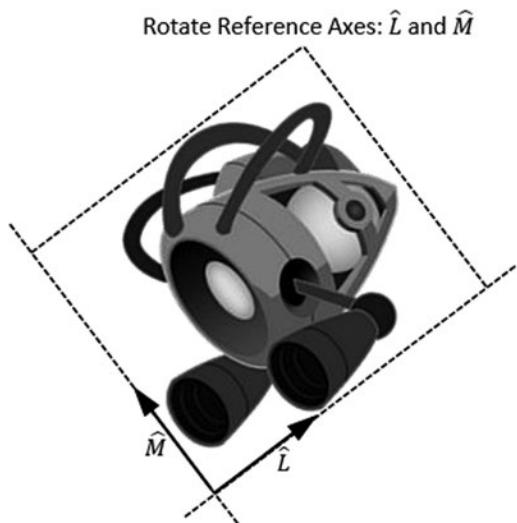


Figure 6-21. A rotated texture and its component vectors

The General Pixel Collisions Project

This project demonstrates how to detect a collision between two rotated TextureRenderable objects with per-pixel accuracy. As in the previous project, a yellow DyePack object (as a test confirmation) will be displayed at the detected colliding position. You can see an example of this project running in Figure 6-22. The source code to this project is defined in the Chapter6/6.5.GeneralPixelCollisions folder.

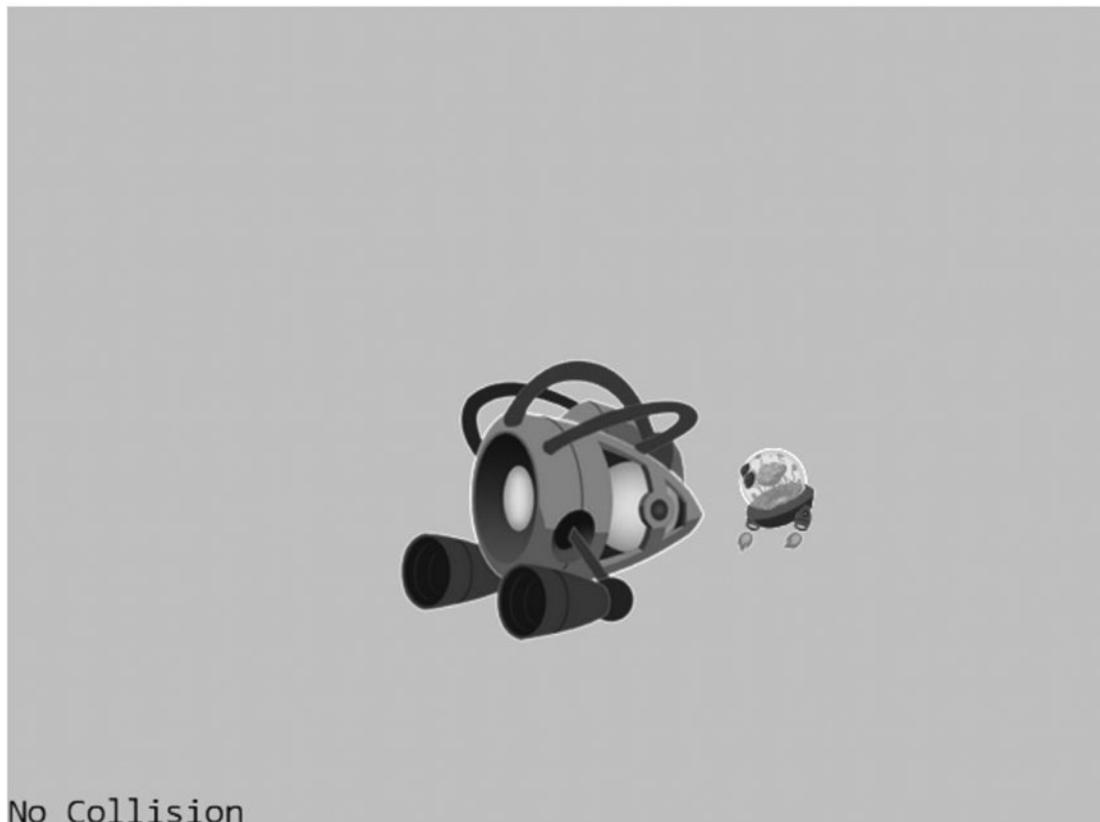


Figure 6-22. Running the General Pixel Collisions project

The controls of the project are as follows:

- *Arrow keys*: Move the small textured object, the Portal minion
- *P key*: Rotates the small textured object, the Portal minion
- *WASD keys*: Move the large textured object, the Collector minion
- *E key*: Rotates the large textured object, the Collector minion

The goals of the project are as follows:

- To access pixels of a rotated image via vector decomposition
- To support per-pixel accurate collision detection between two rotated textured objects

You can find the same external resource files as in the previous project in the assets folder.

Modify TextureRenderable_PixelCollision to Support Rotation

1. Edit the `TextureRenderable_PixelCollision.js` file and modify the `_indexToWCPosition()` function accordingly.

```
TextureRenderable.prototype._indexToWCPosition = function (returnWCPos, i, j, xDir, yDir) {
    var x = i * this.mXform.getWidth() / (this.mTexWidth - 1);
    var y = j * this.mXform.getHeight() / (this.mTexHeight - 1);
    var xDisp = x - (this.mXform.getWidth() * 0.5);
    var yDisp = y - (this.mXform.getHeight() * 0.5);
    var xDirDisp = [];
    var yDirDisp = [];

    vec2.scale(xDirDisp, xDir, xDisp);
    vec2.scale(yDirDisp, yDir, yDisp);
    vec2.add(returnWCPos, this.mXform.getPosition(), xDirDisp);
    vec2.add(returnWCPos, returnWCPos, yDirDisp);
};
```

In the previous code, `xDir` and `yDir` are the \hat{L} and \hat{M} normalized component vectors. The variables `xDisp` and `yDisp` are the displacements to be offset along the `xDir` and `yDir`, respectively. The returned value of `returnWCPos` is a simple displacement from the object's center position along the `xDirDisp` and `yDirDisp` vectors, the scaled `xDir` and `yDir` vectors.

2. In a similar fashion, modify the `_wcPositionToIndex()` function to support the rotated normalized vector components.

```
TextureRenderable.prototype._wcPositionToIndex = function (returnIndex, wcPos, xDir, yDir) {
    // use wcPos to compute the corresponding returnIndex[0 and 1]
    var delta = [];
    vec2.sub(delta, wcPos, this.mXform.getPosition());
    var xDisp = vec2.dot(delta, xDir);
    var yDisp = vec2.dot(delta, yDir);
    returnIndex[0] = this.mTexWidth * (xDisp / this.mXform.getWidth());
    returnIndex[1] = this.mTexHeight * (yDisp / this.mXform.getHeight());

    // recall that xForm.getPosition() returns lower-left corner, yet
    // Texture origin is at lower-left corner!
    returnIndex[0] += this.mTexWidth / 2;
    returnIndex[1] += this.mTexHeight / 2;

    returnIndex[0] = Math.floor(returnIndex[0]);
    returnIndex[1] = Math.floor(returnIndex[1]);
};
```

3. The `pixelTouches()` function needs to be modified to compute the rotated normalized component vectors.

```
TextureRenderable.prototype.pixelTouches = function(other, wcTouchPos) {
    var pixelTouch = false;
    var xIndex = 0, yIndex;
    var otherIndex = [0, 0];
```

```

var xDir = [1, 0];
var yDir = [0, 1];
var otherXDir = [1, 0];
var otherYDir = [0, 1];
vec2.rotate(xDir, xDir, this.mXform.getRotationInRad());
vec2.rotate(yDir, yDir, this.mXform.getRotationInRad());
vec2.rotate(otherXDir, otherXDir, other.mXform.getRotationInRad());
vec2.rotate(otherYDir, otherYDir, other.mXform.getRotationInRad());

while ((!pixelTouch) && (xIndex < this.mTexWidth)) {
    yIndex = 0;
    while ((!pixelTouch) && (yIndex < this.mTexHeight)) {
        if (this._pixelAlphaValue(xIndex, yIndex) > 0) {
            this._indexToWCPosition(wcTouchPos, xIndex, yIndex,
                xDir, yDir);
            other._wcPositionToIndex(otherIndex, wcTouchPos,
                otherXDir, otherYDir);
            if ((otherIndex[0] > 0) && (otherIndex[0] < other.mTexWidth) &&
                (otherIndex[1] > 0) && (otherIndex[1] < other.mTexHeight)) {
                pixelTouch = other._pixelAlphaValue (otherIndex[0], otherIndex[1]) > 0;
            }
        }
        yIndex++;
    }
    xIndex++;
}
return pixelTouch;
};

```

The variables `xDir` and `yDir` are the rotated normalized component vectors \hat{L} and \hat{M} of this `TextureRenderable` object, while `otherXDir` and `otherYDir` are those of the colliding object. These vectors are used as references for computing transformations from texture index to WC and from WC to texture index.

Modify the `GameObject_PixelCollision.js` to Support Rotation

Recall that the `GameObject` class first tests for the bounding-box collision between two objects before it actually invokes the much more expensive per-pixel collision computation. As illustrated in Figure 6-14, the `BoundingBox` object does not support object rotation correctly, and the following code remedies this shortcoming:

```

GameObject.prototype.pixelTouches = function (otherObj, wcTouchPos) {
    // only continue if both objects have getColorArray defined
    // if defined, should have other texture intersection support!
    var pixelTouch = false;
    var myRen = this.getRenderable();
    var otherRen = otherObj.getRenderable();

    if ((typeof myRen.pixelTouches === "function") &&
        (typeof otherRen.pixelTouches === "function")) {
        if ((myRen.getXform().getRotationInRad() === 0) &&
            (otherRen.getXform().getRotationInRad() === 0)) {

```

```

// no rotation, we can use bbox ...
var otherBbox = otherObj.getBBox();
if (otherBbox.intersectsBound(this.getBBox())) {
    myRen.setColorArray();
    otherRen.setColorArray();
    pixelTouch = myRen.pixelTouches(otherRen, wcTouchPos);
}
} else {
    // One or both are rotated, compute an encompassing circle by using the
    // hypotenuse as radius
    var mySize = myRen.getXform().getSize();
    var otherSize = otherRen.getXform().getSize();
    var myR = Math.sqrt(0.5*mySize[0]*0.5*mySize[0] + 0.5*mySize[1]*0.5*mySize[1]);
    var otherR = Math.sqrt(0.5*otherSize[0]*0.5*otherSize[0] +
                           0.5*otherSize[1]*0.5*otherSize[1]);
    var d = [];
    vec2.sub(d, myRen.getXform().getPosition(), otherRen.getXform().getPosition());
    if (vec2.length(d) < (myR + otherR)) {
        myRen.setColorArray();
        otherRen.setColorArray();
        pixelTouch = myRen.pixelTouches(otherRen, wcTouchPos);
    }
}
return pixelTouch;
};

```

The previous 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. The per-pixel collision detection is invoked only if the distance between these two circles is less than the sum of the radii.

Test Generalized Per-Pixel Collision

The code for testing the rotated TextureRenderable objects is essentially identical to that from the previous project, with the exception of the two added controls for rotations. The details of the implementation are not shown. You can now run the project, rotate the two objects, and observe the accurate collision results.

Per-Pixel Collisions for Sprites

The previous project implicitly assumes that the Renderable object is covered by the entire texture map. This assumption means that the per-pixel collision implementation does not support sprite or animated sprite objects. In this section, you will see how to remedy this deficiency.

The Sprite Pixel Collisions Project

This project demonstrates how to move an animated sprite object around the screen and perform per-pixel collision detection with other objects. The project tests for the correctness between collisions of `TextureRenderable`, `SpriteRenderable`, and `SpriteAnimateRenderable` objects. You can see an example of this project running in Figure 6-23. The source code to this project is defined in the Chapter6/6.6.SpritePixelCollisions folder.



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 follow:

- To implement per-pixel collision detection for sprite and animated sprite objects

You can find the following external resource file in the assets folder: the fonts folder that contains the default system fonts and `minion_sprite.png`, and `minion_portal.png`.

Create SpriteRenderable_PixelCollision.js file to Implement Per-Pixel Collision

Create a new file in the `src/Engine/Renderables/` folder and name it `SpriteRenderable_PixelCollision.js`. This file implements the per-pixel specific support for `SpriteRenderable` objects. Remember to include commands to load this new file in `index.html`. In this file, define the `_setTexInfo()` function to override instance variables defined in the superclass, `TextureRenderable`, such that these variables identify the currently active sprite element.

```
SpriteRenderable.prototype._setTexInfo = function () {
    var imageW = this.mTextureInfo.mWidth;
    var imageH = this.mTextureInfo.mHeight;

    this.mTexLeftIndex = this.mTexLeft * imageW;
    this.mTexBottomIndex = this.mTexBottom * imageH;

    this.mTexWidth = ((this.mTexRight - this.mTexLeft) * imageW) + 1;
    this.mTexHeight = ((this.mTexTop - this.mTexBottom) * imageH) + 1;
};
```

Notice that instead of the dimension of the entire texture map, `mTexWidth/Height` now contain values that correspond to the dimension of a single sprite element in the sprite sheet.

Modify Existing SpriteRenderable Functions to Support Per-Pixel Collision

Follow these steps:

1. Edit the `SpriteRenderable.js` file and modify the constructor to call the newly defined `_setTexInfo()` function to properly record the dimension of a sprite element.

```
function SpriteRenderable(myTexture) {
    TextureRenderable.call(this, myTexture);
    Renderable.prototype._setShader.call(this, gEngine.DefaultResources.getSpriteShader());
    this.mTexLeft = 0.0; // bounds of texture coordinate (0 is left, 1 is right)
    this.mTexRight = 1.0; //
    this.mTexTop = 1.0; // 1 is top and 0 is bottom of image
    this.mTexBottom = 0.0; //
    this._setTexInfo();
}
```

2. Remember to call the `_setTexInfo()` function when the current sprite element is updated in the `setElementUVCoordinate()` and `setElementPixelPositions()` functions.

```
SpriteRenderable.prototype.setElementUVCoordinate = function (left, right, bottom, top) {
    this.mTexLeft = left;
    this.mTexRight = right;
    this.mTexBottom = bottom;
    this.mTexTop = top;
    this._setTexInfo();
};
```

```
SpriteRenderable.prototype.setElementPixelPositions = function (left, right, bottom, top) {
    var imageW = this.mTextureInfo.mWidth;
    var imageH = this.mTextureInfo.mHeight;
    this.mTexLeft = left / imageW;
    this.mTexRight = right / imageW;
    this.mTexBottom = bottom / imageH;
    this.mTexTop = top / imageH;
    this._setTexInfo();
};
```

Modify TextureRenderable to Support Accesses to Sprite Pixels

Edit the `TextureRenderable_PixelCollision.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.mTexBottomIndex;
    x += this.mTexLeftIndex;
    x = x * 4;
    y = y * 4;
    return this.mColorArray[(y * this.mTextureInfo.mWidth) + x + 3];
};
```

Test Per-Pixel Collision for Sprites in MyGame

The code for testing this project is a simple modification from previous projects, and the details are not listed. It is important to note the different object types in the scene.

- *Portal minion*: A simple `TextureRenderable` object
- *Hero and Brain*: `SpriteRenderable` objects where the textures shown on the geometries are sprite elements defined in the `minion_sprite.png` sprite sheet
- *Left and Right minions*: `SpriteAnimateRenderable` objects with sprite elements defined in the top two rows of the `minion_sprite.png` animated sprite sheet

You can now run this project and observe the correct results from the collisions of the different object types.

1. Try moving the Hero object and observe how the Brain object constantly seeks out and collides with it. This is the case of collision between two `SpriteRenderable` objects.
2. Press the L/R keys and then move the Portal minion with the WASD keys to collide with the Left or Right minions. Remember that you can rotate the Portal minion with the P key. This is the case of collision between `TextureRenderable` and `SpriteAnimatedRenderable` objects.
3. Type the H key and then move the Portal minion to collide with the Hero object. This is the case of collision between `TextureRenderable` and `SpriteRenderable` objects.
4. Type the B key and then move the Portal minion to collide with the Brain object. This is the case of collision between rotated `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 simpler and better organized control logic in the client MyGame test levels. You reviewed vectors in 2D space. A vector is defined by its direction and magnitude. Vectors are convenient for describing defined displacements (velocities). You reviewed some foundational vector operations, including normalization of a vector and how to calculate 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 becomes prominent as the objects' behaviors increase in sophistication. The axes-aligned bounding boxes were introduced as a crude and yet computationally efficient solution for approximating object collisions. You learned the algorithm for per-pixel-accurate collision detection and that it is accurate but computationally expensive. 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 execute the pixel iteration process based on the texture with lower resolution. When implementing pixel-accurate collision, you began with tackling the basic case of working with 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 and enhances the functionality of the `Camera` object. You will learn to control and manipulate the `Camera` object and work with multiple `Cameras` in the same game.

Game Design Considerations

Chapters 1–5 introduced foundation techniques for drawing, moving, and animating objects on the screen. The Scene Objects project from Chapter 4 described a simple interaction behavior and showed you how to change the game screen based on the location of a rectangle. Recall that moving the rectangle to the left boundary caused the level to visually change, and the Audio Support project added contextual sound to reinforce the overall sense of presence. Although it's possible to build an intriguing (albeit simple) puzzle game using only the elements from Chapters 1–5, things get much more interesting when you can integrate object detection and collision triggers; these behaviors form the basis for many common game mechanics and provide opportunities to design a wide range of interesting gameplay options.

Starting with the Game Objects project, you can see how the screen elements start working together to convey the game setting. Even with the interaction in this project limited to character movement, the setting is beginning to resolve into something that feels coherent. The hero character appears to be flying through a moving scene populated by a number of mechanized robots; there's also a small object in the center of the screen that you might imagine could become some kind of special pickup.

Even at this basic stage of development, it's possible to brainstorm game mechanics that could potentially form the foundation for a full game. If you were designing a simple game mechanic based on only the screen elements found in the Game Object Abstraction project, what kind of behaviors would you choose and what kind of interactions would you require? As one example, imagine that the hero character must avoid colliding with the flying robots and that perhaps some of the robots will detect and follow the hero; maybe the hero is also "zapped" if she comes into contact with a robot. Imagine also that the small

object in the center of the screen allows the hero to be invincible for a certain duration. With these few basic interactions, you have opened opportunities to explore mechanics and levels typically associated with familiar game genres, 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 Object Abstraction 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 11 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, and keeping yourself limited to just the elements and interactions covered through Chapter 6 is actually a beneficial exercise; design constraints often help the creative process by shaping and guiding your ideas.

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. Detection allows you to begin including more robust cause-and-effect 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 versus the precise but resource-intensive per-pixel detection method. There are situations where the bounding-box approach is sufficient, but if players perceive collisions to be arbitrary because the bounding boxes are too removed 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 casual chains around a new object being produced as the result of two objects colliding (for example, 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 reviewed 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 relevant to many genres 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. 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.

CHAPTER 7



Manipulating the Camera

After completing this chapter, you will be able to:

- Implement operations that are commonly employed by 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 Canvas Coordinate space to the World Coordinate (WC) space
- Program with mouse input in a game environment with multiple cameras

Introduction

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 pleasant game objects with well-structured behaviors.

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 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 `setUpViewProjection()` function encapsulates the details and enables the drawing of all game objects inside the WC window bounds to be displayed in the corresponding viewport.

Note In this book, the WC window or WC bounds are used to refer to the WC window bounds.

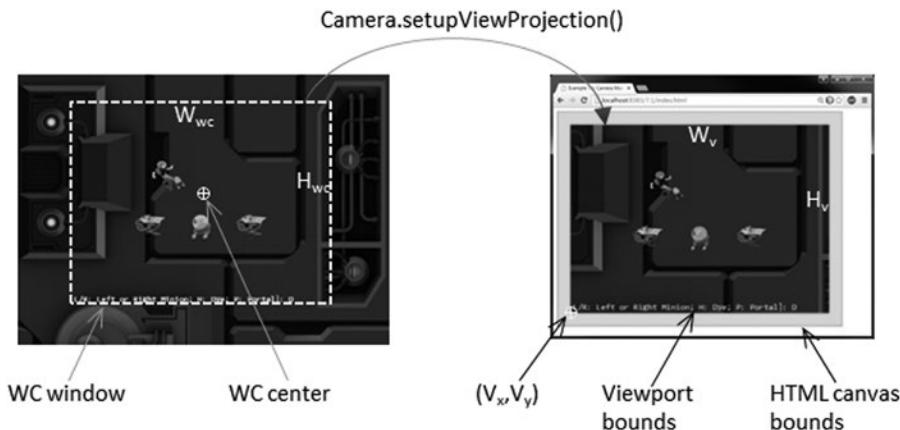


Figure 7-1. Review of WC parameters that define a Camera object

The Camera object abstraction allows the game programmer to ignore the details of WC bounds and the HTML canvas and focus on designing a fun and entertaining gameplay experience. Programming with a Camera object in a game level should reflect the use of a physical video camera in the real world. For example, you may want to pan the video camera to show your audiences the environment, you may want to attach the video camera on an actress and share her journey with your audience, or you may want to play the role of director and instruct the actors in your scene to stay within the visual ranges of the video camera. The distinct characteristics of these examples, such as panning or following a character's view, are the high-level functional specifications. Notice that in the real world 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.

Camera Manipulations

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.

The Camera Manipulations Project

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.CameraManipulations folder.



Figure 7-2. Running the Camera Manipulations project

The controls of the project are as follows:

- *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.
- *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.

The goals of the project are as follows:

- To experience some of the common camera manipulation operations
- To understand the mapping from manipulation operations to the corresponding camera parameter values that must be altered
- To implement camera manipulation operations

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 represented by `bg.png`.

Organize the Source Code

To accommodate the increase in functionality and the complexity of the Camera object, you will create a separate folder for storing the Camera object implementation and all supporting operations' source code files.

- Create a new folder called Cameras in `src/Engine`. Move the `Camera.js` file into this folder, and remember to update the reference path in `index.html`.

Support Clamping to Camera WC Bounds

Edit `Camera.js` and define a function to clamp the bounds associated with a `Transform` object to the camera WC bound.

```
Camera.prototype.clampAtBoundary = function (aXform, zone) {
    var status = this.collideWCBound(aXform, zone);
    if (status !== BoundingBox.eboundCollideStatus.eInside) {
        var pos = aXform.getPosition();
        if ((status & BoundingBox.eboundCollideStatus.eCollideTop) !== 0)
            pos[1] = (this.getWCCenter())[1] + (zone * this.getWCHeight() / 2)
                    - (aXform.getHeight() / 2);
        if ((status & BoundingBox.eboundCollideStatus.eCollideBottom) !== 0)
            pos[1] = (this.getWCCenter())[1] - (zone * this.getWCHeight() / 2)
                    + (aXform.getHeight() / 2);
        if ((status & BoundingBox.eboundCollideStatus.eCollideRight) !== 0)
            pos[0] = (this.getWCCenter())[0] + (zone * this.getWCWidth() / 2)
                    - (aXform.getWidth() / 2);
        if ((status & BoundingBox.eboundCollideStatus.eCollideLeft) !== 0)
            pos[0] = (this.getWCCenter())[0] - (zone * this.getWCWidth() / 2)
                    + (aXform.getWidth() / 2);
    }
    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. Once again, 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

As discussed in the previous chapter, to maintain the readability of source code files, related functions of objects are grouped into separate source code files.

1. Create a new file in the `src/Engine/Cameras` folder and name it `Camera_Manipulation.js`.
2. Edit this file to define functions to pan, or move, the camera by an offset and to a new WC center.

```
Camera.prototype.panBy = function (dx, dy) {
    this.mWCCenter[0] += dx;
    this.mWCCenter[1] += dy;
};

Camera.prototype.panTo = function (cx, cy) {
    this.setWCCenter(cx, cy);
};
```

3. Define a function to pan the camera based on the bounds of a `Transform` object.

```
Camera.prototype.panWith = function (aXform, zone) {
    var status = this.collideWCBound(aXform, zone);
    if (status !== BoundingBox.eboundCollideStatus.eInside) {
        var pos = aXform.getPosition();
        var newC = this.getWCCenter();
        if ((status & BoundingBox.eboundCollideStatus.eCollideTop) !== 0)
            newC[1] = pos[1] + (aXform.getHeight() / 2) -
                (zone * this.getWCHeight() / 2);
        if ((status & BoundingBox.eboundCollideStatus.eCollideBottom) !== 0)
            newC[1] = pos[1] - (aXform.getHeight() / 2) +
                (zone * this.getWCHeight() / 2);
        if ((status & BoundingBox.eboundCollideStatus.eCollideRight) !== 0)
            newC[0] = pos[0] + (aXform.getWidth() / 2) -
                (zone * this.getWCWidth() / 2);
        if ((status & BoundingBox.eboundCollideStatus.eCollideLeft) !== 0)
            newC[0] = pos[0] - (aXform.getWidth() / 2) +
                (zone * this.getWCWidth() / 2);
    }
};
```

The `panWidth()` function is complementary to the `clampAtBoundary()` function, where instead of changing the `aXform` position, the camera is moved to ensure the proper inclusion of the `aXform` bounds. As in the case of the `clampAtBoundary()` function, the camera will not be changed if the `aXform` bounds are completely outside the tested WC bounds area.

- Define functions to zoom the camera with respect to the center or a target position.

```
Camera.prototype.zoomBy = function (zoom) {
    if (zoom > 0)
        this.mWCWidth *= zoom;
};

Camera.prototype.zoomTowards = function (pos, zoom) {
    var delta = [];
    vec2.sub(delta, pos, this.mWCCenter);
    vec2.scale(delta, delta, zoom - 1);
    vec2.sub(this.mWCCenter, this.mWCCenter, delta);
    this.zoomBy(zoom);
};
```

The `zoomBy()` function zooms with respect to the center of the camera, and the `zoomTowards()` function zooms with respect to a world coordinate position. If the `zoom` variable is greater than 1, the WC window size becomes larger, and you will see more of the world and, thus, zoom out. The `zoom` value of less than 1 zooms in. Figure 7-3 shows the results of `zoom=0.5` for zooming with respect to the center of WC and with respect to the position of the Hero object.

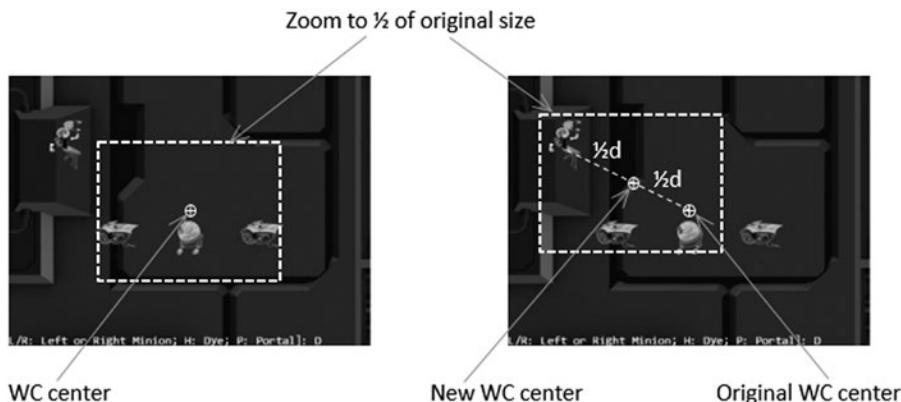


Figure 7-3. Zooming toward the WC Center and toward a target position

Manipulating the Camera in MyGame

There are two important functionality to be tested: panning and zooming. Once again, with the exception of the `update()` function, the majority of the code in the `MyGame.js` file is similar to the previous projects and are not repeated. The `update()` function is modified from the previous project to manipulate the camera.

```
MyGame.prototype.update = function () {
    // ... identical code to previous project ...
```

```

// Pan camera to object
if (gEngine.Input.isKeyClicked(gEngine.Input.keys.L)) {
    this.mFocusObj = this.mLMinion;
    this.mChoice = 'L';
    this.mCamera.panTo(this.mLMinion.getXform().getXPos(),
        this.mLMinion.getXform().getYPos());
}
if (gEngine.Input.isKeyClicked(gEngine.Input.keys.R)) {
    this.mFocusObj = this.mRMinion;
    this.mChoice = 'R';
    this.mCamera.panTo(this.mRMinion.getXform().getXPos(),
        this.mRMinion.getXform().getYPos());
}
if (gEngine.Input.isKeyClicked(gEngine.Input.keys.P)) {
    this.mFocusObj = this.mPortal;
    this.mChoice = 'P';
}
if (gEngine.Input.isKeyClicked(gEngine.Input.keys.H)) {
    this.mFocusObj = this.mHero;
    this.mChoice = 'H';
}

// zoom
if (gEngine.Input.isKeyClicked(gEngine.Input.keys.N))
    this.mCamera.zoomBy(1 - zoomDelta);
if (gEngine.Input.isKeyClicked(gEngine.Input.keys.M))
    this.mCamera.zoomBy(1 + zoomDelta);
if (gEngine.Input.isKeyClicked(gEngine.Input.keys.J))
    this.mCamera.zoomTowards(this.mFocusObj.getXform().getPosition(), 1 - zoomDelta);
if (gEngine.Input.isKeyClicked(gEngine.Input.keys.K))
    this.mCamera.zoomTowards(this.mFocusObj.getXform().getPosition(), 1 + zoomDelta);

// interaction with the WC bounds
this.mCamera.clampAtBoundary(this.mBrain.getXform(), 0.9);
this.mCamera.clampAtBoundary(this.mPortal.getXform(), 0.8);
this.mCamera.panWith(this.mHero.getXform(), 0.9);

    this.mMsg.setText(msg + this.mChoice);
};


```

In the previous code, the first four if statements select the in-focus object, where L and R keys also recenter the camera by calling the `panTo()` function with the appropriate WC positions. The second four if statements control the zoom, whether toward the WC center or toward the current in-focus object. The function ends with clamping the Brain and Portal objects to within 90 percent and 80 percent of the WC bounds, respectively, and panning the camera based on the transform (or position) of the Hero object.

You can now run the project and move the Hero object with the WASD keys. Move the Hero object toward the WC bounds to observe the camera being pushed. Continue pushing the camera with the Hero object; notice that because of the `clampAtBoundary()` function call, the Portal object will in turn be pushed such that it never leaves the camera WC bounds. Now press the L/R key to observe the camera center switching to the center on the Left or Right minion. The N/M keys demonstrate straightforward zooming with respect to the center. To experience zooming with respect to a target, move the Hero object toward the

top left of the canvas and then press the H key to select it as the zoom focus. Now, with your mouse pointer pointing at the head of the Hero object, you can press the K key to zoom out first and then the J key to zoom back in. Notice that as you zoom, all objects in the scene change positions except the areas around the Hero object. This is a convenient functionality to support zooming into a desired region of your game. 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 even 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 sudden appearance of a completely different world is not only visually unpleasant; it can also cause player confusion.

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, an 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 .

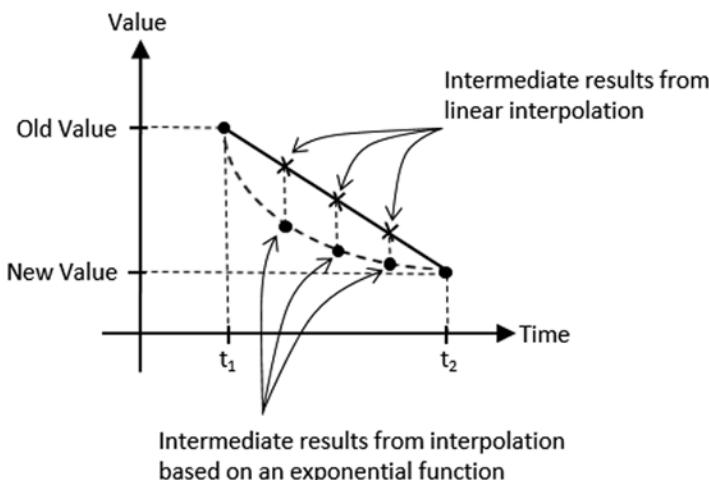


Figure 7-4. Interpolating values based on linear and exponential functions

Figure 7-4 shows that there is more than one way to interpolate values over time. For example, linear interpolation computes intermediate results according to the slope of the line connecting the old and new values. In contrast, an exponential function may compute intermediate results based on percentages from previous values. Linear interpolation interpolates from the old to the new values in constant steps, whereas the shown exponential function will result in rapid changes initially followed by much slower changes. In this way, with linear interpolation, a camera position would move from an old to new position with a constant speed similar to a constant-speed camera that is panning. In comparison, the interpolation based on the given exponential function would move the camera position rapidly at the beginning, with the motion slowing down quickly over time giving a sensation of focusing the camera on a new target.

This section introduces the `Interpolate` and `InterpolateVec2` utility objects to support smooth and gradual camera movements resulting from camera manipulation operations.

The Camera Interpolations Project

This project demonstrates the smoother and visually more pleasing 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.CameraInterpolations folder.



Figure 7-5. Running the Camera Interpolations project

The controls of the project 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.
- *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 understand interpolated results between given values
- To implement interpolation supporting gradual camera parameter changes
- To experience interpolated changes in camera parameters

You can find the same external resource files as in the previous project in the assets folder.

Interpolation as an Utility

Similar to `Transform` objects supporting transformation functionality and `BoundingBox` objects supporting collision detection, an `Interpolate` object can be defined to support interpolation of values. To maintain source code organization, a new folder should be defined to store these utility objects.

- Create the new folder called `src/Engine/Utils` and move the `Transform.js` and `BoundingBox.js` files into this folder. Remember to update the reference path in `index.html`.

The Interpolate Object

Define the `Interpolate` object to compute interpolation between two values.

1. Create a new file in the `src/Engine/Utils/` folder and name it `Interpolate.js`. Add the following code to construct the object:

```
function Interpolate(value, cycles, rate) {
    this.mCurrentValue = value;      // begin value of interpolation
    this.mFinalValue = value;        // final value of interpolation
    this.mCycles = cycles;
    this.mRate = rate;

    // if there is a new value to interpolate to, number of cycles
    // left for interpolation
    this.mCyclesLeft = 0;
}
```

This object interpolates from `mCurrentValue` to `mFinalValue` in `mCycles`. During each update, intermediate results are computed based on the `mRate` increment on the difference between `mCurrentValue` and `mFinalValue`, as shown next.

- Define the function that computes the intermediate results.

```
Interpolate.prototype._interpolateValue = function () {
    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`. However, since during each iteration the `mCurrentValue` is updated to the computed intermediate result, over the entire interpolation cycle `mCurrentValue` changes to the `mFinalValue` following an exponential function.

- Define relevant getter and setter functions.

```
Interpolate.prototype.getValue = function () { return this.mCurrentValue; };
Interpolate.prototype.configInterpolation = function (stiffness, duration) {
    this.mRate = stiffness;
    this.mCycles = duration;
};

Interpolate.prototype.setFinalValue = function (v) {
    this.mFinalValue = v;
    this.mCyclesLeft = this.mCycles; // will trigger interpolation
};
```

- Define the function to trigger the computation of each intermediate result.

```
Interpolate.prototype.updateInterpolation = function () {
    if (this.mCyclesLeft <= 0)
        return;
    this.mCyclesLeft--;
    if (this.mCyclesLeft === 0)
        this.mCurrentValue = this.mFinalValue;
    else
        this._interpolateValue();
};
```

The InterpolateVec2 Object

Since many of the camera parameters are `vec2` objects (for example, the WC center position), it is important to generalize the `Interpolate` object to support the interpolation of `vec2` objects.

- Create a new file in the `src/Engine/Utils/` folder and name it `InterpolateVec2.js`. Add the following to construct the object as a child of `Interpolate`:

```
function InterpolateVec2(value, cycle, rate) {
    Interpolate.call(this, value, cycle, rate);
}
gEngine.Core.inheritPrototype(InterpolateVec2, Interpolate);
```

2. Override the `_interpolateValue()` function to compute intermediate results for `vec2`.

```
InterpolateVec2.prototype._interpolateValue = function () {
    vec2.lerp(this.mCurrentValue, this.mCurrentValue, this.mFinalValue,
              this.mRate);
};
```

The `vec2.lerp()` function, defined in the `gl-matrix.js` file, computes for each of the x and y components of `vec2` with identical calculations as the `_interpolateValue()` function in the `Interpolate` object.

Represent Interpolated Intermediate Results with the CameraState Object

The state of a `Camera` object must be generalized to support gradual changes of interpolated intermediate results. The `CameraState` object is introduced to accomplish this purpose.

1. Create a new file in the `src/Engine/Cameras/` folder and name it `CameraState.js`. Add the following code to construct the object:

```
function CameraState(center, width) {
    this.kCycles = 300; // number of cycles to complete the transition
    this.kRate = 0.1; // rate of change for each cycle
    this.mCenter = new InterpolateVec2(center, this.kCycles, this.kRate);
    this.mWidth = new Interpolate(width, this.kCycles, this.kRate);
}
```

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 `Interpolate` class and are capable of interpolating and computing intermediate results that achieve gradual changes.

2. Define the setting and getting functions for the center and width.

```
CameraState.prototype.getCenter = function ()
    { return this.mCenter.getValue(); };
CameraState.prototype.getWidth = function ()
    { return this.mWidth.getValue(); };

CameraState.prototype.setCenter = function (c)
    { this.mCenter.setFinalValue(c); };
CameraState.prototype.setWidth = function (w)
    { this.mWidth.setFinalValue(w); };
```

3. Define the update function to trigger the interpolation computation.

```
CameraState.prototype.updateCameraState = function () {
    this.mCenter.updateInterpolation();
    this.mWidth.updateInterpolation();
};
```

- Define the function to configure the interpolation.

```
CameraState.prototype.configInterpolation = function (stiffness, duration) {
    this.mCenter.configInterpolation(stiffness, duration);
    this.mWidth.configInterpolation(stiffness, duration);
};
```

The `stiffness` variable defines how quickly the interpolated intermediate results should converge to the final value. This is a number between 0 to 1, where a 0 means the convergence will never happen and a 1 means instantaneous convergence. The `duration` variable defines in how many update cycles the convergence should take place. This must be a positive integer value.

Integrate Interpolation into Camera Manipulation Operations

The `Camera` object must be modified to represent the WC center and width using the newly defined `CameraState` object.

- Modify the `Camera` constructor to replace the center and width variables with an instance of `CameraState`.

```
function Camera(wcCenter, wcWidth, viewportArray) {
    // WC and viewport position and size
    this.mCameraState = new CameraState(wcCenter, wcWidth);
    this.mViewport = viewportArray; // [x, y, width, height]
    this.mNearPlane = 0;
    this.mFarPlane = 1000;

    // transformation matrices
    this.mViewMatrix = mat4.create();
    this.mProjMatrix = mat4.create();
    this.mVPMMatrix = mat4.create();

    // background color
    this.mBgColor = [0.8, 0.8, 0.8, 1]; // RGB and Alpha
}
```

The rest of the changes in the `Camera.js` file are trivial replacements of accessing and setting the center and width through the `this.mCameraState` instance variable and are not shown.

- Now, edit the `Camera_Manipulation.js` file to define the functions to update and configure the interpolation functionality of the `CameraState` object.

```
Camera.prototype.update = function () {
    this.mCameraState.updateCameraState();
};

Camera.prototype.configInterpolation = function (stiffness, duration) {
    this.mCameraState.configInterpolation(stiffness, duration);
};
```

3. Modify the `panBy()` camera manipulation function to support the `CameraState` object as follows:

```
Camera.prototype.panBy = function (dx, dy) {
    var newC = vec2.clone(this.getWCCenter());
    this.mWCCenter[0] += dx;
    this.mWCCenter[1] += dy;
    this.mCameraState.setCenter(newC);
};
```

Similar to the `panBy()` function, the rest of the changes in the `Camera_Manipulation.js` file are trivial replacements to access and set WC window center and width via the `this.mCameraState` instance variable and are not shown.

Testing Interpolations 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.

- Modify the `MyGame update()` function.

```
MyGame.prototype.update = function () {
    var zoomDelta = 0.05;
    var msg = "L/R: Left or Right Minion; H: Dye; P: Portal]: ";
    this.mCamera.update(); // for smoother camera movements
    //... Identical to previous project ...
};
```

The previous call to update the camera for computing interpolated intermediate results is the only change in the `MyGame.js` file. You can now run the project and experiment with the smooth and gradual changes resulting from camera manipulation operations. Notice that the uninterrupted interpolated results mean the rendered image never abruptly changes and the sense of continuation in space from before and after the user's camera manipulation commands is preserved. You can try changing the `stiffness` and `duration` variables to better appreciate the different rates of interpolation convergence.

Camera Shake Effect

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.

Consider how a camera shake may occur in a real-life situation. For instance, while shooting with a video camera, say you are surprised or startled by someone or something could collide with you. Your reaction will probably be slight disorientation followed by quickly refocusing on the original targets of shooting. From the perspective of the camera, this reaction can be described as initial large displacements from the original camera center followed by quick adjustments to recenter the camera. Mathematically, as illustrated in Figure 7-6, damped simple harmonic motions, which can be represented with the damping of trigonometric functions, can be used to describe these types of displacements. However, in contrast to the slightly chaotic and unpredictable randomness of human reactions, straight mathematic formulation is precise, with perfect predictability. For this reason, a pseudorandom simple harmonic function will be introduced to model the camera shake effect.

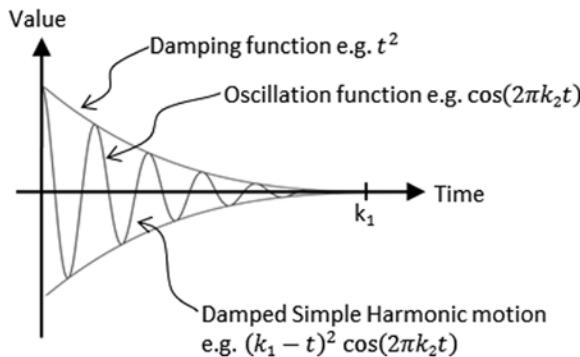


Figure 7-6. The displacements of a damped simple harmonic motion

The Camera Shake Project

This project demonstrates how to implement camera shake as a pseudorandom damped simple harmonic motion. You can see an example of this project running in Figure 7-7. This project is identical to the previous project except for the added command to create the camera shake effect. The source code to this project is defined in the Chapter7/7.3.CameraShake folder.



Figure 7-7. Running the Camera Shake project

The following is the new control of this project:

- *Q key:* Initiates the camera shake effect

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.
- *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 insights into modeling displacements with simple mathematical functions
- To experience with the camera shake effect
- To implement camera shake as a pseudorandom damped simple harmonic motion

You can find the same external resource files as in the previous project in the assets folder.

Create the ShakePosition Class to Model the Shaking of a Position

As discussed, the shaking of a camera is described by the movements of its center position. For this reason, the implementation of shake effects should begin with modeling the shaking of a position.

1. Create a new file in the `src/Engine/Utils/` folder and name it `ShakePosition.js`. Add the following code to construct the object:

```
// xDelta, yDelta: how large a shake
// shakeFrequency: how much movement
// shakeDuration: for how long in number of cycles
function ShakePosition(xDelta, yDelta, shakeFrequency, shakeDuration) {
    this.mXMag = xDelta;
    this.mYMag = yDelta;
    this.mCycles = shakeDuration; // number of cycles to complete the transition
    this.mOmega = shakeFrequency * 2 * Math.PI; // Converts frequency to radians
    this.mNumCyclesLeft = shakeDuration;
}
```

The `xDelta` and `yDelta` variables represent the initial displacements before damping, in WC space. The `shakeFrequency` parameter specifies how much to oscillate with a value of 1 representing one complete period of a cosine function. The `shakeDuration` parameter defines how long to shake, in units of game loop updates.

2. Define the damped simple harmonic motion.

```
ShakePosition.prototype._nextDampedHarmonic = function () {
    // computes (Cycles) * cos( Omega * t )
    var frac = this.mNumCyclesLeft / this.mCycles;
    return frac * frac * Math.cos((1 - frac) * this.mOmega);
};
```

The `frac` variable is a ratio of the number of cycles left in the shake (`mNumCyclesLeft`) to the total number of cycles the camera should shake (`mCycles`). This value decreases from 1 to 0 as `mNumCyclesLeft` decreases from `mCycles` to 0. Figure 7-8 illustrates the damped simple harmonic motion that governs the camera shake displacements.

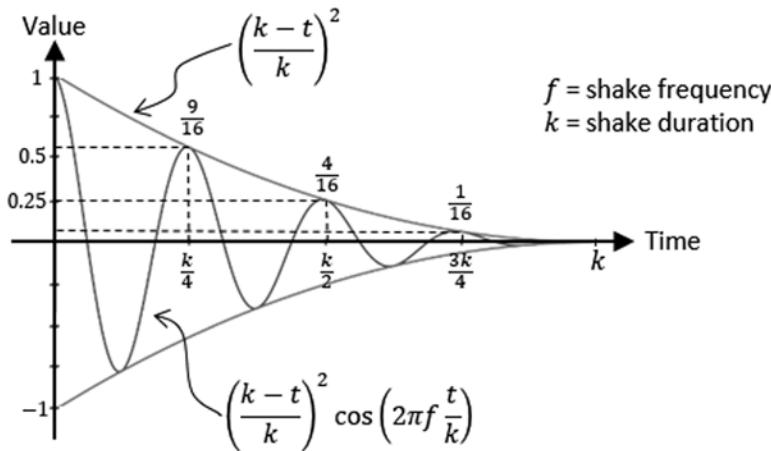


Figure 7-8. The damped simple harmonic motion that specifies camera shake

3. Define a function to check for the status of camera shake.

```
ShakePosition.prototype.shakeDone = function () {
    return (this.mNumCyclesLeft <= 0);
};
```

4. Define a function to trigger the calculation of shake displacements.

```
ShakePosition.prototype.getShakeResults = function () {
    this.mNumCyclesLeft--;
    var c = [];
    var fx = 0;
    var fy = 0;
    if (!this.shakeDone()) {
        var v = this._nextDampedHarmonic();
        fx = (Math.random() > 0.5) ? -v : v;
        fy = (Math.random() > 0.5) ? -v : v;
    }
    c[0] = this.mXMag * fx;
    c[1] = this.mYMag * fy;
    return c;
};
```

For the same `mNumCyclesLeft`, the call to the `_nextDampedHarmonic()` function will return the same value. The call to the `Math.random()` function introduces pseudorandomness into the x and y displacements.

Define the CameraShake Object to Abstract Camera Shaking Effect

With the defined ShakePoint object, it is convenient to trace the displacements of a pseudorandom damped simple harmonic motion. However, the Camera object requires an additional abstraction layer.

1. Create a new file in the `src/Engine/Cameras/` folder and name it `CameraShake.js`. Add the following code to construct the object:

```
function CameraShake(state, xDelta, yDelta, shakeFrequency, shakeDuration) {
    this.mOrgCenter = vec2.clone(state.getCenter());
    this.mShakeCenter = vec2.clone(this.mOrgCenter);
    this.mShake = new ShakePosition(xDelta, yDelta, shakeFrequency,
        shakeDuration);
}
```

The CameraShake object receives the center of a camera via the `CameraState` parameter (`state`). This center is kept as the origin of shake displacements.

2. Define the function that triggers the displacement computation for accomplishing the shaking effect.

```
CameraShake.prototype.updateShakeState = function () {
    var s = this.mShake.getShakeResults();
    vec2.add(this.mShakeCenter, this.mOrgCenter, s);
};
```

3. Define a function to check whether the shaking effect has completed.

```
CameraShake.prototype.shakeDone = function () {
    return this.mShake.shakeDone();
};
```

4. Define the getter and setting functions for the camera center.

```
CameraShake.prototype.getCenter = function () { return this.mShakeCenter; };
CameraShake.prototype.setRefCenter = function (c) {
    this.mOrgCenter[0] = c[0];
    this.mOrgCenter[1] = c[1];
};
```

Modify the Camera to Support Shake Effect

With the proper `CameraShake` abstraction, supporting the shaking of the camera simply means defining, using, initiating, and updating the shake effect.

1. Modify the `Camera` constructor to define a `CameraShake` object.

```
function Camera(wcCenter, wcWidth, viewportArray) {
    // WC and viewport position and size
    this.mCameraState = new CameraState(wcCenter, wcWidth);
    this.mCameraShake = null;
```

```

this.mViewport = viewportArray; // [x, y, width, height]
this.mNearPlane = 0;
this.mFarPlane = 1000;

// transformation matrices
this.mViewMatrix = mat4.create();
this.mProjMatrix = mat4.create();
this.mVPMatrix = mat4.create();

// background color
this.mBgColor = [0.8, 0.8, 0.8, 1]; // RGB and Alpha
}

```

2. Modify step B1 of the `setupViewProjection()` function to use the `CameraShake` object's center if it is defined.

```

// Initializes the camera to begin drawing
Camera.prototype.setupViewProjection = function () {
    var gl = gEngine.Core.getGL();
    // Step A1-A4: Set up and clear the Viewport

    // Identical to previous project ...

    // Step B: Set up the View-Projection transform operator
    // Step B1: define the view matrix
    var center = [];
    if (this.mCameraShake !== null) {
        center = this.mCameraShake.getCenter();
    } else {
        center = this.getWCCenter();
    }
    mat4.lookAt(this.mViewMatrix,
        [center[0], center[1], 10], // WC center
        [center[0], center[1], 0], // 
        [0, 1, 0]); // orientation

    // Identical to previous project ...

    // Step B2: define the projection matrix
    // Step B3: concatenate view and project matrices
};

```

3. Modify the `Camera_Manipulation.js` file to add support to initiate the shake effect.

```

Camera.prototype.shake = function (xDelta, yDelta, shakeFrequency, duration) {
    this.mCameraShake = new CameraShake(this.mCameraState, xDelta, yDelta,
        shakeFrequency, duration);
};

```

4. Continue working with the Camera_Manipulation.js file and modify the Camera object update() function to trigger a camera shake update if one is defined.

```
Camera.prototype.update = function () {
    if (this.mCameraShake !== null) {
        if (this.mCameraShake.shakeDone()) {
            this.mCameraShake = null;
        } else {
            this.mCameraShake.setRefCenter(this.getWCCenter());
            this.mCameraShake.updateShakeState();
        }
    }
    this.mCameraState.updateCameraState();
};
```

Testing the Camera Shake Effect in MyGame

When comparing to the previous project, MyGame.js file adds only a few lines in the update() function: to trigger camera shake effect with the Q key.

```
MyGame.prototype.update = function () {
    // Identical to previous project ...

    if (gEngine.Input.isKeyPressed(gEngine.Input.keys.Q))
        this.mCamera.shake(-2, -2, 20, 30);
    this.mMsg.setText(msg + this.mChoice);
};
```

You can now run the project and experience with the pseudorandom damped simple harmonic motion that simulates the camera shake effect. 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 to the mCamera.shake() function to experiment with different shake configurations. Recall that the first two parameters control the initial shake displacements, and the third and fourth parameters are the shakeFrequency and shakeDuration that control the shake amplitude and how long the shake effect 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 prompting a view of the enemy boss to warn the player of what is to come.

In your game engine, the Camera object abstracts the WC window of the game world to draw from and the viewport for the area on the canvas to draw to. This effective abstraction supports the multiple view idea with multiple Camera objects. Each view in the game can simply be handled with a separate Camera object with distinct WC window and viewport configurations.

The Multiple Cameras Project

This project demonstrates how to represent multiple views into 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.MultipleCameras folder.



Figure 7-9. Running the Multiple Cameras project

The controls of the project are identical to the previous project.

- *Q key:* Initiates the camera shake effect.
- *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.

- *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.

The goals of the project are as follows:

- To understand the camera abstraction for presenting views into the game world
- To experience working with multiple cameras in the same game level
- To appreciate the importance of interpolation configuration for cameras with a specific purpose

You can find the same external resource files as in the previous project 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 would allow easy differentiation of camera views on the canvas.

1. Modify the Camera constructor to allow programmers to define a bound-number of pixels to surround the viewport of the camera.

```
function Camera(wcCenter, wcWidth, viewportArray, bound) {
    // WC and viewport position and size
    this.mCameraState = new CameraState(wcCenter, wcWidth);
    this.mCameraShake = null;

    this.mViewport = []; // [x, y, width, height]
    this.mViewportBound = 0;
    if (bound !== undefined) {
        this.mViewportBound = bound;
    }
    this.mScissorBound = []; // use for bounds
    this.setViewport(viewportArray, this.mViewportBound);
    this.mNearPlane = 0;
    this.mFarPlane = 1000;

    // ... Identical to previous projects ...
}
```

By default, bound is assumed to be zero, and the camera will draw to the entire mViewport. Please refer to the `setViewport()` function that follows. A nonzero bound instructs the camera to leave bound-number of pixels that surround the camera mViewport in the background color, thereby allowing easy differentiation of multiple viewports on a canvas.

2. Define the `setViewport()` function.

```
Camera.prototype.setViewport = function (viewportArray, bound) {
    if (bound === undefined) {
        bound = this.mViewportBound;
    }
    // [x, y, width, height]
    this.mViewport[0] = viewportArray[0] + bound;
    this.mViewport[1] = viewportArray[1] + bound;
    this.mViewport[2] = viewportArray[2] - (2 * bound);
    this.mViewport[3] = viewportArray[3] - (2 * bound);
    this.mScissorBound[0] = viewportArray[0];
    this.mScissorBound[1] = viewportArray[1];
    this.mScissorBound[2] = viewportArray[2];
    this.mScissorBound[3] = viewportArray[3];
};
```

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. 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. Modify 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.

```
Camera.prototype.getViewport = function () {
    var 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 `setupViewProjection()` function to bind scissor bounds with `mScissorBound` instead of the viewport bounds.

```
Camera.prototype.setupViewProjection = function () {
    var gl = gEngine.Core.getGL();
    // ... Identical to previous projects ...
```

```

// Step A2: set up the corresponding scissor area to limit the clear area
gl.scissor(this.mScissorBound[0],      // x pos of bottom-left of the area to be drawn
           this.mScissorBound[1],      // y pos of bottom-left of the area to be drawn
           this.mScissorBound[2],      // width of the area to be drawn
           this.mScissorBound[3]);    // height of the area to be drawn

// ... Identical to previous projects ...
};

```

Testing Multiple Cameras in MyGame

The `MyGame` level must create multiple cameras, configure them properly, and draw each independently. For ease of demonstration, two new `Camera` objects will be created, one to focus on the `Hero` object and one to focus on the chasing `Brain` object. As in the previous examples, the implementation of the `MyGame` level is largely identical. In this example, some portions of the `initialize()`, `draw()`, and `update()` functions are modified to handle the multiple `Camera` objects and are highlighted here:

1. Modify the `initialize()` function to define and configure three `Camera` objects.

```

MyGame.prototype.initialize = function () {
    // Step A: set up the cameras
    this.mCamera = new Camera(
        vec2.fromValues(50, 36), // 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]);

    this.mHeroCam = new Camera(
        vec2.fromValues(50, 30), // will be updated at each cycle to point to hero
        20,
        [490, 330, 150, 150],
        2                      // viewport bounds
    );
    this.mHeroCam.setBackgroundColor([0.5, 0.5, 0.5, 1]);
    this.mBrainCam = new Camera(
        vec2.fromValues(50, 30), // will be updated at each cycle to point to the brain
        10,
        [0, 330, 150, 150],
        2                      // viewport bounds
    );
    this.mBrainCam.setBackgroundColor([1, 1, 1, 1]);
    this.mBrainCam.configInterpolation(0.7, 10);

    // ... Identical to previous projects ...
};

```

Both the `mHeroCam` and `mBrainCam` define a 2-pixel boundary for their viewports, with the `mHeroCam` boundary defined to be gray (the background color) and with `mBrainCam` white. Notice the `mBrainCam` object's stiff interpolation setting informing the camera interpolation to converge to new values in ten cycles.

2. Define a helper function to draw the world that is common to all three cameras.

```
MyGame.prototype.drawCamera = function (camera) {
    camera.setupViewProjection();
    this.mBg.draw(camera);
    this.mHero.draw(camera);
    this.mBrain.draw(camera);
    this.mPortal.draw(camera);
    this.mLMinion.draw(camera);
    this.mRMinion.draw(camera);
};
```

3. Modify the MyGame object draw() function to draw all three cameras.

```
MyGame.prototype.draw = function () {
    // Step A: clear the canvas
    gEngine.Core.clearCanvas([0.9, 0.9, 0.9, 1.0]); // clear to light gray

    // Step B: Draw with all three cameras
    this.drawCamera(this.mCamera);
    this.mMsg.draw(this.mCamera); // only draw status in the main camera
    this.drawCamera(this.mHeroCam);
    this.drawCamera(this.mBrainCam);
};
```

Take note of the mMsg object only being drawn to the mCamera, the main camera. For this reason, the echo message will appear only in the viewport of the main camera.

4. Modify the update() function to pan the mHeroCam and mBrainCam with the corresponding objects and to move the mHeroCam viewport continuously.

```
MyGame.prototype.update = function () {
    var zoomDelta = 0.05;
    var msg = "L/R: Left or Right Minion; H: Dye; P: Portal]:";

    this.mCamera.update(); // for smoother camera movements
    this.mHeroCam.update();
    this.mBrainCam.update();

    // ... Identical to previous projects ...

    // set the hero and brain cams
    this.mHeroCam.panTo(this.mHero.getXform().getXPos(),
                        this.mHero.getXform().getYPos());
    this.mBrainCam.panTo(this.mBrain.getXform().getXPos(),
                        this.mBrain.getXform().getYPos());

    // Move the hero cam viewport just to show it is possible
    var v = this.mHeroCam.getViewport();
    v[0] += 1;
```

```

    if (v[0] > 500) {
        v[0] = 0;
    }
    this.mHeroCam.setViewport(v);

    this.mMsg.setText(msg + this.mChoice);
};

}

```

You can now run the project and notice the three different viewports showing on the HTML canvas. The 2-pixel-wide bounds around the `mHeroCam` and `mBrainCam` 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` object `draw()` function, the `mBrainCam` is drawn last. The last drawn 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.configInterpolation()` 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 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. 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 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 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:

- $\text{mouseDCX} = \text{mouseX} - V_x$
- $\text{mouseDCY} = \text{mouseY} - V_y$

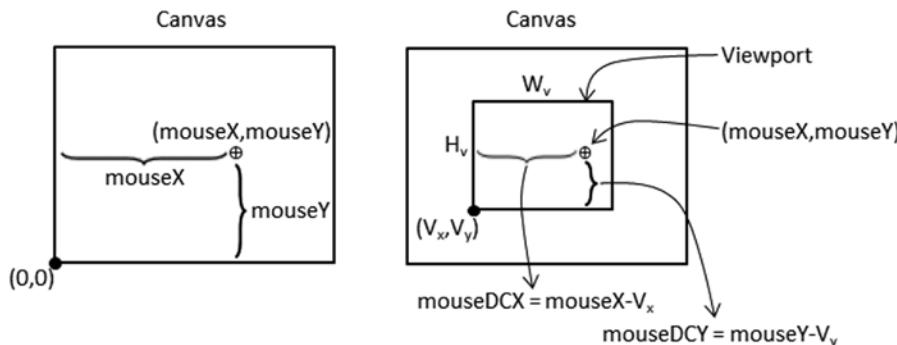


Figure 7-10. Mouse position on canvas and viewport

In this way, $(\text{mouseDCX}, \text{mouseDCY})$ is the offset from the (V_x, V_y) , the lower-left corner of the viewport.

The drawing on the left of Figure 7-11 shows that the Device Coordinate (DC) space defines a pixel position within a viewport with offsets measured with respect to the lower-left corner of the viewport. For this reason, the DC space is also referred to as the pixel space. The computed $(\text{mouseDCX}, \text{mouseDCY})$ position is an example of a position in DC space. The drawing on the right of Figure 7-11 shows that this position can be transformed into the WC space with the lower-left corner located at $(\text{minWCX}, \text{minWCY})$ and a dimension of $W_{wc} \times H_{wc}$ according to these formulae:

- $\text{mouseWCX} = \text{minWCX} + \left(\text{mouseDCX} \times \frac{W_{wc}}{W_v} \right)$
- $\text{mouseWCY} = \text{minWCY} + \left(\text{mouseDCY} \times \frac{H_{wc}}{H_v} \right)$

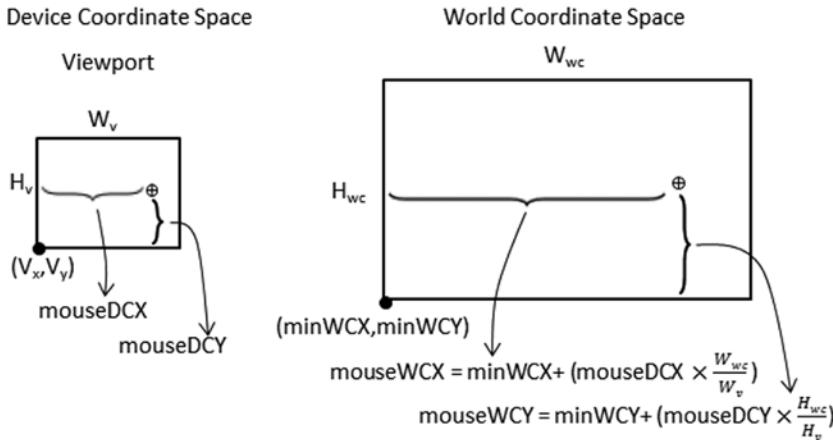


Figure 7-11. Mouse position in viewport DC space and WC space

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.

The Mouse Input Project

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.MouseInput folder.

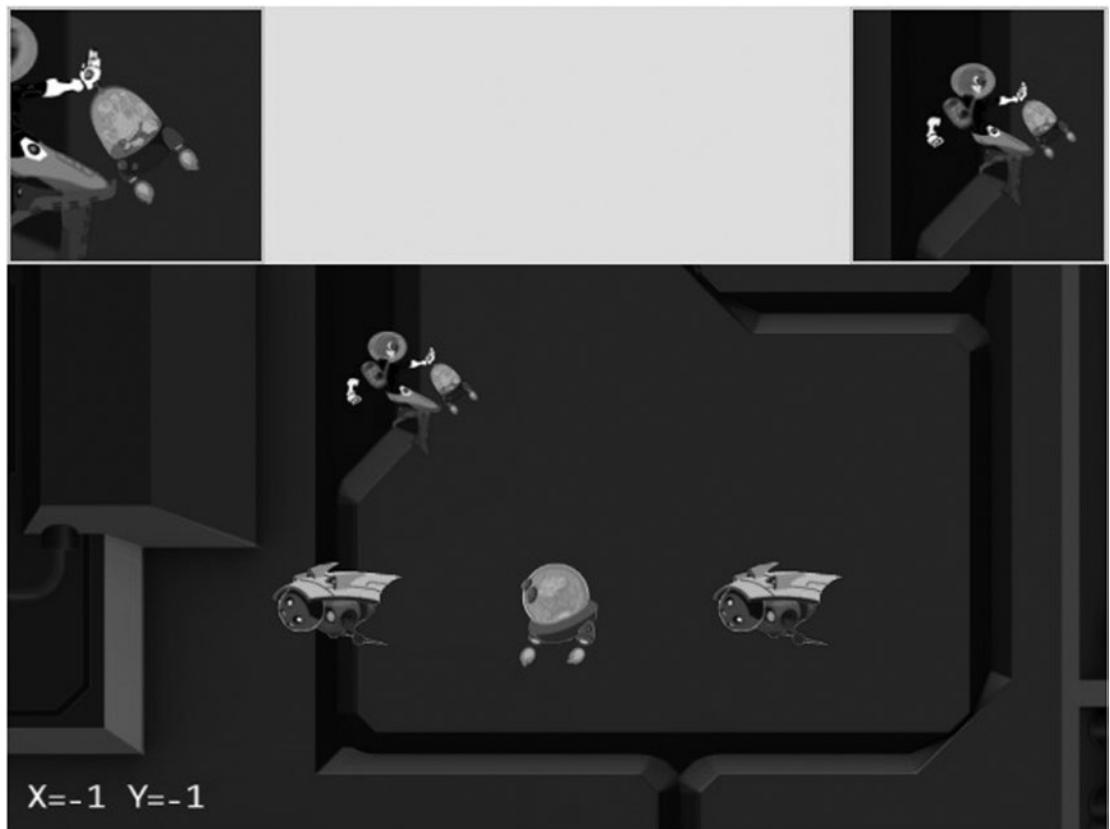


Figure 7-12. Running the Mouse Input project

The new controls of this project are as follows:

- *Left mouse button pressed in the main Camera view:* Drags the Portal object
- *Middle mouse button pressed in the HeroCam view:* Drags the Hero object
- *Right or middle mouse button pressed in any view:* Hides/shows the Portal object

The following controls are identical to the previous project:

- *Q key:* Initiates the camera shake effect
- *WASD keys:* Move the Dye character (the Hero object) and push the camera WC bounds
- *Arrow keys:* Move the Portal object
- *L/R/P/H keys:* Select the in-focus object with L/R keys refocusing the camera to the Left or Right minion
- *N/M and J/K keys:* Zoom into or away from the center of the camera, or the in-focus object

The goals of the project are as follows:

- To understand the Canvas Coordinate space to WC space transform
- To appreciate mouse clicks are specific to individual viewports
- To implement transformation between coordinate spaces and support mouse input

You can find the same external resource files as in the previous project in the assets folder.

Modify gEngine_Core to Pass Canvas ID to Input Component

To receive mouse input information, the gEngine_Input component needs to have access to the HTML canvas. This can be accomplished by editing the Engine_Core.js file and modifying the initializeEngineCore() function to pass the HTML canvas ID to the engine input component.

```
var initializeEngineCore = function (htmlCanvasID, myGame) {
    _initializeWebGL(htmlCanvasID);
    gEngine.VertexBuffer.initialize();
    gEngine.Input.initialize(htmlCanvasID);
    gEngine.AudioClips.initAudioContext();

    // Inits DefaultResources, when done, call startScene(myGame).
    gEngine.DefaultResources.initialize(function () { startScene(myGame); });
```

Implement Mouse Support in gEngine_Input

Follow these steps:

1. Edit Engine_Input.js and define a set of constants to represent the three mouse buttons.

```
var kMouseButton = {
    Left: 0,
    Middle: 1,
    Right: 2
};
```

2. Define the variables to support mouse input.

```
// Support mouse
var mCanvas = null;
var mButtonPreviousState = [];
var mIsButtonPressed = [];
var mIsButtonClicked = [];
var mMousePosX = -1;
var mMousePosY = -1;
```

Similar to keyboard input, mouse button states are arrays of three Boolean elements, each representing the state of the three mouse buttons.

3. Define the mouse movement event handler.

```
var _onMouseMove = function (event) {
    var inside = false;
    var bBox = mCanvas.getBoundingClientRect();

    // In Canvas Space now. Convert via ratio from canvas to client.
    var x = Math.round((event.clientX - bBox.left) *
        (mCanvas.width / bBox.width));
    var y = Math.round((event.clientY - bBox.top) *
        (mCanvas.height / bBox.height));

    if ((x >= 0) && (x < mCanvas.width) && (y >= 0) &&
        (y < mCanvas.height)) {
        mMousePosX = x;
        mMousePosY = mCanvas.height - 1 - y;
        inside = true;
    }
    return inside;
};
```

Notice that the mouse event handler transforms a raw pixel position into the Canvas Coordinate space by first checking whether the position is within the bounds of the canvas and then flipping the y position such that the displacement is measured with respect to the lower-left corner.

4. Define the mouse button press handler to record the button event.

```
var _onMouseDown = function (event) {
    if (_onMouseMove(event))
        mIsButtonPressed[event.button] = true;
};
```

5. Define the button release handler to facilitate the detection of a mouse button click event.

```
var _onMouseUp = function (event) {
    _onMouseMove(event);
    mIsButtonPressed[event.button] = false;
};
```

6. Modify the `initialize()` function to receive the `canvasID` parameter and initialize mouse event handlers.

```
var initialize = function (canvasID) {
    // Keyboard support
    // ... Identical to previous projects ...
```

```
// Mouse support
for (i = 0; i < 3; i++) {
    mButtonPreviousState[i] = false;
    mIsButtonPressed[i] = false;
    mIsButtonClicked[i] = false;
}
window.addEventListener('mousedown', _onMouseDown);
window.addEventListener('mouseup', _onMouseUp);
window.addEventListener('mousemove', _onMouseMove);
mCanvas = document.getElementById(canvasID);
};
```

7. Modify the update() function to process mouse button state changes in a similar fashion to the keyboard.

```
var update = function () {
    var i;
    for (i = 0; i < kKeys.LastKeyCode; i++) {
        mIsKeyClicked[i] = (!mKeyPreviousState[i]) && mIsKeyPressed[i];
        mKeyPreviousState[i] = mIsKeyPressed[i];
    }
    for (i = 0; i < 3; i++) {
        mIsButtonClicked[i] = (!mButtonPreviousState[i]) &&
            mIsButtonPressed[i];
        mButtonPreviousState[i] = mIsButtonPressed[i];
    }
};
```

8. Define the functions to retrieve mouse position and mouse button states.

```
var isButtonPressed = function (button) {
    return mIsButtonPressed[button];
};

var isButtonClicked = function (button) {
    return mIsButtonClicked[button];
};
var getMousePosX = function () { return mMousPosX; };
var getMousePosY = function () { return mMousPosY; };
```

9. When all is done, please remember to update the public interface.

Modify the Camera to Support Transformation to WC Space

The Camera object encapsulates the WC window and viewport and thus should be responsible for transforming mouse positions.

1. Create a new file in the `src/Engine/Camera/` folder and name it `Camera_Input.js`. This file will contain the mouse input support functionality.

2. Define functions to transform mouse positions from Canvas Coordinate space to the DC space, as illustrated in Figure 7-10.

```
Camera.prototype._mouseDCX = function () {
    return gEngine.Input.getMousePosX() - this.mViewport[Camera.eViewport.eOrgX];
};

Camera.prototype._mouseDCY = function () {
    return gEngine.Input.getMousePosY() - this.mViewport[Camera.eViewport.eOrgY];
};
```

3. Define a function to determine whether a given mouse position is within the viewport bounds of the camera.

```
Camera.prototype.isMouseInViewport = function () {
    var dcX = this._mouseDCX();
    var dcY = this._mouseDCY();
    return ((dcX >= 0) && (dcX < this.mViewport[Camera.eViewport.eWidth]) &&
        (dcY >= 0) && (dcY < this.mViewport[Camera.eViewport.eHeight]));
};
```

4. Define the functions to transform the mouse position into the WC space, as illustrated in Figure 7-11.

```
Camera.prototype.mouseWCX = function () {
    var minWCX = this.getWCCenter()[0] - this.getWCWidth() / 2;
    return minWCX + (this._mouseDCX() * (this.getWCWidth() /
        this.mViewport[Camera.eViewport.eWidth]));
};

Camera.prototype.mouseWCY = function () {
    var minWCY = this.getWCCenter()[1] - this.getWCHeight() / 2;
    return minWCY + (this._mouseDCY() * (this.getWCHeight() /
        this.mViewport[Camera.eViewport.eHeight]));
};
```

Testing the Mouse Input in MyGame

The main functionality to be tested includes the ability to detect which view should receive the mouse input, proper mouse button state identification, and correct transformed WC position. As in previous few examples, the `MyGame.js` implementation is largely similar to previous projects. In this case, only the `update()` function contains noteworthy changes that work with the new mouse input functionality.

```
MyGame.prototype.update = function () {
    // ... Identical to previous projects ...
```

```

msg = "";
// testing the mouse input
if (gEngine.Input.isButtonPressed(gEngine.Input.mousePosition.Left)) {
    msg += "[L Down]";
    if (this.mCamera.isMouseInViewport()) {
        this.mPortal.getXform().setXPos(this.mCamera.mousePositionX());
        this.mPortal.getXform().setYPos(this.mCamera.mousePositionY());
    }
}

if (gEngine.Input.isButtonPressed(gEngine.Input.mousePosition.Middle)) {
    if (this.mHeroCam.isMouseInViewport()) {
        this.mHero.getXform().setXPos(this.mHeroCam.mousePositionX());
        this.mHero.getXform().setYPos(this.mHeroCam.mousePositionY());
    }
}
if (gEngine.Input.isButtonClicked(gEngine.Input.mousePosition.Right))
    this.mPortal.setVisibility(false);

if (gEngine.Input.isButtonClicked(gEngine.Input.mousePosition.Middle))
    this.mPortal.setVisibility(true);

msg += " X=" + gEngine.Input.getMousePosX() + " Y="
      + gEngine.Input.getMousePosY();
this.mMsg.setText(msg);
};

```

The `camera.isMouseInViewport()` condition is checked when the viewport context is important, as in the case of a left mouse button press in the main camera view or a middle mouse button press in the `mHeroCam` view. This is in contrast to a right or middle mouse button click for setting the visibility of the `Portal` object. The controls will be executed no matter where the mouse position is.

You can now run the project and verify the correctness of the transformation to WC space. Press 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 centers to the mouse position. Left or middle mouse button drags in the wrong views have no effect on the corresponding objects; for example, a left mouse button drag in the `mHeroCam` or `mBrainCam` view has no effect on the `Portal` object. The right or middle mouse button click properly controls the visibility of the `Portal` object, independent of the location of the mouse pointer. Be aware that the browser maps the right mouse button click to a default pop-up menu. For this reason, you should avoid working with right mouse button clicks in your games.

Summary

This chapter was about controlling and interacting with the `Camera` object. You have learned about the most common camera manipulation operations including clamping, panning, and zooming. These operations are implemented in the game engine with utility functions that map the high-level specifications to actual WC window bound parameters. The sudden, often annoying, and potentially confusing results from camera manipulations are mitigated with the introduction of interpolation. Through the implementation of the camera shake effect, you have discovered that some movements can be modeled by simple mathematical formulations. You have experienced the importance of effective `Camera` object abstraction in supporting multiple camera views. The last section guided you through the implementation of transforming a mouse position from the Canvas Coordinate space to the WC space.

In Chapter 5, you found out how to represent and draw an object with a visually appealing image and control the animation of this object. In Chapter 6, you read about how to define an abstraction to encapsulate the behaviors of an object and the fundamental support to detect collisions between objects. This chapter is about the “directing” of these objects: what should be visible, where the focus should be, how much of the world to show, how to ensure smooth transition between foci, and how to receive input from the mouse. With these capabilities, you now have a well-rounded game engine framework, from representing and drawing objects to modeling and managing the behaviors of these objects to controlling what, where, and how to show the game world.

The following chapters will continue to examine object appearance and behavior at more advanced levels, including creating lighting and illumination effects in a 2D world and simulating and integrating behaviors based on simple classical mechanics.

Game Design Considerations

Now that you’ve learned the basics of object interaction, it’s a good time to start thinking about creating your first simple game mechanic to begin gaining insight into 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 is common in an industry such as video games where the creators typically spend quite a bit of time as content consumers before transitioning into content makers. Game studios 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 approach 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 new systems and mechanics from the ground up.

This begs the question, “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 criteria, including card and board, physical, video, and other game types. Taking things a step further, a *good* game is an interactive experience with rules people *enjoy* learning and applying to achieve an outcome they’re *invested* in. There’s quite a bit to unpack in that brief definition, of course, but as a general rule, players will enjoy a game more when the rules are discoverable, are 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 as well as entire game experiences. To use a metaphor, it can be helpful to think of game designs as 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.

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, it’s often helpful to focus only on creating and refining the basic game mechanics and interaction model; at this stage, try to avoid thinking about setting, meta-game, systems design, and the like (these will be folded into the design as it progresses).

The first design technique is a simple exercise that allows you to start learning the game design alphabet: an “escape the room” scenario with one simple mechanic, where you must accomplish a task in order to unlock a door and claim a reward. One goal of this exercise is to begin developing insight into how to create well-formed and logical rules that are discoverable and consistent, which is much easier to accomplish when the tasks are separated into basic interactions. You’ve already explored the beginnings of potential rule-based scenarios in earlier projects. Recall the Keyboard Support project from Chapter 4, which suggested you might have players move a smaller square completely into the boundary of a larger square in order to trigger some kind of behavior. How might that single interaction (or “letter of the game alphabet”) combine to form a game mechanic (or “word”) that makes sense? Figure 7-13 sets the stage for the locked room sandbox.

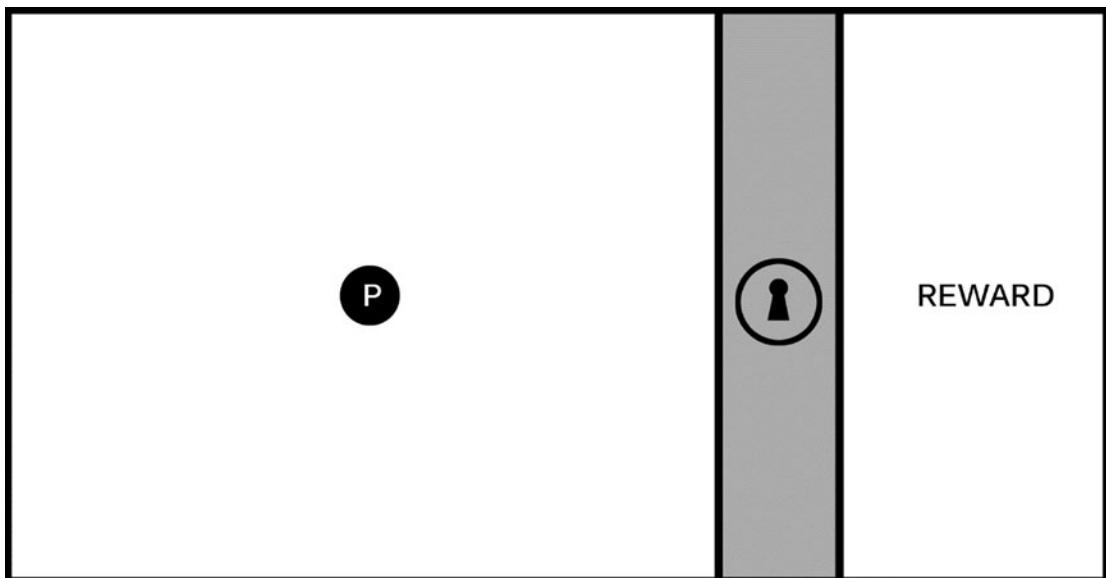


Figure 7-13. The image represents a single game screen divided into three areas: a playable area on the left with a hero character (the circle marked with a P), an impassable barrier marked with a lock icon, and a reward area on the right.

The screen represented in Figure 7-13 is a useful starting place when exploring new mechanics. The goal for this exercise is to create one logical challenge that a player must 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 can involve jumping or shooting, puzzle solving, narrative situations, and 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.

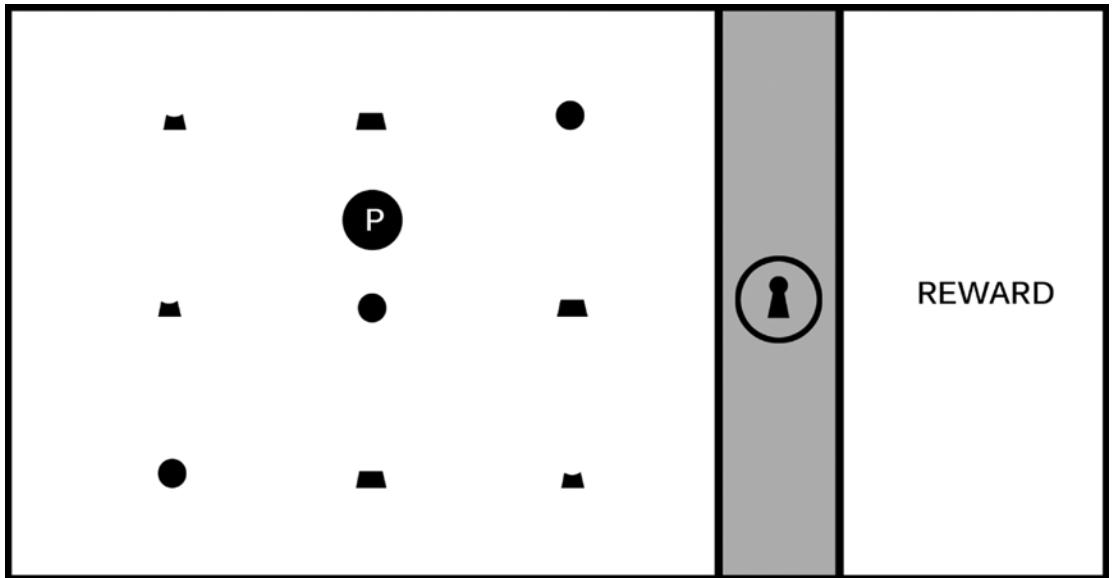


Figure 7-14. The game screen is populated with an assortment of individual objects

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.

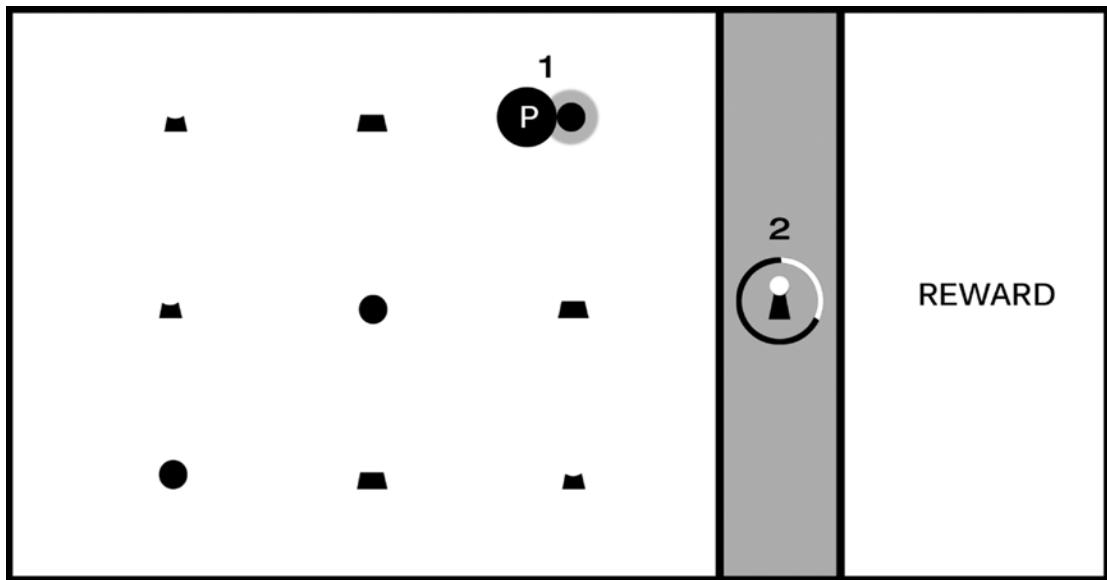


Figure 7-15. As the player moves the hero character around the game screen, the shapes “activate” with a highlight (#1); activating certain shapes causes a section of the lock and one-third of the surrounding ring to glow (#2)

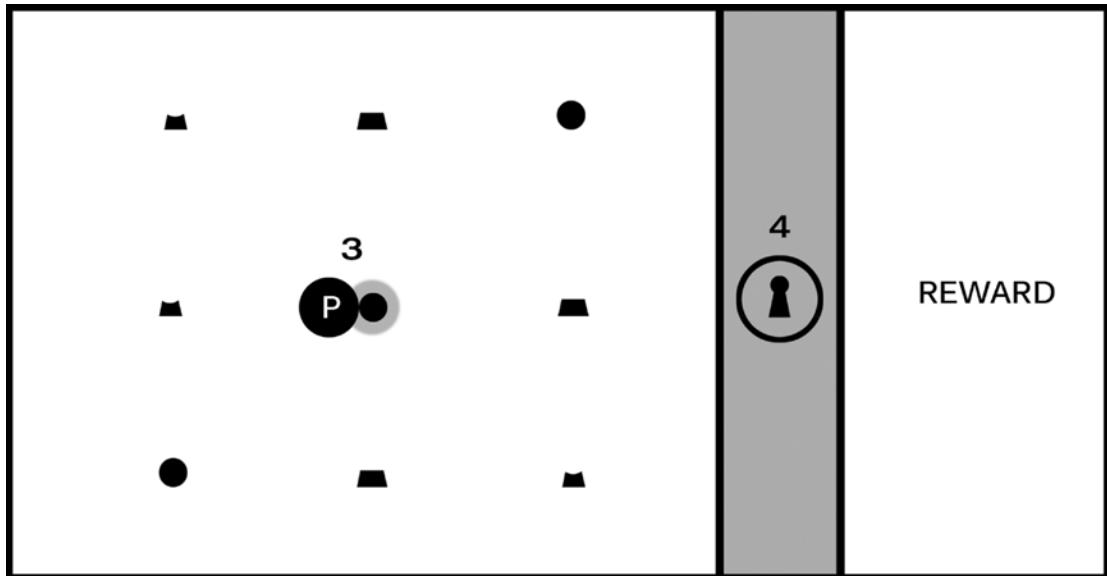


Figure 7-16. Activating some shapes (#3) will not cause the lock and ring to glow (#4)

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.

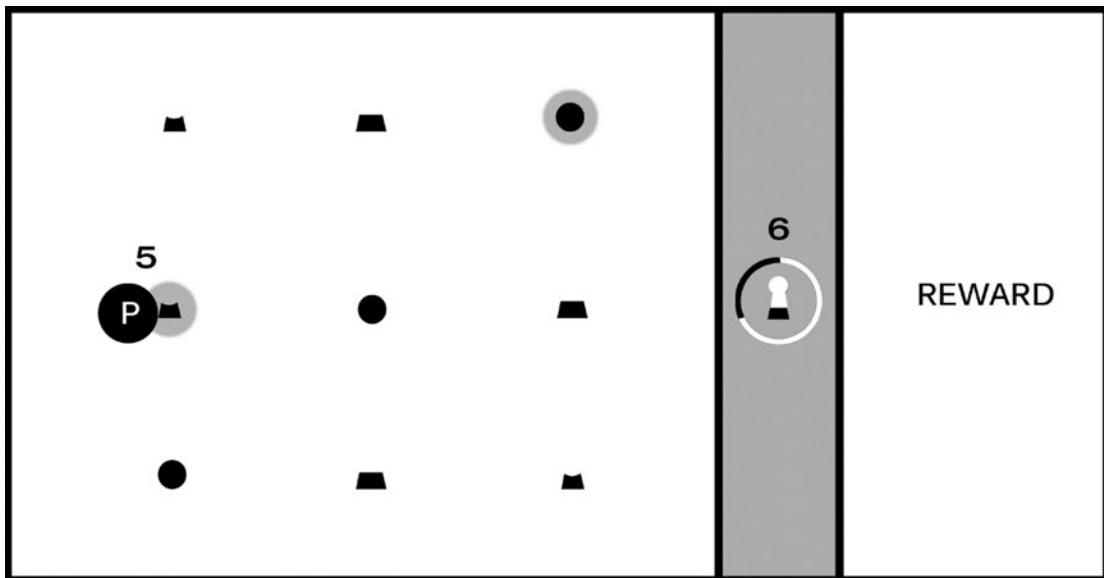


Figure 7-17. After the first object was activated (the circle in the upper-right) and caused the top section of the lock and first third of the ring to glow, as shown in Figure 7-15, the second object in the correct sequence (#5) caused the middle section of the lock and second third of the ring to glow (#6)

You (and players) should now have all required clues to learn the rules of this mechanic and solve the puzzle. There are three instances, each with a unique shape, that the player can interact with, and there is 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).

You’ve now created a well-formed and logically consistent, if simple, puzzle, with all of the elements needed to build a larger and more ambitious level. This unlocking sequence is an abstract mechanic. The game screen is intentionally devoid of game setting, visual style, or genre alignment at this stage of design because I don’t want to burden the exploration with any preconceived expectations. It will benefit you as a designer to spend time exploring the mechanic in its purest form before burdening it with a specific implementation, and you’ll likely be surprised at the directions these simple mechanics will take you as you build them out.

Simple mechanics like the previous, which can be described as “complete a multistage task in the correct sequence to achieve a goal,” are featured in many kinds of games; any game that requires players to collect parts of an object and combine them in an inventory to complete a challenge, for example, utilizes this mechanic. Individual mechanics can also be combined with other mechanics and game features to form compound elements that add complexity and flavor to the game experience.

The camera exercises in this chapter provide good examples for how you might add interest to a single mechanic. The Simple Camera Manipulations project, for example, demonstrates one method for advancing game action. Imagine in the previous example that after a player receives the reward for unlocking the barrier, the player moves the hero object to the right side of the screen and advances to a new “room” or area. Now imagine how gameplay would change if the camera advanced the screen at a fixed rate when the level started; the addition of autoscrolling changes this mechanic considerably because the player must solve the puzzle and unlock the barrier before the advancing barrier pushes the player off the screen. The first instance creates a leisurely puzzle-solving game experience, while the latter increases the tension considerably by giving the player a limited amount of time to complete each screen. In an autoscrolling implementation, how might you lay out the game screen to ensure the player had sufficient time to learn the rules and solve the puzzle?

The Multiple Cameras project can be especially useful as a mini-map that provides information about places in the game world not currently displayed on the game screen; in the case of the previous exercise, imagine that the locked barrier appeared somewhere else in the game world other than the player’s current screen and that a secondary camera acting as a mini-map displayed a zoomed-out view of the entire game world map. As the game designer, you might want to let the player know when they complete 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 the “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 words from the game design alphabet.

A game designer’s primary challenge is 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 not typically sound design to frustrate players by creating scenarios that are logically inconsistent and make players feel that they succeeded in the challenge only by trial-and-error or 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.

CHAPTER 8



Implementing Illumination and Shadow

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

Introduction

Up to now in your game engine you have implemented mostly functional modules in order to provide the core fundamentals required for many types of 2D games (that is, modules that serve to provide functionality directly to the end gameplay of a game created with your engine). 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.

An illumination model, or a lighting model, is a mathematic formulation that describes the color and brightness of a scene based on simulating 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 within a game engine can be more than a simple aesthetic effect. When used 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 within the real 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 intensity that seemingly absorbs the light 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 within your scenes. You will need to define depth values for the light sources to cast light energy upon the game objects, or renderables, 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 variation of the Phong illumination model will be derived and implemented in your game engine. While there are many versions of the Phong illumination model, you will be implementing a simplified version that caters to the 2D aspect of your game engine. 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 discussion in 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 intensive. 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.

- *Ambient Light:* Reviews the effects of lights in the absence of explicit light sources
- *Light Source:* Examines the effect of illumination from a single light source
- *Multiple Light Sources:* Develops game engine infrastructure to support multiple light sources
- *Diffuse Reflection and Normal Maps:* Simulates diffuse light reflection in 2D
- *Specular Light and Material:* Models light reflecting off surfaces and reaching the camera
- *Light Source types:* Introduces illumination based on different types of light sources
- *Shadow:* Approximates the results from light occlusion

Together, the projects in this chapter build a powerful tool for adding visual intricacy into your games.

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 complex and computationally too expensive to simulate in real time. Instead, in computer graphics and most 2D games, ambient lighting is approximated by adding a constant ambient light color to every object within the current scene or world. It is important to note that while ambient lighting can provide desired results, it is not meant to mimic real-world lighting exactly. For your specific engine implementation, each object within the scene needs access to an ambient color and an ambient intensity before it is drawn in order to take into account the ambient lighting of the scene.

The Global Ambient Project

This project demonstrates how to implement ambient lighting within your scenes by providing a global ambient color and a global ambient intensity that each renderable object references before being drawn. You can see an example of this project running in Figure 8-1. The source code of this project is located in the [Chapter8/8.1.GlobalAmbient](#) folder.



Figure 8-1. Running the Global Ambient project

The controls of the project are as follows:

- *Left mouse button:* Increases the global red ambient
- *Middle mouse button:* Decreases the global red ambient
- *Left/right arrow keys:* Decrease/increase the global ambient intensity

The goals of the project are as follows:

- To experience the effects of ambient lighting
- To understand how to implement a simple global ambient across a scene
- To refamiliarize yourself with the Shader/Renderable pair structure to interface to GLSL shaders and the game engine

You can find the following external resources in the assets folder: the fonts folder that 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 for the game engine is the GLSL shader. This is because it allows you to implement the shading technique, which in turn provides the outline for how your engine must be modified in order to support this new shader. Thus, to start, implement the global ambient into your `SimpleFS.glsl`.

1. Modify the fragment shader `SimpleFS.glsl` by adding the `uniform` variables `uGlobalAmbientColor` and `uGlobalAmbientIntensity`. Then utilize them by multiplying them by `uPixelColor` to get the final color for each fragment. You can see this implemented in the following code:

```
precision mediump float;

// Color of the object
uniform vec4 uPixelColor;
uniform vec4 uGlobalAmbientColor; // this is shared globally
uniform float uGlobalAmbientIntensity; // this is shared globally

void main(void) {
    // for every pixel called sets to the user specified color
    gl_FragColor = uPixelColor * uGlobalAmbientIntensity * uGlobalAmbientColor;
}
```

2. Similarly modify the texture fragment shader `TextureFS.glsl` by adding the `uniform` variables `uGlobalAmbientColor` and `uGlobalAmbientIntensity`. Then utilize them by multiplying them by `c` (the fragment color sampled from the texture) to get the final color for each fragment. Remember that the color for `c` was obtained by using the interpolated `vTexCoord` variable from the vertex shader to sample a fragment from the passed-in texture. You can see this implemented in the following code:

```
precision mediump float;

uniform sampler2D uSampler;

// Color of the object
uniform vec4 uPixelColor;
uniform vec4 uGlobalAmbientColor; // this is shared globally
uniform float uGlobalAmbientIntensity;

varying vec2 vTexCoord;

void main(void) {
    // texel color look up based on interpolated UV value in vTexCoord
    vec4 c = texture2D(uSampler, vec2(vTexCoord.s, vTexCoord.t));

    c = c * uGlobalAmbientIntensity * uGlobalAmbientColor;
```

```

    // tint the textured area, and
    // leave transparent area as defined by the texture
    vec3 r = vec3(c) * (1.0-uPixelColor.a) + vec3(uPixelColor) * uPixelColor.a;
    vec4 result = vec4(r, c.a);

    gl_FragColor = result;
}

```

Modifying SimpleShader

With global ambient color and intensity now implemented within the shader, you need to modify the simple shader in order to accommodate the new variables by passing the values onto the GLSL shader.

1. Modify the `SimpleShader.js` file in the `src/Engine/Shaders` folder to hold two new variables in the constructor for storing the references or locations of the ambient color and intensity variables within the GLSL shader.

```

this.mGlobalAmbientColor = null;
this.mGlobalAmbientIntensity = null;

```

2. In step E of the `SimpleShader` constructor, get the locations of the ambient color and intensity within the shader by using WebGL's `getUniformLocation()` function, as shown in the following code:

```

// Step E: references: uniforms:
//           uPixelColor, uModelTransform, and uViewProjTransform
this.mPixelColor = gl.getUniformLocation(this.mCompiledShader, "uPixelColor");
this.mModelTransform = gl.getUniformLocation(
    this.mCompiledShader, "uModelTransform");
this.mViewProjTransform = gl.getUniformLocation(
    this.mCompiledShader, "uViewProjTransform");
this.mGlobalAmbientColor = gl.getUniformLocation(
    this.mCompiledShader, "uGlobalAmbientColor");
this.mGlobalAmbientIntensity = gl.getUniformLocation(
    this.mCompiledShader, "uGlobalAmbientIntensity");

```

3. In the `activateShader()` function, pass the ambient color and intensity values to the shader by utilizing the global values and the locations you obtained in the previous step along with the GLSL uniform set functions provided by WebGL. Notice that the data type used by the set functions for GLSL variables state which data type is used explicitly. As you can probably guess, `uniform4fv` corresponds to `vec4`, which is used to hold the color, and to `uniform1f`, which corresponds to a float and is used to hold the intensity.

```

// Activate the shader for rendering
SimpleShader.prototype.activateShader = function(pixelColor, aCamera) {
    var gl = gEngine.Core.getGL();
    gl.useProgram(this.mCompiledShader);
    gl.uniformMatrix4fv(this.mViewProjTransform, false, aCamera.getVPMatrix());
    gl.bindBuffer(gl.ARRAY_BUFFER, gEngine.VertexBuffer.getGLVertexRef());
    gl.vertexAttribPointer(this.mShaderVertexPositionAttribute,

```

```

    3,           // each element is a 3-float (x,y,z)
    gl.FLOAT,   // data type is FLOAT
    false,      // if the content is normalized vectors
    0,          // number of bytes to skip in between elements
    0);         // offsets to the first element
gl.enableVertexAttribArray(this.mShaderVertexPositionAttribute);
gl.uniform4fv(this.mPixelColor, pixelColor);
gl.uniform4fv(this.mGlobalAmbientColor,
    gEngine.DefaultResources.getGlobalAmbientColor());
gl.uniform1f(this.mGlobalAmbientIntensity,
    gEngine.DefaultResources.getGlobalAmbientIntensity());
};

}

```

Modifying the Engine

Now that the shader and the corresponding shader object for ambient color and intensity are properly integrated, you can modify the engine in order to support the variables for global ambient.

1. Begin by adding a global ambient color and a global ambient intensity variable in the `Engine_DefaultResources.js` file, located in the `src/Engine/Core/Resources` folder, as shown here:

```
// Global Ambient color
var mGlobalAmbientColor = [0.3, 0.3, 0.3, 1];
var mGlobalAmbientIntensity = 1;
```

2. Define basic get and set accessors to allow for the modification of the ambient color and intensity, as shown in the following code:

```
var getGlobalAmbientIntensity = function() { return mGlobalAmbientIntensity; };
var setGlobalAmbientIntensity = function(v) { mGlobalAmbientIntensity = v; };
var getGlobalAmbientColor = function() { return mGlobalAmbientColor; };
var setGlobalAmbientColor = function(v) {
    mGlobalAmbientColor = vec4.fromValues(v[0], v[1], v[2], v[3]); };
```

3. Remember to include the accessors in your `mPublic()` function list.

```
getGlobalAmbientColor: getGlobalAmbientColor,
setGlobalAmbientColor: setGlobalAmbientColor,
getGlobalAmbientIntensity: getGlobalAmbientIntensity,
setGlobalAmbientIntensity: setGlobalAmbientIntensity,
```

Testing the Ambient Illumination

Now that the engine supports ambient lighting for a scene, all that is left is to verify the correctness by utilizing it within `MyGame.js` and observe the shaded results. To get a better picture of what exactly is happening and how exactly the ambient lighting can be utilized, you can begin with a clean `MyGame.js` file and re-implement the core functions.

1. Start by adding variables to the constructor for a camera, a background, a hero, and some minions. Additionally, remember to inherit from the Scene class in order to utilize its functionality.

```
"use strict";
function MyGame() {
    this.kMinionSprite = "assets/minion_sprite.png";
    this.kBg = "assets/bg.png";

    // The camera to view the rectangles
    this.mCamera = null;
    this.mBg = null;

    this.mMsg = null;

    // the hero and the support objects
    this.mHero = null;
    this.mLMinion = null;
    this.mRMinion = null;
}
gEngine.Core.inheritPrototype(MyGame, Scene);
```

2. Next, practice proper implementation by remembering to load and unload the background and the minions. You can see this in the following code:

```
MyGame.prototype.loadScene = function() {
    gEngine.Textures.LoadTexture(this.kMinionSprite);
    gEngine.Textures.LoadTexture(this.kBg);
};

MyGame.prototype.unloadScene = function() {
    gEngine.Textures.UnloadTexture(this.kMinionSprite);
    gEngine.Textures.UnloadTexture(this.kBg);
};
```

3. Now initialize the camera and scene objects by setting them to the values shown here so that the scene is visible by the camera upon startup:

```
MyGame.prototype.initialize = function() {
    // Step A: set up the cameras
    this.mCamera = new 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

    var bgR = new SpriteRenderable(this.kBg);
    bgR.setTexPixelPositions(0, 1900, 0, 1000);
    bgR.getXform().setSize(190, 100);
```

```

bgR.getXform().setPosition(50, 35);
this.mBg = new GameObject(bgR);

// Step B: Create the hero object with texture from lower-left corner
this.mHero = new Hero(this.kMinionSprite);

this.mLMinion = new Minion(this.kMinionSprite, 30, 30);
this.mRMinion = new Minion(this.kMinionSprite, 70, 30);

this.mMsg = new FontRenderable("Status Message");
this.mMsg.setColor([1, 1, 1, 1]);
this.mMsg.getXform().setPosition(1, 2);
this.mMsg.setTextHeight(3);
};

}

```

4. Next, draw each object in the scene, as shown in the following functions:

```

MyGame.prototype.drawCamera = function(camera) {
    camera.setupViewProjection();
    this.mBg.draw(camera);
    this.mHero.draw(camera);
    this.mLMinion.draw(camera);
    this.mRMinion.draw(camera);
};

// This is the draw function, make sure to setup proper drawing environment,
// and more importantly, make sure to _NOT_ change any state.
MyGame.prototype.draw = function() {
    // Step A: clear the canvas
    gEngine.Core.clearCanvas([0.9, 0.9, 0.9, 1.0]); // clear to light gray

    // Step B: draw with all three cameras
    this.drawCamera(this.mCamera);
    this.mMsg.draw(this.mCamera); // only draw status in the main camera
};

```

5. Lastly, implement the following update function to update each object as well as the camera within the scene. Additionally, provide control over the global ambient color and intensity for testing purposes and display a status for color and intensity.

```

// The update function, updates the application state. Make sure to _NOT_ draw
// anything from this function!
MyGame.prototype.update = function() {
    var deltaAmbient = 0.01;
    var msg = "Current Ambient]: ";

    this.mCamera.update(); // to ensure proper interpolated movement effects

    this.mLMinion.update(); // ensure sprite animation
    this.mRMinion.update();
}

```

```

this.mHero.update(); // allow keyboard control to move

this.mCamera.panWith(this.mHero.getXform(), 0.8);

var v = gEngine.DefaultResources.getGlobalAmbientColor();
if (gEngine.Input.isButtonPressed(gEngine.Input.mouseButton.Left))
    v[0] += deltaAmbient;
if (gEngine.Input.isButtonPressed(gEngine.Input.mouseButton.Middle)
    v[0] -= deltaAmbient;
if (gEngine.Input.isKeyPressed(gEngine.Input.keys.Left))
    gEngine.DefaultResources.setGlobalAmbientIntensity(
        gEngine.DefaultResources.getGlobalAmbientIntensity() - deltaAmbient);
if (gEngine.Input.isKeyPressed(gEngine.Input.keys.Right))
    gEngine.DefaultResources.setGlobalAmbientIntensity(
        gEngine.DefaultResources.getGlobalAmbientIntensity() + deltaAmbient);

msg += " Red=" + v[0].toPrecision(3) + " Intensity=" +
    Engine.DefaultResources.getGlobalAmbientIntensity().toPrecision(3);
this.mMsg.setText(msg);
};


```

Observations

You can now see the results of the project by running it. Notice that the scene itself is dark. This is because the RGB values for the global ambient color were all initialized to 0.3, and 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 would happen if the RGB values were set to 1 and the intensity was set 0.3 because applying the ambient values is done through straightforward multiplication. Before moving onto the next project, try fiddling with the ambient red channel and the ambient intensity in order to see its effect on the scene. By pressing the right arrow key, you can increase the intensity of the entire scene and make all objects more visible. The next section describes how to create and direct a light source to illuminate only on selected objects.

Light Source

With ambient lighting for the scene completed, it is now time to implement light with an object-oriented approach while adhering to your expectations of what a light is and how it interacts with the environment. This can be achieved through the definition of a `Light` object to represent a light source. As mentioned, to implement a light source, the 2D engine will need to venture into the third dimension to properly simulate light energy traveling from the source to the surface of your geometries. There are several types of light sources; you will begin with the implementation of a simple point light. A point light is a light that emits light uniformly in all directions. In the real world, a point light can be thought of as a simple lightbulb.

The most basic implementation of a point light can be distilled down to illuminating an area or radius around a specified point. In three-dimensional space, the region of illumination by a point light source can simply be thought of as a sphere, referred to as *volume* of illumination. The volume of illumination of a point light is defined with the light's position being at the center of the sphere and illuminating the volume that is within the radius of the sphere. To observe the effects of a light source, objects must be present and within the volume of illumination. Now, consider your 2D engine; thus far you have implemented a system in which everything is rendered in 2D. Or, rather, everything is rendered at a single plane where $z = 0$ and objects are layered (by draw order) in order to display one object in front of another. On this system, you

are now going to add light sources that reside in 3D. To observe the effects of a light source, its illumination volume must overlap an object on the XY plane where your objects exist. Figure 8-2 shows the volume of illumination from a simple point light intersecting an object on the XY plane where $z = 0$. This intersection results in an illuminated circle on the object.

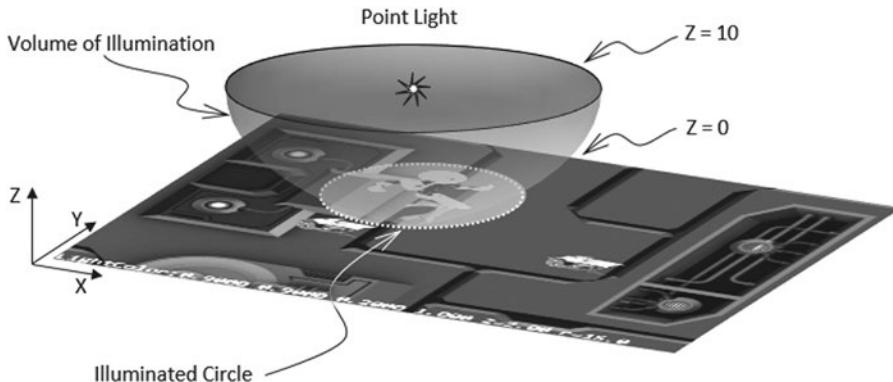


Figure 8-2. Point light and the corresponding volume of illumination in 3D

GLSL Implementation and Integration into the Game Engine

For quality results, the computations associated with illumination models must be performed once for each affected pixel. Recall that in WebGL and GLSL shaders the color of each pixel is computed by the corresponding GLSL fragment shader. In this chapter, as each fundamental element of the Phong illumination model is studied, the accompanied GLSL fragment shader implementation will also be explained.

From your experience building the game engine, you will remember that the engine interfaces to the GLSL shaders with the corresponding subclasses of the `Shader`/`Renderable` object pairs: `Shader` objects to interface to the GLSL shaders and `Renderable` objects to provide programmers with the convenience of manipulating many copies of geometries of the same shader type. For example, `TextureVS` and `TextureFS` 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 `TextureVS/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, `LightFS`, via the `LightShader`/`LightRenderable` pair for computing the appropriate pixel colors. The GLSL vertex shader, `TextureVS`, is reused because light source illumination involves the same information to be processed at each vertex.

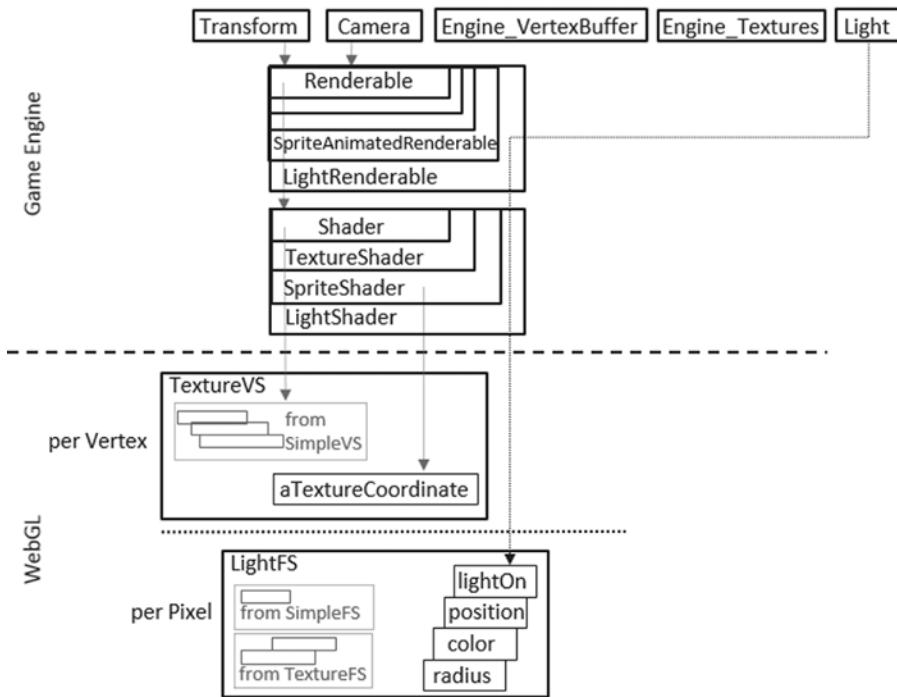


Figure 8-3. LightShader/LightRenderable pair and the corresponding GLSL LightShader

Finally, before you begin learning about the elements of a Phong Illumination model, it is important to point out again that the GLSL fragment shader is invoked once for every pixel covered by the corresponding geometry. This means the GLSL fragment shaders you are about to learn 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 of most importance!

The Simple Light Shader Project

This project demonstrates how to implement a simple point light and illuminate objects within the scene. 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.SimpleLightShader folder.

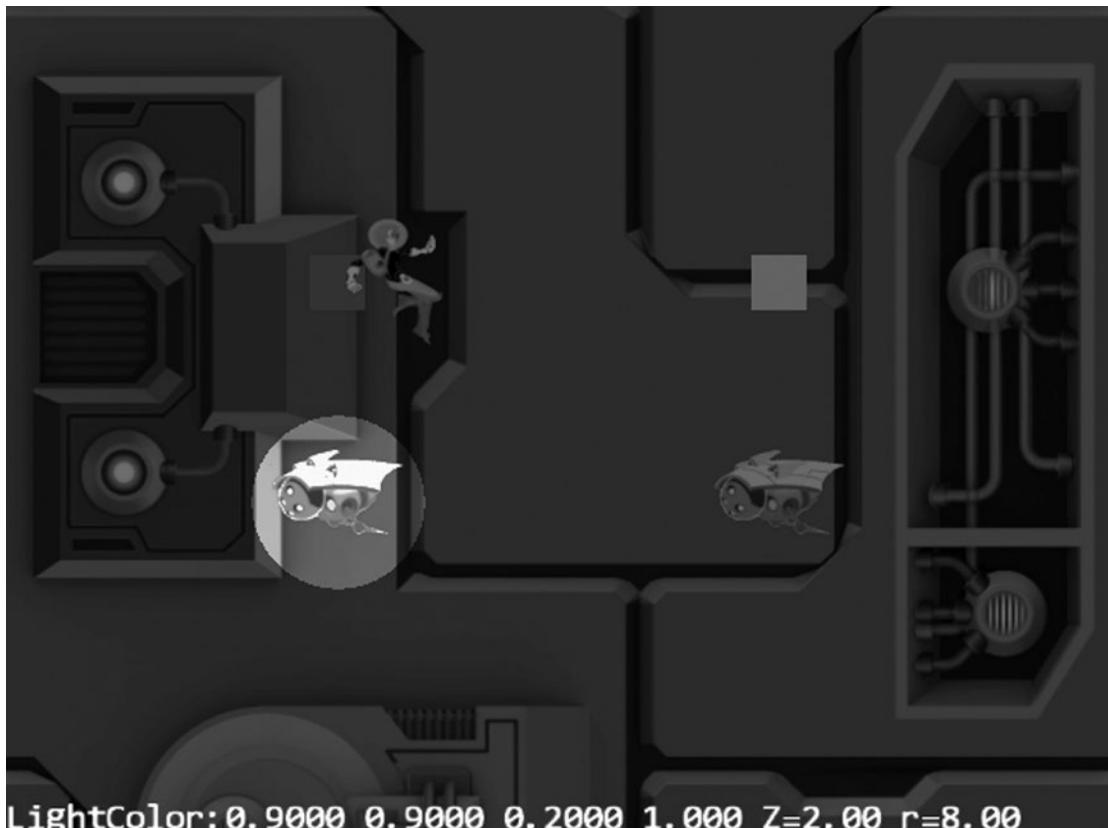


Figure 8-4. Running the Simple Light Shader project

The controls of the project are as follows:

- WASD keys: Move the hero character on the screen
- WASD keys + left mouse button: Move the hero character and the light source around the screen
- Left/right arrow key: Decreases/increases the light intensity
- Z/X key: Increases/decreases the light Z position
- C/V key: Increases/decreases the light radius

The goals of the project are as follows:

- To understand how to simulate the illumination effects from a point light
- To experience illumination results from a point light
- To implement a GLSL shader that supports point light illumination

You can find the following external resources in the assets folder: the fonts folder that contains the default system fonts and two texture images (`minion_sprite.png` and `bg.png`). The objects are sprite elements of `minion_sprite.png`, and the background is represented by `bg.png`.

Creating the GLSL Light Fragment Shader

As with the previous section, the implementation will begin with the GLSL shader. The shader uses light properties to calculate the illuminated circle. The GLSL vertex shader will remain identical to the `TextureVS` since the same information and computation will be performed at each vertex.

- Under the `src/GLSLShaders` folder, create a new file and name it `LightFS.glsl`.
- Add the standard uniform and varying variables for the texture sampler, texture coordinate, ambient properties, and pixel color as in previous projects. Furthermore, you can now add support for a single light by adding variables for each of the light's properties. It is also important to notice that the light's position and radius are in pixel space, or Device Coordinate (DC) space, to facilitate illumination computations.

```
precision mediump float;

// The object that fetches data from texture.
uniform sampler2D uSampler;

// Color of pixel
uniform vec4 uPixelColor;
uniform vec4 uGlobalAmbientColor; // this is shared globally
uniform float uGlobalAmbientIntensity;

// Light information
uniform bool uLightOn;
uniform vec4 uLightColor;
uniform vec4 uLightPosition;    // in pixel space!
uniform float uLightRadius;    // in pixel space!

varying vec2 vTexCoord;
```

- To complete the shader, implement the `main()` function to do the following:
 - Sample the texture color and apply the ambient color and intensity.
 - Determine whether the current fragment should be illuminated by the light source. To do this, first check whether the light is on; if it is, compute the distance between the light's position (in pixel space) and the current fragment's position (in pixel space) that is defined in the GLSL-provided variable `gl_FragCoord.xyz`. If the distance is less than that of the light's radius (again in pixel space), then accumulate the light's color.
 - The last step is to apply the tint and to set the final color via `gl_FragColor`.

```
void main(void) {
    vec4 textureMapColor = texture2D(uSampler, vec2(vTexCoord.s, vTexCoord.t));
    vec4 lgtResults = uGlobalAmbientIntensity * uGlobalAmbientColor;

    // now decide if we should illuminate by the light
    if (uLightOn && (textureMapColor.a > 0.0)) {
        float dist = length(uLightPosition.xyz - gl_FragCoord.xyz);
```

```

        if (dist <= uLightRadius)
            lgtResults += uLightColor;
    }
lgtResults *= textureMapColor;

// tint the textured area, and leave transparent area as defined by the texture
vec3 r = vec3(lgtResults) * (1.0-uPixelColor.a) + vec3(uPixelColor) *
    uPixelColor.a;
vec4 result = vec4(r, lgtResults.a);

gl_FragColor = result;
}

```

Creating a Light Object

With the GLSL LightFS shader defined, you can now create an object to encapsulate a point light source.

1. Create a new folder called `Lights` under the `src/Engine` folder. In the `Lights` folder, add a new file and name it `Lights.js`. Remember to load this new source file in `index.html`.
2. In `Lights.js`, create a simple constructor that initializes a color, position, radius, and on-off variable for the light. Set their initial values as follows:

```

// Constructor
function Light() {
    this.mColor = vec4.fromValues(0.1, 0.1, 0.1, 1); // light color
    this.mPosition = vec3.fromValues(0, 0, 5); // light position in WC
    this.mRadius = 10; // effective radius in WC
    this.mIsOn = true;
}

```

3. Add the get and set accessors shown here to allow for the proper modification of the light's instance variables from outside the object.

```

// simple setters and getters
Light.prototype.setColor = function(c) { this.mColor = vec4.clone(c); };
Light.prototype.getColor = function() { return this.mColor; };

Light.prototype.set2DPosition = function(p) {
    this.mPosition = vec3.fromValues(p[0], p[1], this.mPosition[2]); };
Light.prototype.setXPos = function(x) { this.mPosition[0] = x; };
Light.prototype.setYPos = function(y) { this.mPosition[1] = y; };
Light.prototype.setZPos = function(z) { this.mPosition[2] = z; };
Light.prototype.getPosition = function() { return this.mPosition; };

Light.prototype.setRadius = function(r) { this.mRadius = r; };
Light.prototype.getRadius = function() { return this.mRadius; };

Light.prototype.setLightTo = function(isOn) { this.mIsOn = isOn; };
Light.prototype.isLightOn = function() { return this.mIsOn; };

```

Creating the LightShader Object

The LightShader object subclasses from the SpriteShader to encapsulate its communication, which handles the WebGL-specific details of passing information to the GLSL shader. This provides the engine with a convenient interface for the shader.

- Under the `src/Engine/Shaders` folder, create a new file and name it `LightShader.js`. Remember to load this new source file in `index.html`.
- Now add a constructor in order to initialize the references for the light's variables and to obtain their reference locations within the shader. Remember to inherit from the `SpriteShader` object.

```
"use strict";
function LightShader(vertexShaderPath, fragmentShaderPath) {
    // Call super class constructor
    SpriteShader.call(this, vertexShaderPath, fragmentShaderPath);

    // glsl uniform position references
    this.mColorRef = null;
    this.mPosRef = null;
    this.mRadiusRef = null;
    this.mIsOnRef = null;
    this.mLight = null; // <-- this is the light source in the Game Engine

    // create the references to these uniforms in the LightShader
    var shader = this.mCompiledShader;
    var gl = gEngine.Core.getGL();
    this.mColorRef = gl.getUniformLocation(shader, "uLightColor");
    this.mPosRef = gl.getUniformLocation(shader, "uLightPosition");
    this.mRadiusRef = gl.getUniformLocation(shader, "uLightRadius");
    this.mIsOnRef = gl.getUniformLocation(shader, "uLightOn");
}
gEngine.Core.inheritPrototype(LightShader, SpriteShader);
```

- Provide a basic set function to specify which light the shader should use.

```
LightShader.prototype.setLight = function(l) {
    this.mLight = l;
};
```

- Override the `activateShader()` function from the `SpriteShader` object to add the new functionality of turning the light on and off, as shown here. Notice that you still call the superclass's `activateShader()` function.

```
LightShader.prototype.activateShader = function (pixelColor, aCamera) {
    // first call the super class's activate
    SpriteShader.prototype.activateShader.call(this, pixelColor, aCamera);
```

```
// now push the light information to the shader
if (this.mLight !== null) {
    this._loadToShader(aCamera);
} else {
    gEngine.Core.getGL().uniform1i(this.mIsOnRef, false); // <-- switch off
    the light!
}
};
```

5. Implement a function to load the light's properties into the corresponding shader. Recall that this is achieved by using the references created in the constructor and WebGL's uniform set functions. Also notice that the camera provides the new coordinate space functionality of `wcPosToPixel` and `wcSizeToPixel`. The implementation of these functions will be examined shortly.

```
LightShader.prototype._loadToShader = function(aCamera) {
    var gl = gEngine.Core.getGL();
    gl.uniform1i(this.mIsOnRef, this.mLight.isLightOn());
    if (this.mLight.isLightOn()) {
        var p = aCamera.wcPosToPixel(this.mLight.getPosition());
        var r = aCamera.wcSizeToPixel(this.mLight.getRadius());
        var c = this.mLight.getColor();

        gl.uniform4fv(this.mColorRef, c);
        gl.uniform4fv(this.mPosRef, vec4.fromValues(p[0], p[1], p[2], 1));
        gl.uniform1f(this.mRadiusRef, r);
    }
};
```

Creating the LightRenderable Object

With the engine's `LightShader` object defined to interface to the GLSL `LightFS` shader, you can now focus on creating a new `Renderable` object that subclasses from `SpriteAnimateRenderable` to support the interaction with lights. You can think of this object as a `SpriteAnimateRenderable` that can be illuminated by a `Light` object.

1. Begin by creating a new file in the `src/Engine/Renderables` folder and naming it `LightRenderable.js`. Remember to load this new source file in `index.html`.
2. Next, add a constructor to call the superclass's constructor to set the corresponding light shader and to create a variable for the light that will illuminate this object.

```
function LightRenderable(myTexture) {
    SpriteAnimateRenderable.call(this, myTexture);
    Renderable.prototype._setShader.call(this,
        gEngine.DefaultResources.getLightShader());

    // here is the Light source
    this.mLight = null;
}
gEngine.Core.inheritPrototype(LightRenderable, SpriteAnimateRenderable);
```

3. Add a draw function that passes the illuminating light source to the LightShader object for communicating with the GLSL fragment shader.

```
LightRenderable.prototype.draw = function(aCamera) {
    this.mShader.setLight(this.mLight);
    SpriteAnimateRenderable.prototype.draw.call(this, aCamera);

};
```

4. Lastly, simply add the support to get and set the light, as shown in the following code:

```
LightRenderable.prototype.getLight = function() {
    return this.mLight;
};
LightRenderable.prototype.addLight = function(l) {
    this.mLight = l;
};
```

Defining a Default LightShader Instance

You can now modify the engine to support the initializing, loading, and unloading of the new LightShader object.

1. Begin by adding a variable for the light shader in the Engine_DefaultResources.js file located in the src/Engine/Core/Resources folder. Also, define an accessor function as shown here:

```
// Light Shader
var kLightFS = "src/GLSLShaders/LightFS.glsl";
var mLightShader = null;

var getLightShader = function() { return mLightShader; };
```

2. Now instantiate a new light shader in the _createShaders() function, as shown here:

```
var _createShaders = function(callBackFunction) {
    gEngine.ResourceMap.setLoadCompleteCallback(null);
    mConstColorShader = new SimpleShader(kSimpleVS, kSimpleFS);
    mTextureShader = new TextureShader(kTextureVS, kTextureFS);
    mSpriteShader = new SpriteShader(kTextureVS, kTextureFS);
    mLineShader = new LineShader(kSimpleVS, kSimpleFS);
    mLightShader = new LightShader(kTextureVS, kLightFS);
    callBackFunction();
};
```

3. In the initialize() function, add the following code to properly load the file:

```
gEngine.TextFileLoader.loadTextFile(kLightFS,
    gEngine.TextFileLoader.eTextFileType.eTextFile);
```

4. In the `cleanUp()` function, add the following line of code to unload the file when it is no longer needed:

```
gEngine.TextFileLoader.unloadTextFile(kLightFS);
```

5. Don't forget to add the `get` accessor function to `mPublic()` so that it can be accessed.

```
getLightShader: 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, origin of the Camera) 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.js` and define the constructor for the `PerRenderCache` object 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 a rendering begins.

```
function PerRenderCache() {
    this.mWCToPixelRatio = 1; // WC to pixel transformation
    this.mCameraOrgX = 1; // Lower-left corner of camera in WC
    this.mCameraOrgY = 1;
}
```

2. Modify the constructor of the `Camera` to instantiate a new `PerRenderCache` object. It is important to note that this variable should be used only for rendering purposes. It should not be used for functionality within the game or game engine.

```
function Camera(wcCenter, wcWidth, viewportArray, bound) {
    // WC and viewport position and size
    this.mCameraState = new CameraState(wcCenter, wcWidth);
    this.mCameraShake = null;

    // ...

    // per-rendering cached information
    // needed for computing transforms for shaders
    // updated each time in setupViewProjection()
```

```

this.mRenderCache = new PerRenderCache();
    // SHOULD NOT be used except
    // xform operations during the rendering
    // Client game should not access this!
}

```

3. Initiate the per-render cache in the `setupViewProjection()` function by adding step B4 to calculate and set the cache using the existing Camera viewport width, world width, and world height.

```

// Step B4: compute and cache per-rendering information
this.mRenderCache.mWCToPixelRatio =
    this.mViewport[Camera.eViewport.eWidth] / this.getWCWidth();
this.mRenderCache.mCameraOrgX = center[0] - (this.getWCWidth()/2);
this.mRenderCache.mCameraOrgY = center[1] - (this.getWCHeight()/2);

```

Adding Camera Transform Functions

Now that the per-render cache is defined and properly initialized, you can extend the functionality of the camera by implementing the functions to convert from WC and pixel space. For code readability and maintainability, this functionality has been delegated to a separate file because of its specific purpose. Another important note is that since you are converting from WC to pixel space and pixel space has no z-axis, you need to calculate a fake z-value for the pixel space coordinate.

1. Under the `src/Engine/Cameras` folder, create a new file and name it `Camera_Xform.js`. Remember to load this new source file in `index.html`.
2. Approximate a fake pixel space z value by scaling the input parameter according to the `mWCToPixelRatio` variable.

```

Camera.prototype.fakeZInPixelSpace = function(z) {
    return z * this.mRenderCache.mWCToPixelRatio;
};

```

3. Provide a function to convert from WC to pixel space for a `vec3` position. This is accomplished by subtracting the camera origin followed by scaling with the `mWCToPixelRatio`. The 0.5 offset at the end of the x and y conversion ensure 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
    // Convert the position to pixel space
    var x = this.mViewport[Camera.eViewport.eOrgX] +
        ((p[0] - this.mRenderCache.mCameraOrgX) *
            this.mRenderCache.mWCToPixelRatio) + 0.5;
    var y = this.mViewport[Camera.eViewport.eOrgY] +
        ((p[1] - this.mRenderCache.mCameraOrgY) *
            this.mRenderCache.mWCToPixelRatio) + 0.5;
    var z = this.FakeZInPixelSpace(p[2]);
    return vec3.fromValues(x, y, z);
};

```

4. Simply provide a function for converting a length from WC to pixel space by scaling with the `mWCToPixelRatio` variable.

```
Camera.prototype.wcSizeToPixel = function(s) {
    return (s * this.mRenderCache.mWCToPixelRatio) + 0.5;
};
```

Testing the Light

The MyGame level must be modified to utilize and test the new light functionality.

Modifying the Hero and Minion

Make a few quick modifications to the Hero and Minion objects to accommodate the new `LightRenderable` object.

1. In the `Hero.js` file within the `src/MyGame/Objects` folder, replace the `SpriteRenderable` instantiation with a `LightRenderable` instantiation.

```
this.mDye = new LightRenderable(spriteTexture);
```

2. In the `Minion.js` file within the `src/MyGame/Objects` folder, replace the `SpriteRenderable` instantiation with a `LightRenderable` instantiation.

```
this.mMinion= new LightRenderable(spriteTexture);
```

Modifying the MyGame Object

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 in the `MyGame.js` file of adding variables for the new objects, initializing the objects, drawing the objects, and updating the objects, each line of code changed will not be listed. Rather, you can open the `MyGame.js` file within the `src/MyGame` folder and look at changes made in order to test the newly added light source.

Observations

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 z position to observe the illuminated circle decreases (smaller intersection area) and increases in size. Alternatively, you can 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. Another observation to take note of is 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 two blocks are not `LightRenderable` objects and thus cannot be illuminated by the defined light source.

Multiple Light Sources and Distance Attenuation

In the previous project, a single point light source was defined with the capability of illuminating a spherical volume. This type of light source is useful in many games, but it is restrictive to limit a game to only a single light source. The engine should support the illumination from multiple light sources to fulfill the design needs of different games. This shortcoming is remedied in the next project with general support for multiple light sources. The implementation principle for multiple lights remains the same as the previous project, with the modification of replacing the single light source with an array of lights. As illustrated in Figure 8-5, a new Light object will be defined, while the LightRenderable object will be modified to support an array of the Light objects. The LightShader object will define an array of ShaderLightAtIndex objects that are capable of communicating light source information to the uLights array in the GLSL LightFS fragment shader for illumination computations.

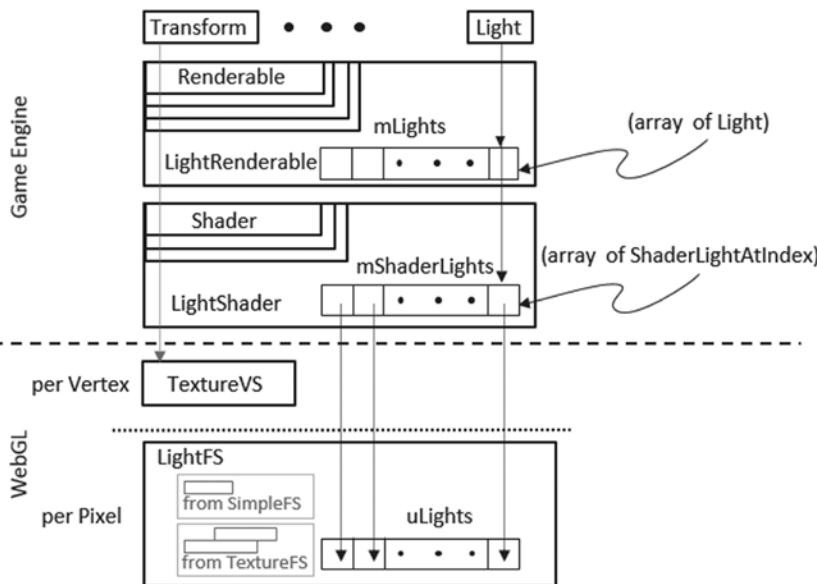


Figure 8-5. Support for multiple light sources

The point light illumination results from the previous project can be improved. You have observed that the illuminated circle disappears abruptly with a sharp illuminated bright boundary. This sudden disappearance of illumination results does not reflect real life where effects from a given light source decrease gradually over 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 disappear gradually. This gradual decrease of light illumination effect over distance is referred to as *distance attenuation*. It is a common practice to approximate distant attenuation with quadratic functions because they produce effects that resemble the real world. In general, distance attenuation can be approximated in many ways, and it is often refined to suit the needs of the game. In addition to distance attenuation, you will also implement a near cutoff distance and a far cutoff distance (that is, two distances from the light source at which the distant attenuation effect will begin and end). These two values give you control over a light source to show a fully illuminated center area with illumination drop-off occurring only at a specified distance. Lastly, a light intensity will be defined to allow dramatically different effects. For example, you can have a soft, barely noticeable light that covers a wide area or an oversaturated glowing light that is concentrated over a small area in the scene.

The Multiple Lights Project

This project demonstrates how to implement multiple point lights within a single scene. It also demonstrates how to increase the complexity of your point lights so that they are more flexible to serve a wider variety of purposes. You can see an example of this project running in Figure 8-6. The source code of this project is located in the Chapter8/8.3.MultipleLights folder.

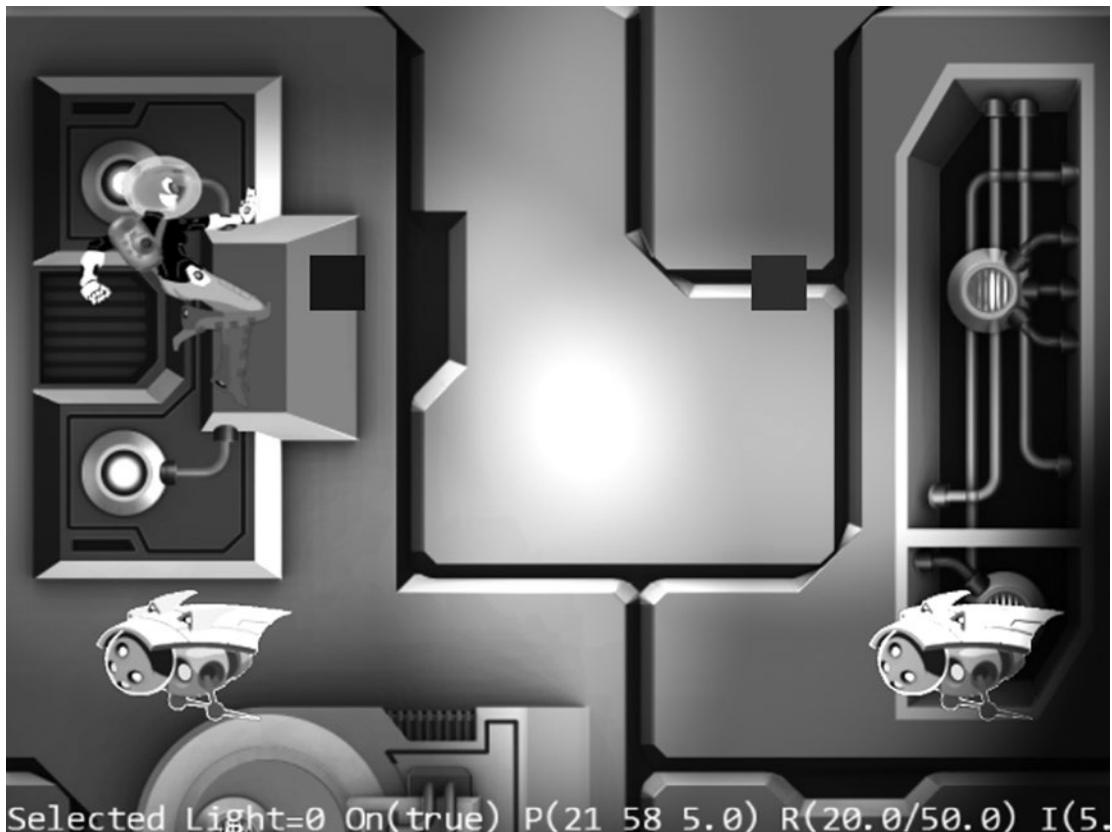


Figure 8-6. Running the Multiple Lights project

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 key:* 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 key:* Increase/decrease the intensity of the selected light
- *H key:* Toggles the selected light on/off

The goals of the project are as follows:

- To build the infrastructure for supporting multiple light sources in the engine and in GLSL shaders
- To understand and examine the distance attenuation effects of light
- To experience controlling and manipulating multiple light sources in a scene

You can find the following external resources in the assets folder: the fonts folder that contains the default system fonts and two texture images (`minion_sprite.png` and `bg.png`). The objects are sprite elements of `minion_sprite.png`, and the background is represented by `bg.png`.

Modifying the GLSL Light Fragment Shader

The LightFS fragment shader needs to be modified to support the distance attenuation, cutoffs, and multiple light sources.

1. In the `LightFS.glsl` file, remove the light variables that were added for a single light and add a struct for light information that holds the position, color, near distance, far distance, intensity, and on-off variables. With the struct defined, add a uniform array of lights to the fragment shader. Notice that a `#define` has been added to hold the number of light sources to be used. You can see these changes in the following code.

Note You can define as many lights as the hardware can support. For example, you can try increasing the number of lights to 50 and then test and measure its performance.

```
// Light information
#define kGLSLuLightArraySize 4
    // GLSL Fragment shader requires loop control
    // variable to be a constant number. This number 4
    // says, this fragment shader will _ALWAYS_ process
    // all 4 light sources.
    // *****WARNING*****
    // This number must correspond to the constant with
    // the same name defined in LightShader.js file.
    // *****WARNING*****
    // To change this number MAKE SURE: to update the
    //     kGLSLuLightArraySize
    // defined in LightShader.js file.

struct Light {
    vec4 Position;    // in pixel space!
    vec4 Color;
    float Near;       // distance in pixel space
    float Far;        // distance in pixel space
    float Intensity;
    bool IsOn;
};

uniform Light uLights[kGLSLuLightArraySize];
    // Maximum array of lights this shader supports
```

2. Next add a function called `LightEffect()` that takes a light parameter and returns a color as a result. This function calculates the distance between the light and the current fragment and determines whether it lies within the near radius, in between near and far radii, or farther than the far radius. If the fragment's current position lies within the near radius, there is no attenuation, so a value of 1 is applied. If the fragment's current position lies in between the near and far radii, then a quadratic attenuation is applied. A distance of greater than the far radius will result in no illumination from the corresponding light source, or a 0 value will be applied. You can see how this is achieved in the following code:

```
vec4 LightEffect(Light lgt) {
    vec4 result = vec4(0);
    float atten = 0.0;
    float dist = length(lgt.Position.xyz - gl_FragCoord.xyz);
    if (dist <= lgt.Far) {
        if (dist <= lgt.Near)
            atten = 1.0; // no attenuation
        else {
            // simple quadratic drop off
            float n = dist - lgt.Near;
            float d = lgt.Far - lgt.Near;
            atten = smoothstep(0.0, 1.0, 1.0-(n*n)/(d*d)); // blended attenuation
        }
    }
    result = atten * lgt.Intensity * lgt.Color;
    return result;
}
```

3. The main function iterates through all the defined light sources and calls 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;
```

```

    // tint the textured area, and leave transparent area as defined by the
    // texture
    vec3 r = vec3(lgtResults) * (1.0-uPixelColor.a) +
        vec3(uPixelColor) * uPixelColor.a;
    vec4 result = vec4(r, lgtResults.a);

    gl_FragColor = result;
}

```

Modifying the Light Object

The game engine Light object must be modified to reflect the newly added properties: near and far attenuation and intensity.

1. Modify the Lights.js constructor to include variables for near and far attenuation and intensity. You can see this achieved in the following code:

```

// Constructor
function Light() {
    this.mColor = vec4.fromValues(0.1, 0.1, 0.1, 1); // light color
    this.mPosition = vec3.fromValues(0, 0, 5); // light position in WC
    this.mNear = 5; // within Near is fully lighted
    this.mFar = 10; // farther than Far is not lighted
    this.mIntensity = 1;
    this.mIsOn = true;
}

```

2. Define the get and set accessors for the new variables. Note that the radius variable has been generalized and replaced by the near and far cutoff distances.

```

Light.prototype.setColor = function(c) { this.mColor = vec4.clone(c); };
Light.prototype.getColor = function() { return this.mColor; };

Light.prototype.set2DPosition = function(p) {
    this.mPosition = vec3.fromValues(p[0], p[1], this.mPosition[2]); };
Light.prototype.setXPos = function(x) { this.mPosition[0] = x; };
Light.prototype.setYPos = function(y) { this.mPosition[1] = y; };
Light.prototype.setZPos = function(z) { this.mPosition[2] = z; };
Light.prototype.getPosition = function() { return this.mPosition; };

Light.prototype.setNear = function(r) { this.mNear = r; };
Light.prototype.getNear = function() { return this.mNear; };

Light.prototype.setFar = function(r) { this.mFar = r; };
Light.prototype.getFar = function() { return this.mFar; };

Light.prototype.setIntensity = function(i) { this.mIntensity = i; };
Light.prototype.getIntensity = function() { return this.mIntensity; };

Light.prototype.isLightOn = function() { return this.mIsOn; };
Light.prototype.setLightTo = function(on) { this.mIsOn = on; };

```

Creating the LightSet Object

Under the `src/Engine/Lights` folder, create a new file and name it `LightSet.js`. Provide a basic interface for a light set that makes the process of working with the light array more convenient. Remember to load this new source file in `index.html`.

```
function LightSet() {
    this.mSet = [];
}

LightSet.prototype.NumLights = function() { return this.mSet.length; };

LightSet.prototype.getLightAt = function(index) {
    return this.mSet[index];
};

LightSet.prototype.AddToSet = function(light) {
    this.mSet.push(light);
};
```

Creating the ShaderLightAtIndex Object

Define the `ShaderLightAtIndex` object to send information from a `Light` object to an element in the `uLights` array in the `LightFS` GLSL fragment shader.

1. Under the `src/Engine/Shaders` folder, create a new file and name it `ShaderLightAtIndex.js`. Remember to load this new source file in `index.html`.
2. Define a constructor to set the light property references to a specific index in the `uLights` array in the fragment shader.

```
function ShaderLightAtIndex(shader, index) {
    this._setShaderReferences(shader, index);
}
ShaderLightAtIndex.prototype._setShaderReferences = function(aLightShader, index)
{
    var gl = gEngine.Core.getGL();
    this.mColorRef = gl.getUniformLocation(aLightShader,
                                           "uLights[" + index + "].Color");
    this.mPosRef = gl.getUniformLocation(aLightShader,
                                         "uLights[" + index + "].Position");
    this.mNearRef = gl.getUniformLocation(aLightShader,
                                         "uLights[" + index + "].Near");
    this.mFarRef = gl.getUniformLocation(aLightShader,
                                         "uLights[" + index + "].Far");
    this.mIntensityRef = gl.getUniformLocation(aLightShader,
                                              "uLights[" + index + "].Intensity");
    this.mIsOnRef = gl.getUniformLocation(aLightShader,
                                         "uLights[" + index + "].IsOn");
};
```

3. Implement the `loadToShader()` function to push the light's properties to the GLSL shader. Notice that this function is similar to the previous `loadToShader()` function defined in the `LightShader.js` file.

```
ShaderLightAtIndex.prototype.loadToShader = function (aCamera, aLight) {
    var gl = gEngine.Core.getGL();
    gl.uniform1i(this.mIsOnRef, aLight.isLightOn());
    if (aLight.isLightOn()) {
        var p = aCamera.wcPosToPixel(aLight.getPosition());
        var ic = aCamera.wcSizeToPixel(aLight.getNear());
        var oc = aCamera.wcSizeToPixel(aLight.getFar());
        var c = aLight.getColor();
        gl.uniform4fv(this.mColorRef, c);
        gl.uniform3fv(this.mPosRef, vec3.fromValues(p[0], p[1], p[2]));
        gl.uniform1f(this.mNearRef, ic);
        gl.uniform1f(this.mFarRef, oc);
        gl.uniform1f(this.mIntensityRef, aLight.getIntensity());
    }
};
```

4. Provide a function to update the shader's on/off property for the light.

```
ShaderLightAtIndex.prototype.switchOffLight = function() {
    var gl = gEngine.Core.getGL();
    gl.uniform1i(this.mIsOnRef, false);
};
```

Modifying the LightShader Object

You must modify the `LightShader` object to properly handle the communication between the `Light` object and the array of lights in the `LightFS` fragment shader.

1. Begin by editing the `LightShader.js` file and removing the `loadToShader()` function because the actual loading of light information to GLSL shaders will be handled by the newly defined `ShaderLightAtIndex` objects.
2. Modify the constructor to define `mLights`, which is an array of `ShaderLightAtIndex` objects to correspond to the `uLights` array defined in the `LightFS` fragment shader. It is important to note that the `mLights` and `uLights` arrays must be the exact same size. You can see this in the following code:

```
function LightShader(vertexShaderPath, fragmentShaderPath) {
    // Call super class constructor
    SpriteShader.call(this, vertexShaderPath, fragmentShaderPath);

    this.mLights = null; // lights from the renderable

    //*****WARNING*****
    // this number MUST correspond to the GLSL uLight[] array size
    // (for LightFS.gls and IllumFS.gls)
    //*****WARNING*****
```

```

this.kGLSLuLightArraySize = 4; // <-- make sure this is
                                // the same as LightFS.glsl
this.mShaderLights = [];
for (var i = 0; i<this.kGLSLuLightArraySize; i++) {
    var ls = new ShaderLightAtIndex(this.mCompiledShader, i);
    this.mShaderLights.push(ls);
}
}
gEngine.Core.inheritPrototype(LightShader, SpriteShader);

```

3. Modify the activateShader() function to iterate and load the contents of each ShaderLightAtIndex object to the LightFS shader by calling the corresponding loadToShader() function. Recall that the GLSL fragment shader requires the for loop control variable to be a constant. This implies that all elements of the uLights array will be processed on each LightFS invocation, and thus it is important to ensure all unused lights are switched off. This is ensured by the last while loop in the following code:

```

// Overriding the Activation of the shader for rendering
LightShader.prototype.activateShader = function(pixelColor, aCamera) {
    // first call the super class's activate
    SpriteShader.prototype.activateShader.call(this, pixelColor, aCamera);

    // now push the light information to the shader
    var numLight = 0;
    if (this.mLights !== null) {
        while (numLight < this.mLights.length) {
            this.mShaderLights[numLight].loadToShader(
                aCamera, this.mLights[numLight]);
            numLight++;
        }
    }
    // switch off the left over ones.
    while (numLight < this.kGLSLuLightArraySize) {
        this.mShaderLights[numLight].switchOffLight(); // switch off unused lights
        numLight++;
    }
};

```

4. Make a simple modification to the setLight() function so that it becomes setLights() and handles the array rather than the single light.

```

LightShader.prototype.setLights = function(l) {
    this.mLights = l;
};

```

Modifying the LightRenderable Object

You can now modify the LightRenderable object to support multiple light sources.

1. In the LightRenderable constructor, replace the single light reference variable with an array.

```
function LightRenderable(myTexture) {
    SpriteAnimateRenderable.call(this, myTexture);
    Renderable.prototype._setShader.call(
        this, gEngine.DefaultResources.getLightShader());

    // here is the light source
    this.mLights = []; // allocates a new array
}
```

2. Make sure to update the draw function to reflect the change to multiple light sources.

```
LightRenderable.prototype.draw = function(aCamera) {
    this.mShader.setLights(this.mLights);
    SpriteAnimateRenderable.prototype.draw.call(this, aCamera);
};
```

3. Define the corresponding accessor functions for the light array.

```
LightRenderable.prototype.getLightAt = function(index) {
    return this.mLights[index];
};
LightRenderable.prototype.addLight = function(l) {
    this.mLights.push(l);
};
```

Testing the Light Sources with MyGame

With multiple lights support properly integrated 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 each light. Because of the scope of this MyGame, you will divide the light instantiation and controls into separate files to maintain the readability of the source code. To avoid redundancy and repetitive code listings, the details to the straightforward implementations are not shown.

1. Modify the `MyGame.js` file in the `src/MyGame` folder to reflect the changes to the constructor, initialize function, draw function, and update function. All these changes revolve around handling multiple lights through a light set.
2. In the `src/MyGame` folder, create the new file `MyGame_Lights.js` to instantiate and initialize the lights. Remember to load this new source file in `index.html`.
3. In the `src/MyGame` folder, create the new file `MyGame_lightControls.js` to implement the controls of the lights. Remember to load this new source file in `index.html`.

Observations

Run the project to examine the implementation. Try selecting the lights with the 0, 1, 2, and 3 keys and toggling the selected light on/off. Notice that all lights illuminate the background, while the hero is illuminated only by lights 0 and 3, the left minion is illuminated only by lights 1 and 3, and the right minion is illuminated only by lights 2 and 3. Move the Hero object with the WASD keys to observe how the illumination on the object changes as she is moved through the near and far radii of light source 0. Select and 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. The two constant color squares are in the scene to confirm that nonilluminated objects can still be rendered.

Diffuse Reflection and Normal Mapping

You can now place or move many light sources and control the illumination or shading at localized regions. However, if you run the previous project and move the lights around while paying attention to the vertical boundaries of the geometric blocks in the background, you will notice the peculiar effect that the illumination along these boundaries changes uniformly when a light position moves across it. You are observing boundary surfaces being illuminated by light sources that seem to be spatially behind the surface! Although visually odd, this is to be expected in a 2D world. The vertical boundaries 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.

As illustrated by the left drawing in Figure 8-7, a surface normal vector, a surface normal, or a normal vector is the vector that is perpendicular to a given surface element. The right drawing of Figure 8-7 shows that in 3D space the surface normal vectors of an object describe the shape or contour of the object.

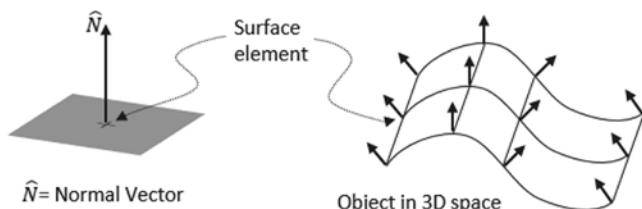


Figure 8-7. Surface normal vectors of an object

A human's observation of light illumination is the result of visible energy from the light sources reflecting off object surfaces and reaching the eyes. A diffuse, or Lambertian, surface reflects light energy uniformly in all directions. Examples of diffuse surfaces include typical printer papers or matte painted surfaces. Figure 8-8 shows a light source illuminating three diffuse surface element positions, A, B, and C. First, notice that the direction from the position being illuminated toward the light source is defined as the position's light vector, \hat{L} . It is important to note that the direction of the \hat{L} vector is always toward the light source and that this is a normalized vector with a magnitude of 1. Figure 8-8 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 light vector, or $\hat{N} \cdot \hat{L}$.

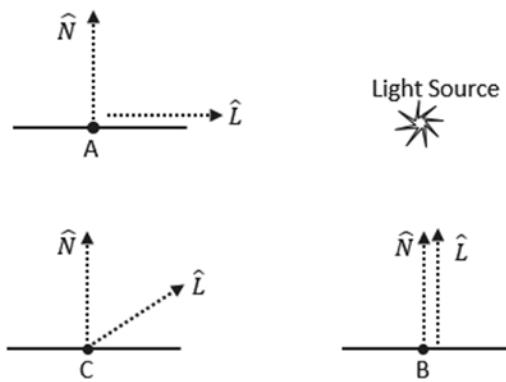


Figure 8-8. Normal and light vectors and diffuse illumination

Displaying the $\hat{N} \cdot \hat{L}$ computation results, or the diffuse component, cues 3D shape contours for the human vision system. For example, Figure 8-9 shows a sphere and torus (doughnut shape object) with (the left image) and without (the right image) the corresponding diffuse components. Clearly, in both cases, the 3D contour of the objects are captured by the left versions of the image with the $\hat{N} \cdot \hat{L}$ computation.

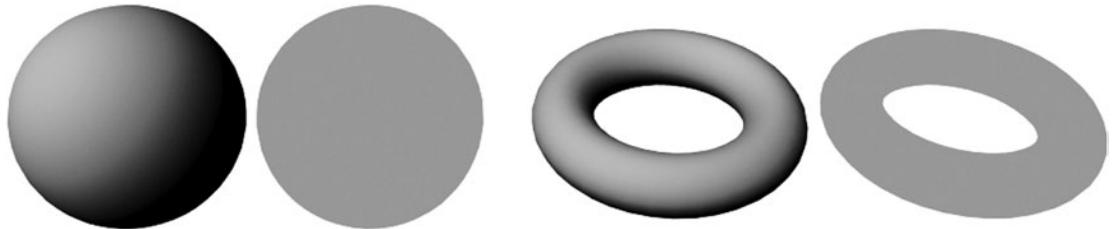


Figure 8-9. Examples of 3D objects with and without diffuse component

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: the vector in the z direction. This lack of distinct normal vectors for objects implies that it is not possible to compute the diffuse component of objects. Fortunately, similar to how texture mapping addresses the limitation of each geometry having only a single color, normal mapping can resolve the problem of each geometry having only a single normal vector. Figure 8-10 shows the idea behind normal mapping where in addition to the color texture image a corresponding normal texture image is required. The left image of Figure 8-10 is a typical color texture image, and the two right images are zoomed images of the highlighted square on the left image. Notice once again that two images are involved in normal mapping: the color texture image where the RGB channels of the texture record the color of objects (bottom of the right image of Figure 8-10) and a corresponding normal texture image where the RGB channels record the x, y, and z values of the normal vector for the corresponding object in the color texture (top of the right image).

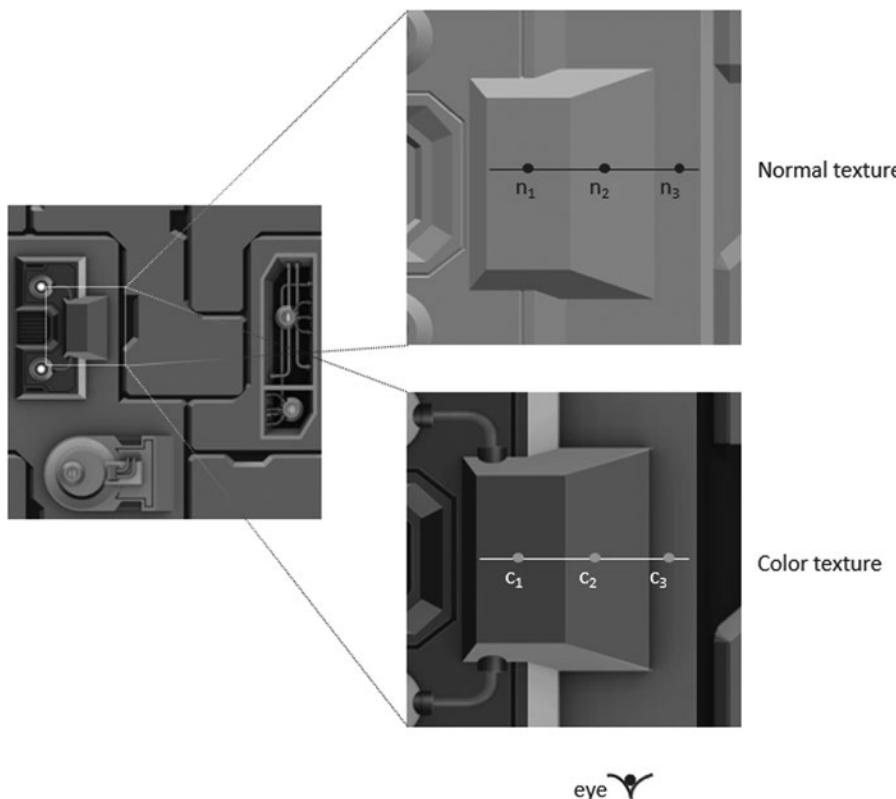


Figure 8-10. Normal mapping with two texture images: the normal and the color texture

Figure 8-11 captures the view of the three corresponding positions labeled on the right images of Figure 8-10 (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 8-11 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 8-11 shows that the RGB components of the normal texture records the normal vector xyz values of objects at the corresponding color texture positions. The top layer of Figure 8-11 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.

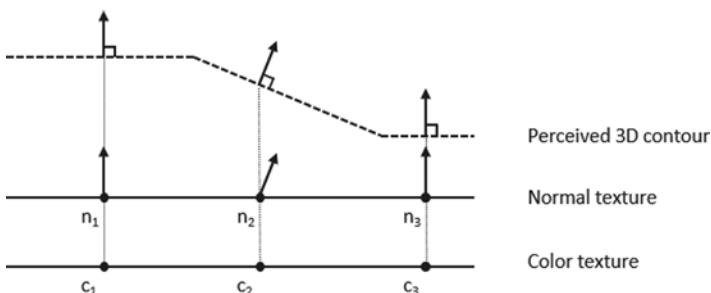


Figure 8-11. Normal mapping with two texture images: the normal and the color texture

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 used for computing the diffuse component. When computing light illumination, the $\hat{N} \cdot \hat{L}$ computation is referred to as *diffuse* computation, lighting, or illumination. The mathematic term $\hat{N} \cdot \hat{L}$ is referred to as the *diffuse term*.

In the previous project, you expanded the engine to support multiple light sources with individual light to model the real world closely. In this section, you will define the `IllumShader` object to generalize the `LightShader` object to support the computation of the diffuse component based on normal mapping.

The Normal Maps and Illumination Shaders Project

This project demonstrates how to integrate normal mapping into your game engine and use the results to compute the diffuse component of objects. You can see an example of this project running in Figure 8-12. The source code of this project is located in the Chapter8/8.4.NormalMapsAndIlluminationShaders folder.

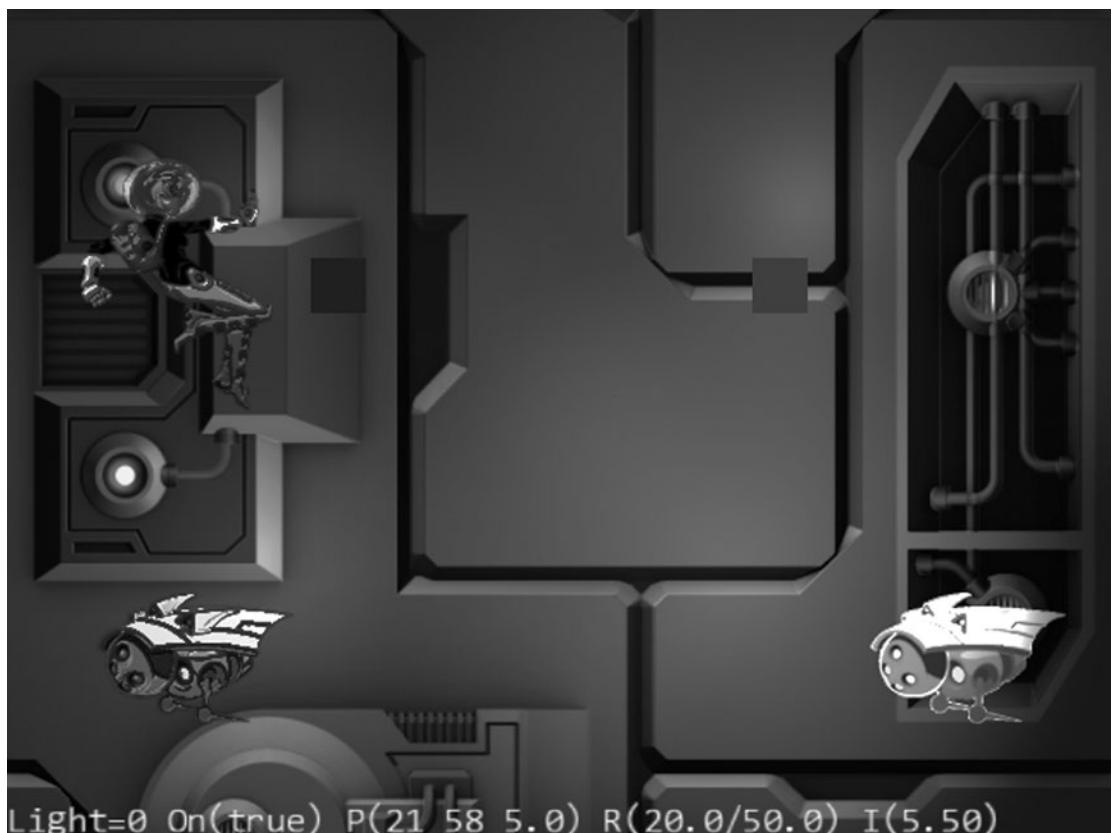


Figure 8-12. Running the Normal Maps and Illumination Shaders project

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 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
- *K/L key*: Increases/decreases the intensity of the selected light
- *H key*: Toggles the selected light on/off

The goals of the project are as follows:

- To understand and work with normal maps
- To implement normal maps as textures in the game engine
- To implement GLSL shaders that support diffuse component illumination
- To examine the diffuse component illumination of $\hat{N} \cdot \hat{L}$

You can find the following external resource files in the assets folder: the fonts folder that 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 automatically based on the `minion_sprite.png` image from <http://cpetry.github.io/NormalMap-Online/>.

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 `LightFS.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.

1. Begin by copying and pasting `LightFS.glsl` and naming the new file `IllumFS.glsl` within the `src/GLSLShaders` folder.
2. Edit the `IllumFS.glsl` file and add a `sampler2D` object, `uNormalSampler`, to sample the normal map.

```
precision mediump float;

// The object that fetches data from texture.
// Must be set outside the shader.
uniform sampler2D uSampler;
uniform sampler2D uNormalSampler;
```

```

// Color of pixel
uniform vec4 uPixelColor;
uniform vec4 uGlobalAmbientColor; // this is shared globally
uniform float uGlobalAmbientIntensity;

// Light information
#define kGLSLuLightArraySize 4
struct Light {
    vec4 Position; // in pixel space!
    vec4 Color;
    float Near; // distance in pixel space
    float Far; // distance in pixel space
    float Intensity;
    bool IsOn;
};
uniform Light uLights[kGLSLuLightArraySize];
// Maximum array of lights this shader supports

varying vec2 vTexCoord;

```

3. Modify the `LightEffect()` function to receive a normal vector parameter, `N`. This normal vector `N` is assumed to be normalized with a magnitude of 1 and will be used in the diffuse component $\hat{N} \cdot \hat{L}$ computation. Include code to compute the \hat{L} vector, remember to normalize the vector, and use the result of $\hat{N} \cdot \hat{L}$ to scale the light attenuation accordingly as follows:

```

vec4 LightEffect(Light lgt, vec3 N) {
    vec4 result = vec4(0);
    float atten = 0.0;
    vec3 L = lgt.Position.xyz - gl_FragCoord.xyz;
    float dist = length(L);
    if (dist <= lgt.Far) {
        if (dist <= lgt.Near)
            atten = 1.0; // no attenuation
        else {
            // simple quadratic drop off
            float n = dist - lgt.Near;
            float d = lgt.Far - lgt.Near;
            atten = smoothstep(0.0, 1.0, 1.0-(n*n)/(d*d)); // blended attenuation
        }
        L = L / dist; // To normalize L
        // Not calling the normalize() function to avoid re-computing
        // the "dist". This is computationally more efficient.
        float NdotL = max(0.0, dot(N, L));
        atten *= NdotL;
    }
    result = atten * lgt.Intensity * lgt.Color;
    return result;
}

```

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 direction of the surface element at that given point. 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.

Note Normal maps can be created in a variety of different layouts where x or y might need to be flipped in order to properly represent the surface geometries desired. This entirely depends upon the tool or artist that created the map.

```

void main(void) {
    // simple tint based on uPixelColor setting
    vec4 textureMapColor = texture2D(uSampler, vTexCoord);
    vec4 normal = texture2D(uNormalSampler, vTexCoord);
        // using the same coordinate as the sprite texture!
    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);
            }
        }
    }
    lgtResult *= textureMapColor;

    // tint the textured area, and
    // leave transparent area as defined by the texture
    vec3 r = vec3(lgtResult) * (1.0-uPixelColor.a) +
        vec3(uPixelColor) * uPixelColor.a;
    vec4 result = vec4(r, lgtResult.a);

    gl_FragColor = result;
}

```

Creating the IllumShader Object

With the GLSL shader now supporting normal maps, you can create the JavaScript `IlluminationShader` object to interface to it.

1. Under the `src/Engine/Shaders` folder, create a new file and name it `IllumShader.js`. Remember to load this new source file in `index.html`.
2. Create the constructor by first subclassing from the `LightShader` to take advantage of the functionality related to light sources and define a variable, `mNormalSamplerRef`, to maintain a reference to the normal sampler in the GLSL shader.

```
function IllumShader(vertexShaderPath, fragmentShaderPath) {
    // Call super class constructor
    LightShader.call(this, vertexShaderPath, fragmentShaderPath);

    // reference to the normal map sampler
    var gl = gEngine.Core.getGL();
    this.mNormalSamplerRef = gl.getUniformLocation(
        this.mCompiledShader, "uNormalSampler");
}
gEngine.Core.inheritPrototype(IllumShader, LightShader);
```

3. Override and extend the `activateShader()` function by binding the normal texture sampler reference to WebGL texture unit 1. So far, you have been working with the color texture sampler that is bounded to the default texture unit of 0. In this way, the WebGL texture system can work with two active textures: units 0 and 1. As will be discussed, the `TextureShader` object must now explicitly bind the color texture to unit 0, and in `gEngine.Textures`, it is important to configure the WebGL to activate the appropriate texture units for the corresponding purpose: color versus normal texture mapping.

```
IllumShader.prototype.activateShader = function(pixelColor, aCamera) {
    // first call the super class's activate
    LightShader.prototype.activateShader.call(this, pixelColor, aCamera);
    var gl = gEngine.Core.getGL();
    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
};
```

Note WebGL supports simultaneous activation of multiple texture units during rendering. Depending on the graphics card capability, up to 32 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 TextureShader Object

So far, you have been working with the default binding of the color texture map to WebGL texture unit 0. With the addition of the normal texture, you now need to explicitly bind your color texture to WebGL texture unit 0. Fortunately, this is a straightforward change.

1. Modify the constructor to include and initialize a reference to the sampler location, as shown in the following code:

```
function TextureShader(vertexShaderPath, fragmentShaderPath) {
    // Call super class constructor
    SimpleShader.call(this, vertexShaderPath, fragmentShaderPath);

    // reference to aTextureCoordinate within the shader
    this.mShaderTextureCoordAttribute = null;
    this.mSamplerRef = null; // reference to the uSampler, when using only texture,
        // this is not necessary,
        // with NormalMap, we must do this.

    // get the reference of uSampler and aTextureCoordinate within the shader
    var gl = gEngine.Core.getGL();
    this.mSamplerRef = gl.getUniformLocation(this.mCompiledShader, "uSampler");
    this.mShaderTextureCoordAttribute =
        gl.getAttributeLocation(this.mCompiledShader, "aTextureCoordinate");
}
```

2. Add a line to the activateShader() function to bind the texture to unit 0, as shown in the following code:

```
// Overriding the Activation of the shader for rendering
TextureShader.prototype.activateShader = function(pixelColor, aCamera) {
    // first call the super class's activate
    SimpleShader.prototype.activateShader.call(this, pixelColor, aCamera);

    // now our own functionality: enable texture coordinate array
    var gl = gEngine.Core.getGL();
    gl.bindBuffer(gl.ARRAY_BUFFER, gEngine.VertexBuffer.getGLTexCoordRef());
    gl.enableVertexAttribArray(this.mShaderTextureCoordAttribute);
    gl.vertexAttribPointer(this.mShaderTextureCoordAttribute, 2, gl.FLOAT,
        false, 0, 0);
    gl.uniform1i(this.mSamplerRef, 0); // <-- binds to texture unit 0
};
```

Creating the IllumRenderable Object

You can now create your renderable object to leverage the illumination shader.

1. Begin by creating a new file under the `src/Engine/Renderables` folder and naming it `IllumRenderable.js`. Remember to load this new source file in `index.html`.

2. Create a constructor to subclass from the LightRenderable object and define a `mNormalMap` instance variable to record the normal map ID. The IllumRenderable object works with two texture maps: `myTexture` for color texture map and `myNormalMap` for normal mapping. Note that these two texture maps share the same texture coordinates defined in `mTexCoordBuffer` in the SpriteShader superclass. This assumes that the geometry of the object is depicted in the color texture map and the normal texture map is derived to capture the contours of the object, which is almost always the case.

```
function IllumRenderable(myTexture, myNormalMap) {
    LightRenderable.call(this, myTexture);
    Renderable.prototype._setShader.call(
        this, gEngine.DefaultResources.getIllumShader());

    // here is the normal map resource id
    this.mNormalMap = myNormalMap;

    // Normal map texture coordinate will reproduce the corresponding sprite sheet
    // This means, the normal map MUST be based on the sprite sheet
}
gEngine.Core.inheritPrototype(IllumRenderable, LightRenderable);
```

3. Next override the `draw()` function to activate the normal map before calling the base class's `draw()` method.

```
IllumRenderable.prototype.draw = function (aCamera) {
    gEngine.Textures.activateNormalMap(this.mNormalMap);
    // Here the normal map texture coordinate is copied from those of
    // the corresponding sprite sheet
    LightRenderable.prototype.draw.call(this, aCamera);
};
```

Modifying the Engine

You have to update the engine to support the new texture map, shader, and renderable objects.

Defining the Default in Engine_DefaultResources

You must modify the `Engine_DefaultResources.js` file to define the default instance of the `IllumShader` object.

1. Define a constant file path and variable for the newly created fragment shader.

```
// Illumination Shader
var kIllumFS = "src/GLSLShaders/IllumFS.gsls";
// Path to the Illumination FragmentShader
var mIllumShader = null;
```

2. Modify the `_createShaders()` function to instantiate an `IllumShader` object.

```
var _createShaders = function(callBackFunction) {
    //...
    mIllumShader = new IllumShader(kIllumVS, kIllumFS);
    callBackFunction();
};
```

3. Add a simple accessor for the illumination shader.

```
var getIllumShader = function() { return mIllumShader; };
```

4. Modify the `initialize()` function to load the text file that defines the illumination shader.

```
var initialize = function(callBackFunction) {
    //...

    // Illumination Shader
    gEngine.TextFileLoader.LoadTextFile(kIllumFS,
        gEngine.TextFileLoader.eTextFileType.eTextFile);

    // load default font
    gEngine.FONTS.LoadFont(kDefaultFont);

    gEngine.ResourceMap.setLoadCompleteCallback(function()
        {_createShaders(callBackFunction)} );
};
```

5. Modify the `cleanUp()` function to unload the text file that defines the illumination shader.

```
// Unload all resources
var cleanUp = function() {
    //...
    mIllumShader.cleanUp();
    // Illumination Shader
    gEngine.TextFileLoader.unloadTextFile(kIllumFS);
    // default font
    gEngine.FONTS.unloadFont(kDefaultFont);
};
```

6. Export the get accessor in the public function list.

```
var mPublic = {
    initialize: initialize,
    getConstColorShader: getConstColorShader,
    // ...
    getIllumShader: getIllumShader,
    getDefaultFont: getDefaultFont,
    // ...
};
```

Configuring WebGL Texture Units in Engine_Textures

The engine texture support in the `Engine_Textures.js` file must be updated to support configuring the WebGL texture units accordingly: color texture binding to unit 0 and normal texture binding to unit 1.

1. Add a line to the `activateTexture()` function to specify the texture unit 0.

Recall that this function activates the color texture mapping functionality.
The `gl.TEXTURE0` constant informs WebGL to bind to the texture unit 0.

```
var activateTexture = function(textureName) {
    var gl = gEngine.Core.getGL();
    var texInfo = gEngine.ResourceMap.retrieveAsset(textureName);

    // Binds our texture reference to the current webGL texture functionality
gl.activeTexture(gl.TEXTURE0);
    gl.bindTexture(gl.TEXTURE_2D, texInfo.mGLTexID);

    //...
};
```

2. Add a function to activate the normal map texture and bind this texture to WebGL texture unit 1 with the `TEXTURE1` constant.

```
// texture 1 is always normal map for this game engine
var activateNormalMap = function(textureName) {
    var gl = gEngine.Core.getGL();
    var texInfo = gEngine.ResourceMap.retrieveAsset(textureName);

    // Binds our texture reference to the current webGL texture functionality
gl.activeTexture(gl.TEXTURE1);
    gl.bindTexture(gl.TEXTURE_2D, texInfo.mGLTexID);

    // To prevent texture wrappings
    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.
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.LINEAR);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR_MIPMAP_LINEAR);
};
```

3. Remember to export the activation of the normal map in the public function list.

```
var mPublic = {
    // ...
    activateTexture: activateTexture,
activateNormalMap: activateNormalMap,
    // ...
};
```

Testing the Normal Map

Testing the newly integrated normal map functionality must include the verification that the non-normal mapped simple color texture is working correctly. To accomplish this, the background, hero, and left minion will be created as the newly defined `IllumRenderable` object, while the right minion will remain a `LightRenderable` object.

Modifying the Hero and the Minion

The `Hero` and `Minion` objects should be modified to support the newly defined `IllumRenderable` object.

1. Modify the `Hero`'s constructor to utilize the `IllumRenderable`.

```
function Hero(spriteTexture, normalMap) {
    this.kDelta = 0.3;

    this.mDye = new IllumRenderable(spriteTexture, normalMap);
    this.mDye.setColor([1, 1, 1, 0]);
    this.mDye.getXform().setPosition(15, 50);
    this.mDye.getXform().setSize(18, 24);
    this.mDye.setElementPixelPositions(0, 120, 0, 180);
    GameObject.call(this, this.mDye);
}
```

2. Modify the `Minion`'s constructor to utilize the `IllumRenderable`, and notice that depending on whether a normal texture map is present, a `Minion` can be either an `IllumRenderable` or a `LightRenderable`.

```
function Minion(spriteTexture, normalMap, atX, atY) {
    this.kDelta = 0.2;

    if (normalMap === null) {
        this.mMinion = new LightRenderable(spriteTexture);
    } else {
        this.mMinion = new IllumRenderable(spriteTexture, normalMap);
    }

    this.mMinion.setColor([1, 1, 1, 0]);
    this.mMinion.getXform().setPosition(atX, atY);
    this.mMinion.getXform().setSize(18, 14.4);
    // first element pixel position:
    //    top-right 512 is top of image, 0 is right of image
    this.mMinion.setSpriteSequence(512, 0,
        204, 164,      // widthxheight in pixels
        5,              // number of elements in this sequence
        0);            // horizontal padding in between
    this.mMinion.setAnimationType(SpriteAnimateRenderable.eAnimationType.eAnimateSwing);
    this.mMinion.setAnimationSpeed(30);
        // show each element for mAnimSpeed updates

    GameObject.call(this, this.mMinion);
}
```

Modifying MyGame

You can now modify MyGame to test and display your implementation of the illumination shader. Modify the `MyGame.js` file in the `src/MyGame` 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 minimum; as such, the details are not shown here.

Observations

With the project now complete, 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 (most notably, in the background where the composing geometric blocks now appear to be individually defined 3D shapes). The fact that the normal maps for the Hero and left Minion objects are generated automatically can be observed with their slightly pixelated and rough appearances. Select one of the light sources, such as light 2, and move the light position with the arrow keys. Take note of the boundary edges of the geometric blocks in the background image; these edges are surfaces facing either horizontally or vertically, whereas the corresponding normal vector directions point either toward the x or y direction. As the light position moves across such a boundary, the sign of the $\hat{N} \cdot \hat{L}$ term would flip, and the corresponding surface illumination would undergo drastic changes (from dark to lit, or vice versa). In this way, with the normal map and diffuse computation, you have turned a static background image into a background that is defined by complex 3D geometric shapes. Try moving the other light sources and observe the illumination changes on all the objects as the light sources move across them.

Specular Reflection and Materials

The diffuse lighting you have implemented is suitable for simulating the illumination of matted surfaces such as typical printer papers, many painted interior walls, or even a traditional blackboard. The Phong illumination model extends this simple diffuse lighting by introducing a specular term to simulate the reflection of the light source across a shiny surface. Figure 8-13 illustrates that given a shiny or reflective surface like a polished floor or 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 shiny surface is referred to as *specular reflection*, *specular highlight*, or *specularity*.

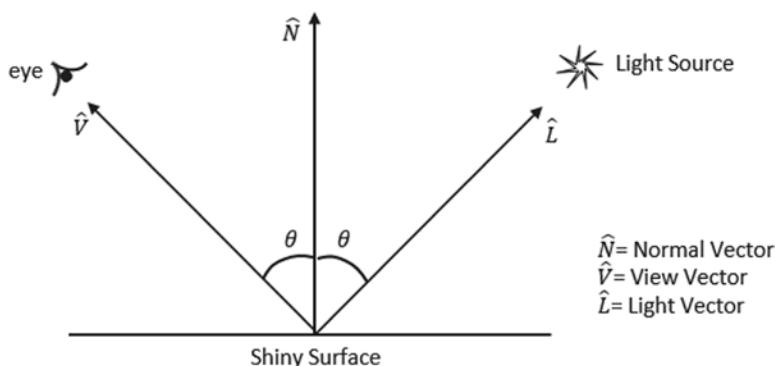


Figure 8-13. Specularity: the reflection of the light source

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 8-14, 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} , the less likely you will observe the light reflection. In fact, you know that when α , the angle between \hat{V} and \hat{R} , 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.

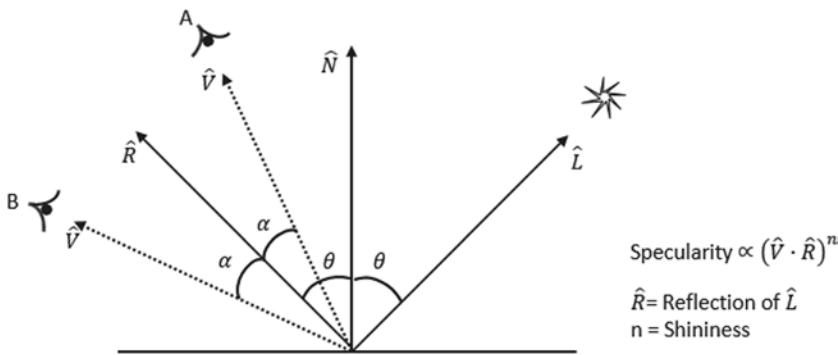


Figure 8-14. The Phong specularity model

The Phong illumination model simulates the characteristic 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 will roll off. 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 8-15, the right sphere has a n value of 0, the middle sphere has a n value of 5, and the right sphere's n value is 30.

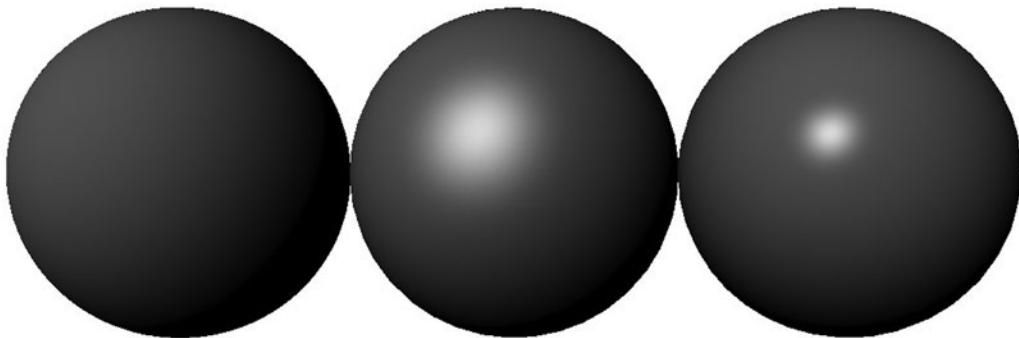


Figure 8-15. Specularity and shininess (n)

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 8-16, \hat{H} is the halfway vector, which is the vector halfway between 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 in computing the \hat{H} vector. The halfway vector will be used to approximate specularity in your implementation.

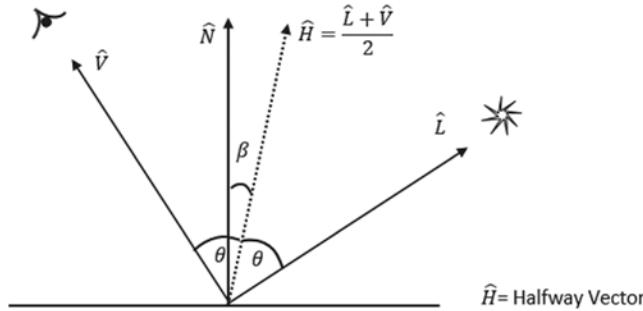


Figure 8-16. The halfway vector

As illustrated in Figure 8-17, 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 participating elements: the global ambient lighting and the light source. The materials property of an object are 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. The three terms of the Phong illumination model are as follows:

- *The ambient term:* $K_a + I_a C_a$
- *The diffuse term:* $I_L C_L K_d (\hat{N} \cdot \hat{L})$
- *The specular term:* $I_L C_L K_s (\hat{N} \cdot \hat{H})^n$

The Phong Illumination Model

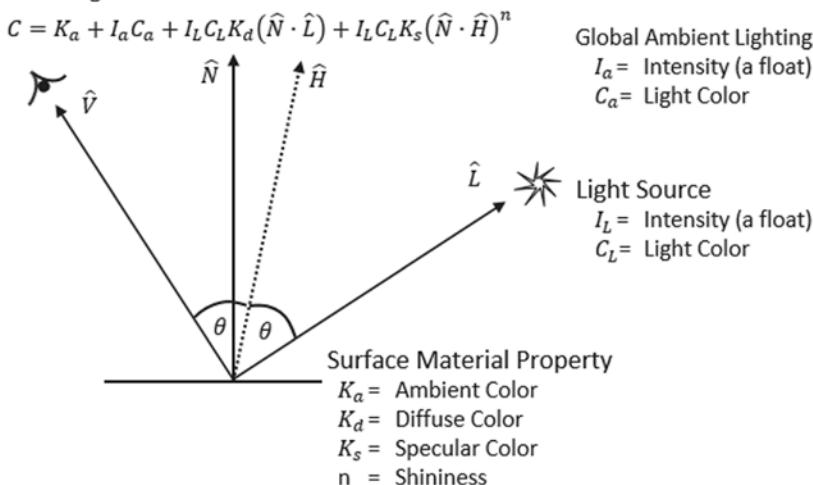


Figure 8-17. The Phong illumination model

Note that the first two terms, the ambient and diffuse terms, have been covered in the previous examples. The `IllumFS` 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 `IllumFS` GLSL shader with the engine support in the `IllumShader/IllumRenderable` object pair.

Integration of Material in the Game Engine and GLSL Shaders

To implement the Phong illumination model, a `Material` object that corresponds to the surface material property in Figure 8-17 must be defined and referenced by each `IllumRenderable` object that is to be shaded by the corresponding `IllumFS` GLSL shader. Figure 8-18 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 `IllumFS` GLSL fragment shader.

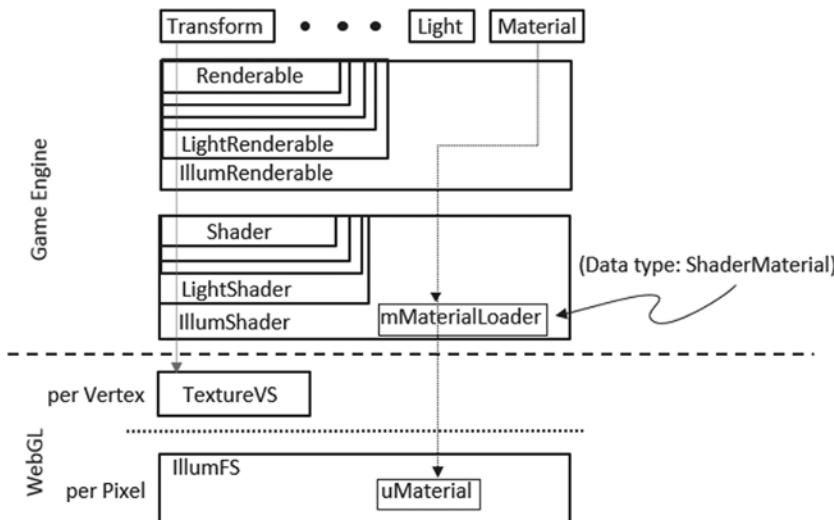


Figure 8-18. Support for material

The Material and Specularity Project

This project demonstrates the implementation of the Phong illumination model utilizing the normal map and the camera's position. It also implements a system that stores and forwards per-`Renderable` object material properties to the GLSL shader for the Phong lighting computation. You can see an example of the project running in Figure 8-19. The source code of this project is located in the `Chapter8/8.5.MaterialAndSpecularity` folder.

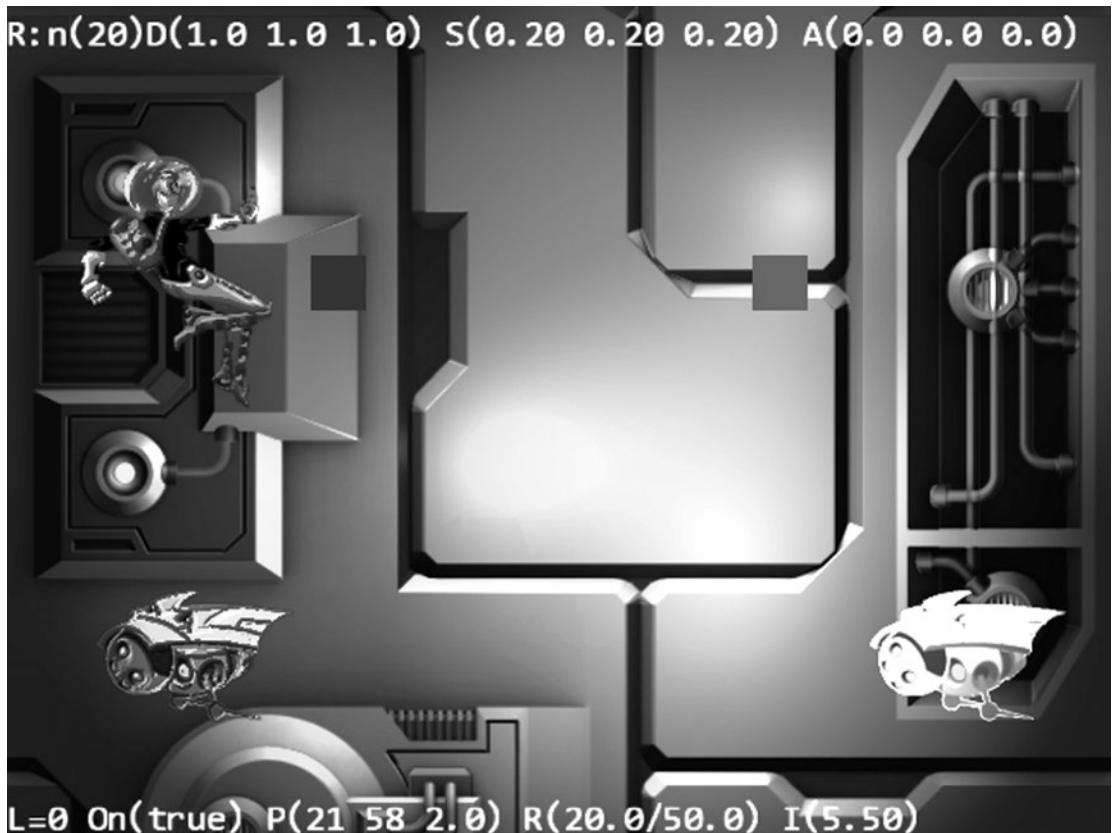


Figure 8-19. Running the Material and Specularity project

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
 - 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
 - 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 specular reflection and the Phong specular term
- To implement specular highlight illumination in GLSL shaders
- To understand and experience the control of Material of illuminated objects
- To examine specular highlights in illuminated images

In the assets folder you can find the same set of external resource files as in the previous project: the fonts folder that contains the default system fonts, two texture images, 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`.

Modifying the GLSL Illumination Fragment Shader

As in the previous projects, you will begin with implementing the actual illumination model in the GLSL fragment shader.

1. Edit the `IllumFS.glsl` file and define a variable, `uCameraPosition`, for storing the camera position. This position is used to compute the \hat{V} vector. In addition, create a material struct and a corresponding variable, `uMaterial`, for storing the per-object material properties. Note the correspondence between the variable names K_a , K_d , K_s , and n and the terms in the Phong illumination model in Figure 8-17.

```
// for supporting a simple Phong-like illumination model
uniform vec3 uCameraPosition; // for computing the V-vector

// material properties
struct Material {
    vec4 Ka;      // simple boosting of color
    vec4 Kd;      // Diffuse
    vec4 Ks;      // Specular
    float Shininess; // this is the "n"
};
uniform Material uMaterial;
```

2. As in the previous project, create the `LightAttenuation()` function to calculate the distance attenuation of the light.

```
// Computes the L-vector, and returns attenuation
float LightAttenuation(Light lgt, dist) {
    float atten = 0.0;
    if (dist <= lgt.Far) {
        if (dist <= lgt.Near)
            atten = 1.0; // no attenuation
```

```

        else {
            // simple quadratic drop off
            float n = dist - lgt.Near;
            float d = lgt.Far - lgt.Near;
            atten = smoothstep(0.0, 1.0, 1.0-(n*n)/(d*d)); // blended attenuation
        }
    }
    return atten;
}

```

3. Define the `SpecularResult()` and `DiffuseResults()` functions to calculate the specular and diffuse terms. The \hat{V} vector, V , is computed by subtracting `uCameraPosition` from the current fragment coordinate, `gl_FragCoord`. It is important to observe that this operation is performed in the pixel space, and the `IllumShader/IllumRenderable` object pair must transform the WC camera position to pixel space before sending over the information. In addition, notice that the texture map color is accumulated in the diffuse and not the specular term.

```

vec4 SpecularResult(vec3 N, vec3 L) {
    vec3 V = normalize(uCameraPosition - gl_FragCoord.xyz);
    vec3 H = (L + V) * 0.5;
    return uMaterial.Ks * pow(max(0.0, dot(N, H)), uMaterial.Shininess);
}
vec4 DiffuseResult(vec3 N, vec3 L, vec4 textureMapColor) {
    return uMaterial.Kd * max(0.0, dot(N, L)) * textureMapColor;
}

```

4. Implement a `ShadedResult()` function to compute and accumulate the diffuse and specular terms. Notice that $lgt.\text{Intensity}$, I_L in Figure 8-17, and $lgt.\text{Color}$, C_L in Figure 8-17, are factored out and multiplied to the sum of diffuse and specular results. The scaling by the light distance attenuation, `atten`, is the only variation between this implementation and the diffuse/specular terms listed in Figure 8-17.

```

vec4 ShadedResult(Light lgt, vec3 N, vec4 textureMapColor) {
    vec3 L = lgt.Position.xyz - gl_FragCoord.xyz;
    float dist = length(L);
    L = L / dist;
    float atten = LightAttenuation(lgt, dist);
    vec4 diffuse = DiffuseResult(N, L, textureMapColor);
    vec4 specular = SpecularResult(N, L);
    vec4 result = atten * lgt.Intensity * lgt.Color * (diffuse + specular);
    return result;
}

```

5. 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 `IllumFS.glsL` file from the previous project; the only important differences are highlighted in bold in the following:

```

void main(void) {
    // simple tint based on uPixelColor setting
    vec4 textureMapColor = texture2D(uSampler, vTexCoord);
    vec4 normal = texture2D(uNormalSampler, vNormalMapCoord);
    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 shadedResult = uMaterial.Ka +
    (textureMapColor * uGlobalAmbientColor * uGlobalAmbientIntensity);

    // now decide if we should illuminate by the light
    if (textureMapColor.a > 0.0) {
        for (int i=0; i<4; i++) {
            if (uLights[i].IsOn) {
                shadedResult += ShadedResult(uLights[i], N, textureMapColor);
            }
        }
    }

    // tint the textured area, and leave transparent area as defined by the texture
    vec3 tintResult = vec3(shadedResult) * (1.0-uPixelColor.a) +
                      vec3(uPixelColor) * uPixelColor.a;
    vec4 result = vec4(tintResult, shadedResult.a);

    gl_FragColor = result;
}

```

Creating the Material Object

As described, a simple `Material` object is required to store the per-object material property for the Phong illumination model.

1. Create a new file under the `src/Engine` folder and name it `Material.js`. Remember to load this new source file in `index.html`.
2. Define a constructor to initialize the variables as defined in the surface material property in Figure 8-17. Notice that ambient, diffuse, and specular (`Ka`, `Kd`, and `Ks`) are colors, while shininess is a floating point number.

```

function Material() {
    this.mKa = vec4.fromValues(0.0, 0.0, 0.0, 0);
    this.mKs = vec4.fromValues(0.2, 0.2, 0.2, 1);
}

```

```

        this.mKd = vec4.fromValues(1.0, 1.0, 1.0, 1);
        this.mShininess = 20;
    }
}

```

- Provide straightforward get and set accessors to these properties.

```

Material.prototype.setAmbient = function(a) { this.mKa = vec4.clone(a); };
Material.prototype.getAmbient = function() { return this.mKa; };

Material.prototype.setDiffuse = function(d) { this.mKd = vec4.clone(d); };
Material.prototype.getDiffuse = function() { return this.mKd; };

Material.prototype.setSpecular = function(s) { this.mKs = vec4.clone(s); };
Material.prototype.getSpecular = function() { return this.mKs; };

Material.prototype.setShininess = function(s) { this.mShininess = s; };
Material.prototype.getShininess = function() { return this.mShininess; };

```

Defining the ShaderMaterial Object

Define a new `ShaderMaterial` object to communicate the contents of `Material` to the GLSL `IllumFS` shader.

- Create a new file under the `src/Engine/Shaders` folder and name it `ShaderMaterial.js`. Remember to load this new source file in `index.html`.
- Define a constructor to initialize the variables as references to the ambient, diffuse, specular, and shininess in the `IllumFS` GLSL shader.

```

function ShaderMaterial(aIllumShader) {
    // reference to the normal map sampler
    var gl = gEngine.Core.getGL();
    this.mKaRef = gl.getUniformLocation(aIllumShader, "uMaterial.Ka");
    this.mKdRef = gl.getUniformLocation(aIllumShader, "uMaterial.Kd");
    this.mKsRef = gl.getUniformLocation(aIllumShader, "uMaterial.Ks");
    this.mShineRef = gl.getUniformLocation(aIllumShader, "uMaterial.Shininess");
}

```

- Define the `loadToShader()` function to push the content of a `Material` to the GLSL shader.

```

ShaderMaterial.prototype.loadToShader = function (aMaterial) {
    var gl = gEngine.Core.getGL();
    gl.uniform4fv(this.mKaRef, aMaterial.getAmbient());
    gl.uniform4fv(this.mKdRef, aMaterial.getDiffuse());
    gl.uniform4fv(this.mKsRef, aMaterial.getSpecular());
    gl.uniform1f(this.mShineRef, aMaterial.getShininess());
};

```

Modifying the IllumShader Object

Recall that the `IllumShader` object is the engine's interface to the corresponding GLSL `IllumFS` shader. Modify the `IllumShader` object to define an instance of the `ShaderMaterial` object to load the contents of the `Material` object.

1. Edit the `IllumShader.js` file to define variables for `Material` and `ShaderMaterial`. Recall that `ShaderMaterial` is the `Material` loader. In addition, define the variables for the camera position and the reference to the camera uniform location in the GLSL shader.

```
function IllumShader(vertexShaderPath, fragmentShaderPath) {
    var gl = gEngine.Core.getGL();

    // Call super class constructor
    LightShader.call(this, vertexShaderPath, fragmentShaderPath);

    // this is the material property of the renderable
    this.mMaterial = null;
    this.mMaterialLoader = new ShaderMaterial(this.mCompiledShader);

    // Reference to the camera position
    this.mCameraPos = null; // points to a vec3
    this.mCameraPosRef = gl.getUniformLocation(
        this.mCompiledShader, "uCameraPosition");

    //...
}
```

2. Modify the `activateShader()` function to include the loading of the material and camera position to the shader.

```
IllumShader.prototype.activateShader = function(pixelColor, aCamera) {
    //...
    this.mMaterialLoader.loadToShader(this.mMaterial);
    gl.uniform3fv(this.mCameraPosRef, this.mCameraPos);
};
```

3. Define the `setMaterialAndCameraPos()` function to set the corresponding variables for Phong illumination computation.

```
IllumShader.prototype.setMaterialAndCameraPos = function(m, p) {
    this.mMaterial = m;
    this.mCameraPos = p;
};
```

Modifying the IllumRenderable Object

You need to modify the `IllumRenderable` object to support a material property. This is a straightforward change.

1. Edit the `IllumRenderable.js` file and modify the constructor to instantiate a new `Material` object.

```
function IllumRenderable(myTexture, myNormalMap) {
    //...

    // Material for this renderable
    this.mMaterial = new Material();
}
```

2. Modify the `draw()` function to set the material in camera position before the actual rendering. Notice the call to `aCamera.getPosInPixelSpace()`: the camera position obtained is properly transformed into pixel space.

```
IllumRenderable.prototype.draw = function(aCamera) {
    gEngine.Textures.ActivateNormalMap(this.mNormalMap);
    // Here the normal map texture coordinate is copied from those of
    // the corresponding sprite sheet
    this.mShader.setMaterialAndCameraPos(this.mMaterial,
                                         aCamera.getPosInPixelSpace());
    LightRenderable.prototype.draw.call(this, aCamera);
};
```

3. Define a simple accessor for the material.

```
IllumRenderable.prototype.getMaterial = function() { return this.mMaterial; };
```

Modifying the Camera Object

As you have seen in the `IllumFS` GLSL shader implementation, the camera position required for computing the \hat{V} vector must be in pixel space. The `Camera` object must be modified to provide such information. Since the `Camera` object stores its position in WC space, this position must be transformed to pixel space for each `IllumRenderable` object rendered. There may be a large number of `IllumRenderable` objects in a scene, and the camera position cannot be changed once rendering begins. These observations suggest that a pixel space camera position should be computed and cached.

1. Edit the `Camera.js` file and add a `vec3` to your `PerRenderCache` function to cache the camera's position in pixel space.

```
function PerRenderCache() {
    this.mWCToPixelRatio = 1; // WC to pixel transformation
    this.mCameraOrgX = 1; // Lower-left corner of camera in WC
    this.mCameraOrgY = 1;
    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 depth information is required for illumination computation.

```
this.kCameraZ = 10; // This is for illumination computation
```

3. In step B4 of the `setupViewProjection()` function, call the `wcPosToPixel()` function to transform the camera's position to 3D pixel space and cache the computed results.

```
// Step B4: compute and cache per-rendering information
this.mRenderCache.mWCtoPixelRatio = this.mViewport.Camera.eViewport.eWidth /
    this.getWCWidth();
this.mRenderCache.mCameraOrgX = center[0] - (this.getWCWidth() / 2);
this.mRenderCache.mCameraOrgY = center[1] - (this.getWCHeight() / 2);
var p = this.wcPosToPixel(this.getWCCenter());
this.mRenderCache.mCameraPosInPixelSpace[0] = p[0];
this.mRenderCache.mCameraPosInPixelSpace[1] = p[1];
this.mRenderCache.mCameraPosInPixelSpace[2] =
    this.fakeZInPixelSpace(this.kCameraZ);
```

4. Define a simple get accessor function for the camera position in pixel space.

```
Camera.prototype.getPosInPixelSpace = function () {
    return this.mRenderCache.mCameraPosInPixelSpace; };
```

Testing Specular Reflection

You can now test your implementation of the Phong illumination model and observe the effects of altering object material property and specularity. Since the background, Hero, and left Minion are already instances of the `IllumRenderable` object, these three objects will exhibit specularity by default. To ensure prominence of specular reflection, the specular material property, `Ks`, of the background object is set to bright red in the `initialize()` function.

```
MyGame.prototype.initialize = function () {
    // ...

    // the Background
    var bgR = new IllumRenderable(this.kBg, this.kBgNormal);
    bgR.setElementPixelPositions(0, 1024, 0, 1024);
    bgR.getXform().setSize(100, 100);
    bgR.getXform().setPosition(50, 35);
    // set background material properties
    bgR.getMaterial().setShininess(100);
    bgR.getMaterial().setSpecular([1, 0, 0, 1]);

    // ...
}
```

A new function, `_selectCharacter()`, is defined to allow the user to work with the material property of either the `Hero` or the left `Minion` object. The file `MyGame_MaterialControl.js` implements the actual user interaction for controlling the selected material property.

Observations

You can run the project and interactively control the object's material property. For example, by default the material property of the `Hero` object is selected. You can try changing the diffuse RGB components by pressing the E/R, T/Y, or U/I keys. Notice that you can press multiple keys simultaneously to change multiple color channels at the same time.

The normal map of the background image is carefully generated and thus is best for examining specularity effects. Press the 1 key to select the first light source, and press the right arrow key to move the light toward the right. You should see the illuminated circle of the light moving toward the right from its initial position that is slightly toward the left of the center of the window. After the light crosses the center of the window and as you continue to move it toward the right, notice there will be a red specular highlight showing up at around the right boundary of the background image. This is the specular reflection of light source number 1 reflecting off the background image reaching the current camera position.

Do experiment with selecting and manipulating the material property of the left `Minion` object and moving the other light sources around to observe the red specular highlight on the background. Notice that because the normal maps of the `Hero` and `Minion` objects are generated automatically and do not fully represent the geometric characteristics of these objects, it can be tricky to observe specular reflection off them.

Light Source Types

So far your game engine supports the illumination by many instances of a single type of light, a point light. A point light, behaving much like a lightbulb in the real world, illuminates from a single position with near and far radii where objects can be fully, partial, or not lit at all by the light source. There are two other light types that are popular in most game engines: the directional light and the spotlight.

A directional light models the sun rays where the light appears to arrive in parallel from the same direction, instead of a single position, and does not seem to suffer from distance attenuation. While in reality the sun casts light in all directions, from the perspective of the earth, the light rays from the sun are practically parallel because of the great distance. 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.

A spotlight models a desk lamp with a cone-shape lampshade. As illustrated in Figure 8-20, 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.

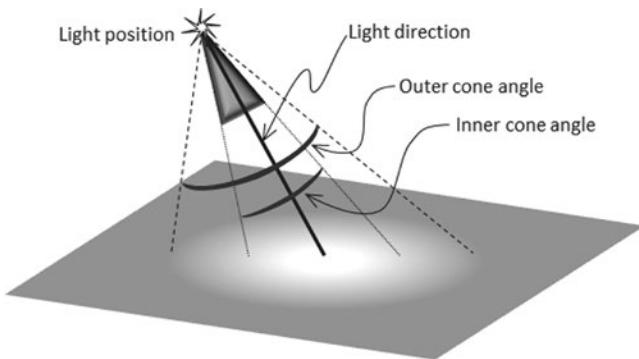


Figure 8-20. A spotlight and its parameters

Note In illustrative diagrams, like Figure 8-20, for clarity purposes light directions are usually represented by lines extending from the light position toward the environment. These lines are usually for illustrative purposes and do not carry mathematic meanings. These illustrative diagrams are contrasted with vector diagrams that explain illumination computations, like Figures 8-13 and 8-14. In illumination vector diagrams, all vectors always point away from the position to be illuminated, and all vectors are assumed to be normalized vectors with a magnitude of 1.

The Directional and Spot Lights Project

This project demonstrates how to integrate directional lights and a spotlight into your engine to support a wider range of illumination effects. You can see an example of the project running in Figure 8-21. The source code of this project is located in the Chapter8/8.6.DirectionalAndSpotLights folder.

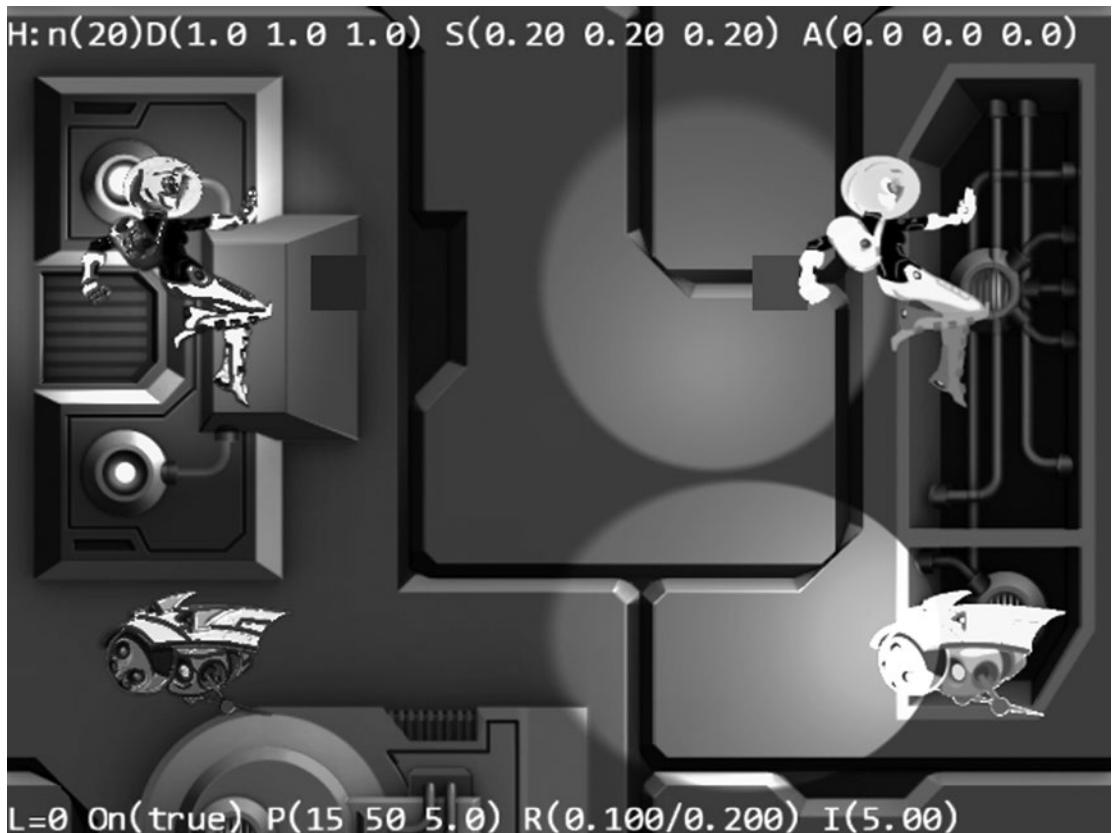


Figure 8-21. Running the Directional and Spotlights project

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
 - Arrow keys with spacebar pressed: Change the direction of the currently selected light
 - Z/X key: Increases/decreases the light z position
 - C/V and B/N keys: Increase/decrease the inner and outer cone angles of the selected light
 - 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_o , 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

In the assets folder you can find the same set of external resource files as in the previous project: the fonts folder that contains the default system fonts, two texture images, 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`.

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 `IllumFS.gls1` and defining constants for the three light types. Notice that to support proper communications between the WebGL 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 within the shader 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. Notice that, as will be detailed next, instead of the actual angle values, the cosine of the inner and outer angles are stored in the struct to facilitate 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 {
    vec4 Position; // in pixel space!
    vec4 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 ePointLight, eDirectionalLight, eSpotLight
};
```

3. Define an `AngularDropOff()` function to compute the angular attenuation for the spotlight.

```
float AngularDropOff(Light lgt, vec3 lgtDir, vec3 L) {
    float atten = 0.0;
    float cosL = dot(lgtDir, L);
    float num = cosL - lgt.CosOuter;
    if (num > 0.0) {
        if (cosL > lgt.CosInner)
            atten = 1.0;
        else {
            float denom = lgt.CosInner - lgt.CosOuter;
            atten = smoothstep(0.0, 1.0, pow(num/denom, lgt.DropOff));
        }
    }
    return atten;
}
```

The parameter `lgt` is the reference to the spotlight in light struct, `lgtDir` is the direction of the spotlight (or `Light.Direction`), and `L` is the light vector of the current position to be illuminated. Note that since the dot product of normalized vectors is the cosine of the angle between the vectors, it is convenient to represent all angular displacements by their corresponding cosine values and to carry out the computations based on cosines of the actual angular displacements. Figure 8-22 illustrates the parameters involved in angular attenuation computation.

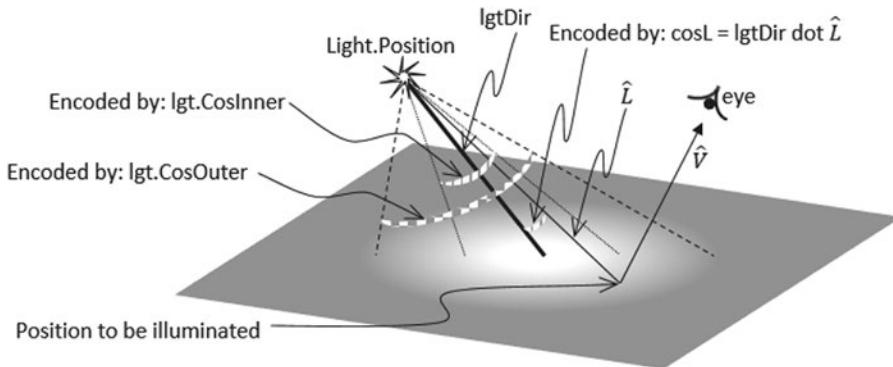


Figure 8-22. Computing the angular attenuation of a spotlight

- The `cosL` is the dot product of \mathbf{L} with \mathbf{lgtDir} ; it records the angular displacement of the position currently being illuminated.
 - The `num` variable stores the difference between `cosL` and `cosOuter`. A negative `num` would mean the position currently being illuminated is outside the outer cone, and the angular attenuation would result in no contribution. Thus, no further computation would be required.
 - If the point to be illuminated is within the inner cone, `cosL` would be less than `lgt.CosInner`, and the angular attenuation would result in full contribution.
 - If the point to be illuminated is in between the inner and outer cone angles, use the `smoothstep()` function to compute a drop-off.
4. Rename the `LightAttenuation()` function from the previous project to `DistanceDropOff()`.

```
float DistanceDropOff(Light lgt, float dist) {
    float atten = 0.0;
    if (dist <= lgt.Far) {
        if (dist <= lgt.Near)
            atten = 1.0; // no attenuation
        else {
            // simple quadratic drop off
            float n = dist - lgt.Near;
            float d = lgt.Far - lgt.Near;
            atten = smoothstep(0.0, 1.0, 1.0-(n*n)/(d*d)); // blended attenuation
        }
    }
    return atten;
}
```

5. Modify the ShadedResults() function to handle each separate case of light source type before combining the results into a color.

```

vec4 ShadedResult(Light lgt, vec3 N, vec4 textureMapColor) {
    float aAtten = 1.0, dAtten = 1.0;
    vec3 lgtDir = -normalize(lgt.Direction.xyz);
    vec3 L; // light vector
    float dist; // distant to light
    if (lgt.LightType == eDirectionalLight) {
        L = lgtDir;
    } else {
        L = lgt.Position.xyz - gl_FragCoord.xyz;
        dist = length(L);
        L = L / dist;
    }
    if (lgt.LightType == eSpotLight) {
        // spotlight: do angle dropoff
        aAtten = AngularDropOff(lgt, lgtDir, L);
    }
    if (lgt.LightType != eDirectionalLight) {
        // both spot and point light has distant dropoff
        dAtten = DistanceDropOff(lgt, dist);
    }
    vec4 diffuse = DiffuseResult(N, L, textureMapColor);
    vec4 specular = SpecularResult(N, L);
    vec4 result = aAtten * dAtten * lgt.Intensity * lgt.Color * (diffuse + specular);
    return result;
}

```

Modifying the GLSL Light Fragment Shader

You can now modify the GLSL LightFS fragment shader to support the two new light types. The modifications involved are remarkably similar to the case of IllumFS discussed previously, 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 attenuate the light. Please refer to the LightFS.gls1 source code file for details of the implementation.

Modifying the Light Object

You must extend the Light object to support the parameters of the two new light types.

1. Edit the Light.js file and define 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 IllumFS and LightFS shaders.

```

// **** WARNING: The following enumerate values must be identical to
// the values of
//   ePointLight, eDirectionalLight, eSpotLight
// defined in LightFS.gls1 and IllumFS.gls1

```

```
Light.eLightType = Object.freeze({
    ePoint: 0,
    eDirectionalLight: 1,
    eSpotLight: 2
});
```

2. Modify the constructor to define and initialize the new variables that correspond to the parameters of directional light and spotlight.

```
function Light() {
    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
    this.mOuter = 0.3;
    this.mIntensity = 1;
    this.mDropOff = 1;
    this.mLightType = Light.eLightType.ePointLight;
    this.mIsOn = true;
}
```

3. Define the get and set accessors for the new variables. The exhaustive listing of these functions is not shown here. Please refer to the `Light.js` source code file for details.

Modifying the `ShaderLightAtIndex` Object

Recall that the `ShaderLightAtIndex` object is responsible for loading the values in a light source to the GLSL fragment shader. This object must be refined to support the new light source parameters that correspond to directional lights and spotlights.

1. Edit the `ShaderLightAtIndex.js` file and modify the `setShaderReferences()` function to obtain and save the references to the newly added light properties, as shown in the following code:

```
ShaderLightAtIndex.prototype._setShaderReferences = function (aLightShader, index) {
    var gl = gEngine.Core.getGL();
    this.mColorRef = gl.getUniformLocation(aLightShader,
        "uLights[" + index + "].Color");
    this.mPosRef = gl.getUniformLocation(aLightShader,
        "uLights[" + index + "].Position");
    this.mDirRef = gl.getUniformLocation(aLightShader,
        "uLights[" + index + "].Direction");
    this.mNearRef = gl.getUniformLocation(aLightShader, "uLights[" + index + "].Near");
    this.mFarRef = gl.getUniformLocation(aLightShader, "uLights[" + index + "].Far");
    this.mInnerRef = gl.getUniformLocation(aLightShader,
        "uLights[" + index + "].CosInner");
```

```

this.mOuterRef = gl.getUniformLocation(aLightShader,
                                      "uLights[" + index + "].CosOuter");
this.mIntensityRef = gl.getUniformLocation(aLightShader,
                                         "uLights[" + index + "].Intensity");
this.mDropOffRef = gl.getUniformLocation(aLightShader,
                                         "uLights[" + index + "].DropOff");
this.mIsOnRef = gl.getUniformLocation(aLightShader, "uLights[" + index + "].IsOn");
this.mLightTypeRef = gl.getUniformLocation(aLightShader,
                                         "uLights[" + index + "].LightType");
};

}

```

2. 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.

```

ShaderLightAtIndex.prototype.loadToShader = function (aCamera, aLight) {
    var gl = gEngine.Core.getGL();
    gl.uniform1i(this.mIsOnRef, aLight.isLightOn());
    if (aLight.isLightOn()) {
        var p = aCamera.wcPosToPixel(aLight.getPosition());
        var n = aCamera.wcSizeToPixel(aLight.getNear());
        var f = aCamera.wcSizeToPixel(aLight.getFar());
        var c = aLight.getColor();
        gl.uniform4fv(this.mColorRef, c);
        gl.uniform3fv(this.mPosRef, vec3.fromValues(p[0], p[1], p[2]));
        gl.uniform1f(this.mNearRef, n);
        gl.uniform1f(this.mFarRef, f);
        gl.uniform1f(this.mInnerRef, 0.0);
        gl.uniform1f(this.mOuterRef, 0.0);
        gl.uniform1f(this.mIntensityRef, aLight.getIntensity());
        gl.uniform1f(this.mDropOffRef, 0);
        gl.uniform1i(this.mLightTypeRef, aLight.getLightType());

        if (aLight.getLightType() === Light.eLightType.ePointLight) {
            gl.uniform3fv(this.mDirRef, vec3.fromValues(0, 0, 0));
        } else {
            // either spot or directional lights: must compute direction
            var d = aCamera.wcDirToPixel(aLight.getDirection());
            gl.uniform3fv(this.mDirRef, vec3.fromValues(d[0], d[1], d[2]));
            if (aLight.getLightType() === Light.eLightType.eSpotLight) {
                gl.uniform1f(this.mInnerRef, Math.cos(0.5 * aLight.getInner()));
                // stores cosine of half of inner angle
                gl.uniform1f(this.mOuterRef, Math.cos(0.5 * aLight.getOuter()));
                // stores cosine of half of outer cone
                gl.uniform1f(this.mDropOffRef, aLight.getDropOff());
            }
        }
    }
};

```

Note, for `mInnerRef` and `mOuterRef`, the cosine of half the angle is actually computed and passed. Half angles are used because they capture the angular displacements from the light direction. This optimization relieves the GLSL fragment shaders from computing the cosine of these angles for every invocation.

Modifying the Camera Transform Object

Directional lights and spotlights require a light direction, and the GLSL `IllumFS` and `LightFS` shaders expect this direction to be specified in pixel space. Edit the `Camera_Xform.js` file of the `Camera` object to define the `wcDirToPixel()` function to transform a direction from WC to pixel space.

```
Camera.prototype.wcDirToPixel = function (d) { // d is a vec3 direction in WC
    // Convert the position to pixel space
    var x = d[0] * this.mRenderCache.mWCToPixelRatio;
    var y = d[1] * this.mRenderCache.mWCToPixelRatio;
    var z = d[2];
    return vec3.fromValues(x, y, z);
};
```

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; `MyGame_Lights.js` is modified to create all three light types, and `MyGame_LightControl.js` is modified to support the manipulation of the direction of the selected light when the arrow and space keys are pressed simultaneously. The listing to these simple changes are not shown here; please refer to the source code files for details of the implementation.

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 pressing 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 specular reflections in red. Now, press the H key to switch off the directional light and observe as the entire scene becomes darker. Without any kinds of attenuation, directional lights can be used as effective tools for brightening the entire scene.

Press the 2 or 3 key to select one of the spotlights. Once again, by holding the spacebar while taking turns pressing the left/right or up/down keys, 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.

Try experimenting with the different light settings, including overlapping the light illumination regions and setting the light intensities to negative numbers. While impossible in the physical world, negative intensity lights are completely valid options in a game world.

Shadow Simulation

Shadow is the result of light being obstructed. As an everyday phenomenon, shadow is something you observe but probably do not give much thought to. However, shadow plays a vital role in a human's vision system. For example, the shadows of objects convey important cues of relative sizes, depths, distances, orderings, and so on. In video games, proper simulation of shadows not only can increase the quality of appearance of the games but can significantly increase the fidelity of the games. For example, you can use shadows to properly convey the distance between two game objects or the height that the hero is jumping.

Shadows can be simulated by determining the visibility between the position to be illuminated and each of the light source positions in the environment. Computationally, this is an expensive operation because general visibility determination is an $O(n)$ operation, where n is the number of objects in the scene. Algorithmically, this is a challenging problem because the visibility computation needs to occur within the fragment shader during illumination computation. In this section, you will learn about simulating shadows using a dedicated shadow caster and receiver to facilitate the rendering of the shadow based on the WebGL stencil buffer.

Figure 8-23 shows an example where a game wants to cast the shadow of the Hero object on the minion and yet not on the background. In this case, the background object will not participate in the shadow simulation computation and thus will not receive the shadow.

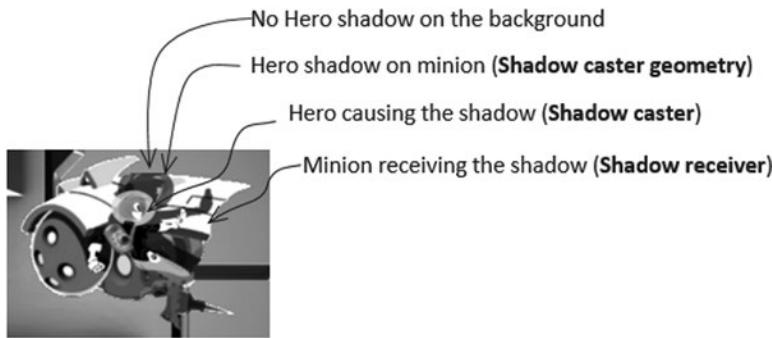


Figure 8-23. Hero casting shadow on the minion but not on the background

To properly simulate and render the shadow in Figure 8-23, as illustrated in Figure 8-24, there are three important elements.

- *Shadow caster:* This is the object that causes the shadow. In the Figure 8-23 example, the Hero object is the shadow caster.
- *Shadow receiver:* This is the object that the shadow appears on. In the Figure 8-23 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 8-23 example, the dark imprint of the hero appearing on the minion behind the actual hero object is the shadow caster geometry.

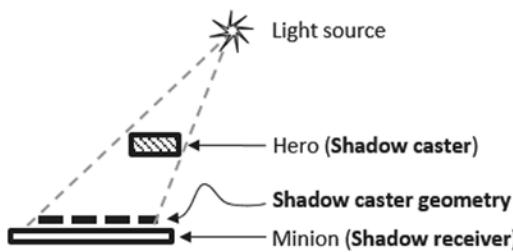


Figure 8-24. The three participating elements of shadow simulation: the caster, the caster geometry, and the receiver

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 8-23, 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. As illustrated in Figure 8-25, the challenging problem of this simple simulation occurs when the shadow caster geometry extends beyond the bounds of the shadow receiver. This situation can be observed in Figure 8-23: the top portion of the hero helmet shadow extends beyond the bounds of the minion and is not drawn.

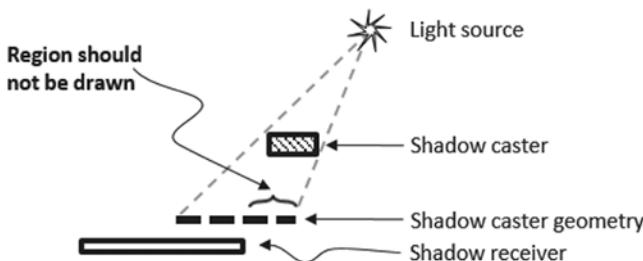


Figure 8-25. Shadow caster extends beyond the bounds of shadow receiver

Fortunately, the WebGL stencil buffer is designed specifically to resolve these types of situations. The WebGL stencil buffer can be configured as a buffer 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 only pixels in the canvas that can be drawn on will be those with corresponding stencil buffer pixels that are switched on. Figure 8-26 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 the triangular region that corresponds to the stencil triangle appearing in the canvas (bottom layer). In this way, the stencil buffer acts exactly like a stencil over the canvas where only the on-regions can be drawn on.

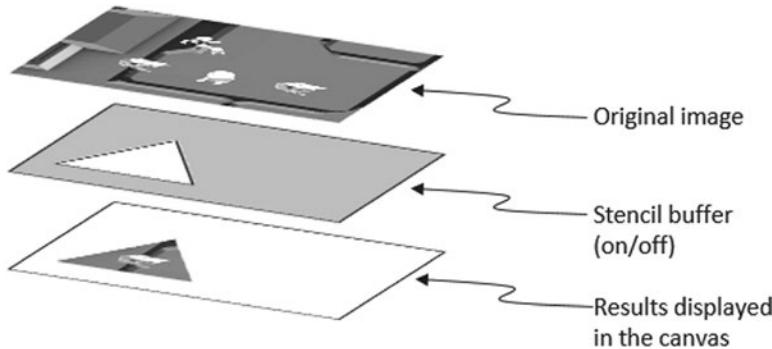


Figure 8-26. The WebGL stencil buffer

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 8-23 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 is a fake and virtual simulation. 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.

The Shadow Simulation Algorithm

Given the WebGL stencil buffer, the shadow simulation and rendering algorithm can now be outlined as follows:

```

Given a shadowReceiver
  A: Draw the shadowReceiver to the canvas as usual

  // Stencil operations to enable the region for drawing shadowCaster
  B1: Initialize all stencil buffer pixels to off
  B2: Switch on the stencil buffer pixels that correspond to the shadowReceiver object
  B3: Enable stencil buffer checking

  // Compute shadowCaster geometries and draw them on the shadowReceiver
  C: For each shadowCaster of this shadowReceiver
    D: For each shadow casting light source
      D1: Compute the shadowCaster geometry
      D2: Draw the shadowCaster geometry

```

The previous listing renders the shadow receiver and all the shadow caster geometries without rendering the actual shadow caster objects. The B1, B2, and B3 steps switch on the stencil buffer pixels that correspond to the shadow receiver; this is similar to switching on the white triangle in Figure 8-26, enabling the region that can be drawn on. The loops of steps C and D point out that a separate geometry must be computed for each shadow casting light source. By the time step D1 draws the shadow caster geometry, with

the stencil buffer containing the shadow receiver imprint and checking enabled, only pixels occupied by the shadow receiver will be drawn on in the canvas.

The Shadow Shaders Project

This project demonstrates how to implement and integrate the shadow simulation algorithm into your game engine. You can see an example of the project running in Figure 8-27. The source code of this project is located in the Chapter8/8.7.ShadowShaders folder.



Figure 8-27. Running the Shadow Shaders project

The controls of the project are as follows:

- WASD keys: Move both of the hero characters on the screen
- Lighting controls:
 - Number keys 0, 1, 2, and 3: Select the corresponding light source
 - Arrow keys: Move the currently selected light
 - Arrow keys with spacebar pressed: Change the direction of the currently selected light
 - Z/X key: Increases/decreases the light z position

- *C/V and B/N keys*: Increase/decrease the inner and outer cone angles of the selected light
- *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 simulated by 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

In the assets folder, you can find the same set of external resource files as in the previous project: the fonts folder that contains the default system fonts, two texture images, 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`.

Create 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.

Creating the GLSL Shadow Caster Fragment Shader

The `GLSLShadowCasterFS` fragment shader is the shader for drawing the shadow caster geometries.

1. Under the `src/GLSLShaders` folder, make a copy of the `IllumFS.gls1` file and name it `ShadowCasterFS.gls1`.
2. Keep the light type constants and `Light` struct definitions (not shown), and define new constants: `kMaxShadowOpacity` as how opaque shadows should be and `kLightStrengthCutOff` as a cutoff where a light with intensity less than this value will not cast shadows.

```
// ... the same as IllumFS.gls1

#define kMaxShadowOpacity 0.7      // max of shadow opacity
#define kLightStrengthCutOff 0.05 // any less will not cause shadow

// ...
```

3. Leave the `AngularDropOff()` and `DistanceDropOff()` functions the same (not shown) and create a `LightStrength()` function to compute the strength of a given light source. This function is similar to the `ShadedResult()` function of the `IllumFS` shader, except that this function computes the light strength arriving at the position to be illuminated instead of a shaded color.

```
float LightStrength() {
    float aAtten = 1.0, dAtten = 1.0;
    vec3 lgtDir = -normalize(uLights[0].Direction.xyz);
    vec3 L; // light vector
    float dist; // distant to light
    if (uLights[0].LightType == eDirectionalLight) {
        L = lgtDir;
    } else {
        L = uLights[0].Position.xyz - gl_FragCoord.xyz;
        dist = length(L);
        L = L / dist;
    }
    if (uLights[0].LightType == eSpotLight) {
        // spotlight: do angle dropoff
        aAtten = AngularDropOff(lgtDir, L);
    }
    if (uLights[0].LightType != eDirectionalLight) {
        // both spot and point light has distant dropoff
        dAtten = DistanceDropOff(dist);
    }
    float result = aAtten * dAtten;
    return result;
}
```

4. Compute 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 shadow caster geometry's color 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) {
    vec4 texFragColor = texture2D(uSampler, vTexCoord);
    float lgtStrength = LightStrength();
    if (lgtStrength < kLightStrengthCutOff)
        discard;
    vec3 shadowColor = lgtStrength * uPixelColor.rgb;
    shadowColor *= uPixelColor.a * texFragColor.a;
    gl_FragColor = vec4(shadowColor, kMaxShadowOpacity * lgtStrength * texFragColor.a);
}
```

Creating the GLSL Shadow Receiver Fragment Shader

The GLSL `ShadowReceiverFS` 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, and the shader returning any value in `gl_FragColor` will switch the corresponding pixel to on. For this reason, transparent receiver fragments must be discarded.

1. Under the `src/GLSLShaders` folder, create a new file and name it `ShadowReceiverFS.gls1`.
2. Define a `sampler2D` object such that the shadow receiver object's color texture map can be properly sampled. In addition, define the constant `kSufficientlyOpaque`. Fragments with opacity of less than this value 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.
// Must be set outside the shader.
uniform sampler2D uSampler;

// The "varying" keyword is for signifying that the texture coordinate will be
// interpolated and thus varies.
varying vec2 vTexCoord;

#define kSufficientlyOpaque      0.1
```

Note that to facilitate engine Shader object code reuse, the variable names of `uSampler` and `vTexCoord` must not be changed. These correspond to the variables names defined in `TextureFS.gls1`, and the game engine can use the existing `SpriteShader` to facilitate the loading of information to this shader.

3. Implement the `main()` function to sample the texture for the shadow receiver object and test for sufficient opacity for receiving shadow caster geometries.

```
void main(void) {
    vec4 texFragColor = texture2D(uSampler, vTexCoord);
    if (texFragColor.a < kSufficientlyOpaque)
        discard;
    else
        gl_FragColor = vec4(1, 1, 1, 1);
}
```

Interfacing the GLSL Shaders to the Engine

With two new GLSL shaders defined, you may expect that it is necessary to define two corresponding `Shader/Renderable` object pairs to facilitate the communications. This is not the case for two reasons.

- With the strategic variable naming in the `ShadowReceiverFS` shader, the existing `SpriteShader` object can be used to communicate with the `ShadowReceiverFS` GLSL fragment shader.

- Recall that the Renderable objects are designed to support instantiating and manipulating multiple game objects with the corresponding shaders. In this case, the ShadowCasterFS shader is meant for drawing shadow caster geometries, while the ShadowReceiverFS shader is meant for drawing the shadow receiver object into the stencil buffer. Notice that neither of the shaders is designed to support objects that are suitable for direct instantiation or manipulations. For these reasons, there is no need for the corresponding Renderable objects.

Creating the Shadow Caster Shader

A JavaScript Shader object 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 `ShadowCasterFS` fragment shader.

1. Under the `src/Engine/Shaders` folder, create a new file and name it `ShadowCasterShader.js`. Remember to load this new source file in `index.html`.
2. Define a constructor to inherit `ShadowCasterShader` from the `SpriteShader` object. Since each shadow caster geometry is created by one casting light source, define a single light source for the shader.

```
function ShadowCasterShader(vertexShaderPath, fragmentShaderPath) {
    // Call super class constructor
    SpriteShader.call(this, vertexShaderPath, fragmentShaderPath);

    this.mLight = null; // The light that casts the shadow

    // **** The GLSL Shader must define uLights[1] <-- as the only light source!!
    this.mShaderLight = new ShaderLightAtIndex(this.mCompiledShader, 0);
}
gEngine.Core.inheritPrototype(ShadowCasterShader, SpriteShader);
```

3. Override the `activateShader()` function to ensure the single light source is loaded to the shader.

```
// Overriding the Activation of the shader for rendering
ShadowCasterShader.prototype.activateShader = function (pixelColor, aCamera) {
    // first call the super class's activate
    SpriteShader.prototype.activateShader.call(this, pixelColor, aCamera);
    this.mShaderLight.loadToShader(aCamera, this.mLight);
};
```

4. Define a function to set the current light source for this shader.

```
ShadowCasterShader.prototype.setLight = function (l) {
    this.mLight = l;
};
```

Modifying the Engine Core

The core of the game engine and objects defined under the `src/Engine/Core` folder must be updated in two ways. First, the WebGL stencil buffer must be enabled and maintained. Second, default instances of the engine shaders must be defined to interface to the new GLSL shaders.

Configuring and Maintaining the WebGL Stencil and Depth Buffers

The WebGL stencil buffer must be allocated during WebGL system initialization and cleared when the canvas is cleared. With the well-designed and organized engine system, both of these operations should be defined in the `Engine_Core.js` file.

1. Edit the `Engine_Core.js` file. In the `_initializeWebGL()` 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.

```
var _initializeWebGL = function (htmlCanvasID) {
    var canvas = document.getElementById(htmlCanvasID);

    // Get the standard or experimental webgl and binds to the Canvas area
    // store the results to the instance variable mGL
    mGL = canvas.getContext("webgl", {alpha: false, depth: true, stencil: true}) ||
          canvas.getContext("experimental-webgl",
                            {alpha: false, depth: true, stencil: true});

    // 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);

    // make sure depth testing is enabled
    mGL.enable(mGL.DEPTH_TEST);
    mGL.depthFunc(mGL.LEQUAL);

    if (mGL === null) {
        document.write("<br><b>WebGL is not supported!</b>");
        return;
    }
};
```

2. Modify the `clearCanvas()` function. In addition to clearing the canvas, the stencil and depth buffers must also be cleared.

```
var clearCanvas = function (color) {
    mGL.clearColor(color[0], color[1], color[2], color[3]);      // set the color
    to be cleared
    mGL.clear(mGL.COLOR_BUFFER_BIT | mGL.STENCIL_BUFFER_BIT | mGL.DEPTH_BUFFER_BIT);
    // clear to the color, stencil bit, and depth buffer bits
};
```

Instantiating Default Shadow Caster and Receiver Shaders

Default instances of engine shaders must be created to connect to the newly defined GLSL shader caster and receiver fragment shaders.

1. Create constants and variables for the shaders in the `Engine_DefaultResources.js` file located in the `src/Engine/Core/Resources` folder.

```
// Shadow shaders
var kShadowReceiverFS = "src/GLSLShaders/ShadowReceiverFS.glsl";
// Path to the FragmentShader
var mShadowReceiverShader = null;
var kShadowCasterFS = "src/GLSLShaders/ShadowCasterFS.glsl";
// Path to the FragmentShader
var mShadowCasterShader = null;
```

2. Define engine shaders to interface to the new GLSL fragment shaders. Notice that both of the engine shaders are based on the `TextureVS` GLSL vertex shader. In addition, as discussed, the engine `SpriteShader` is created to interface to the `ShadowReceiverFS` GLSL fragment shader.

```
var _createShaders = function (callBackFunction) {
    gEngine.ResourceMap.setLoadCompleteCallback(null);
    // ...
    mShadowReceiverShader = new SpriteShader(kTextureVS, kShadowReceiverFS);
    mShadowCasterShader = new ShadowCasterShader(kTextureVS, kShadowCasterFS);
    callBackFunction();
};
```

3. The rest of the modifications to the `Engine_DefaultResources.js` file are routine, involving defining accessors, loading and unloading the GLSL source code files, cleaning up the shaders, and exporting the accessors via the public function list. 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.

Creating the Shadow Caster Object

As mentioned, creating a `Renderable` object to pair with the `ShadowCasterShader` object would allow game clients to create and manipulate shadow casters as game objects. However, shadow casters and the associated geometries are implicitly computed based on the associated shadow receiver and light sources. For this reason, shadow casters cannot be directly manipulated by the game clients.

Instead of the familiar Renderable object hierarchy, the `ShadowCaster` object is defined to encapsulate the implicitly defined shadow caster geometry functionality. A `ShadowCaster` object represents a Renderable game object that will cast shadow on a shadow receiver, another Renderable game object. To support receiving shadows on an animated sprite element, the shadow receiver must be at least a `SpriteRenderable` object. The shadow casting Renderable object must be able to receive light sources and thus is at least a `LightRenderable` object. The `ShadowCaster` object maintains references to the actual shadow casting and receiving Renderable objects and defines the algorithm to compute and render shadow caster geometries for each of the light sources referenced by the caster `LightRenderable` object. The details of the `ShadowCaster` object implementation are as follows:

1. Create the new `src/Engine/Shadows` folder for organizing shadow-related support files.
2. Create a new file in the `src/Engine/Shadows/` folder and name it `ShadowCaster.js`. Remember to load this new source file in `index.html`.
3. Define the constructor to initialize the instance variables and constants required for caster geometry computations.

```
function ShadowCaster (shadowCaster, shadowReceiver) {
    this.mShadowCaster = shadowCaster;
    this.mShadowReceiver = shadowReceiver;
    this.mCasterShader = gEngine.DefaultResources.getShadowCasterShader();
    this.mShadowColor = [0, 0, 0, 0.2];
    this.mSaveXform = new Transform();

    this.kCasterMaxScale = 3; // maximum size a caster will be scaled
    this.kVerySmall = 0.001; //
    this.kDistanceFudge = 0.01;
        // Ensure shadow caster is not at the same depth as receiver
    this.kReceiverDistanceFudge = 0.6;
        // Reduce the projection size increase of the caster
}
```

As discussed, the `mShdwCaster` is a reference to the shadow caster `GameObject`, which must reference at least a `LightRenderable` object, and the `mShadowReceiver` is a `GameObject` referencing at least a `SpriteRenderable` object. As will be detailed in the next step, `mCasterShader`, `mShadowColor`, and `mSaveXform` are variables to support the rendering of shadow caster geometries.

4. 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`.

```
ShadowCaster.prototype.draw = function(aCamera) {
    var casterRenderable = this.mShadowCaster.getRenderable();
    this.mShadowCaster.getXform().cloneTo(this.mSaveXform);
    var s = casterRenderable.swapShader(this.mCasterShader);
    var c = casterRenderable.getColor();
    casterRenderable.setColor(this.mShadowColor);
    var l, lgt;
    for (l = 0; l < casterRenderable.numLights(); l++) {
        lgt = casterRenderable.getLightAt(l);
```

```

        if (lgt.isLightOn() && lgt.isLightCastShadow()) {
            this.mSaveXform.cloneTo(this.mShadowCaster.getXform());
            if (this._computeShadowGeometry(lgt)) {
                this.mCasterShader.setLight(lgt);
                SpriteRenderable.prototype.draw.call(casterRenderable, aCamera);
            }
        }
    this.mSaveXform.cloneTo(this.mShadowCaster.getXform());
    casterRenderable.swapShader(s);
    casterRenderable.setColor(c);
};


```

`casterRenderable` is the `Renderable` object that is actually casting the shadow. The `draw()` function first saves the current transform, shader, and color of the `casterRenderable` object; iterates through all light sources, turning the `casterRenderable` into the shadow caster geometry; and renders it in three steps.

- Sets the `casterRenderable` shader to `ShadowCasterShader`.
- Calls the `_computeShadowGeometry()` function for each illuminating light source to project the `casterRenderable` onto the shadow receiver.
- Renders the `casterRenderable` as a `SpriteRenderable`. Recall that the `ShadowCasterShader` will sample the texture map, compute the strength of the current light source to scale the `mShadowColor`, and turn the pixel into the resulting color.

The `casterRenderable` state is restored before the `draw()` function returns.

- Define the `_computeShadowGeometry()` function to compute the shadow caster geometry based on the `mShadowCaster`, the `mShadowReceiver`, and a casting light source. Although slightly intimidating in length, the following function can be logically separated into four regions. The first region declares and initializes the variables. The second and third regions are the two cases of the `if` statement that handle the computation of transform parameters for directional and point/spotlights. The last and fourth region sets the computed parameters to the `cxf` transform.

```

ShadowCaster.prototype._computeShadowGeometry = function(aLight) {
    // Region 1: variable initialization
    var cxf = this.mShadowCaster.getXform();
    var rxf = this.mShadowReceiver.getXform();
    // vector from light to caster
    var lgtToCaster = vec3.create();
    var lgtToReceiverZ;
    var receiverToCasterZ;
    var distToCaster, distToReceiver; // measured along the lgtToCaster vector
    var scale;
    var offset = vec3.fromValues(0, 0, 0);
    receiverToCasterZ = rxf.getZPos() - cxf.getZPos();
}

```

```

if (aLight.getLightType() === Light.eLightType.eDirectionalLight) {
    // Region 2: parallel projection based on the directional light
    if (((Math.abs(aLight.getDirection()[2]) < this.kVerySmall) ||
        ((receiverToCasterZ * (aLight.getDirection()[2])) < 0)) {
        return false; // direction light casting side way or
                      // caster and receiver on different sides of light in Z
    }
    vec3.copy(lgtToCaster, aLight.getDirection());
    vec3.normalize(lgtToCaster, lgtToCaster);

    distToReceiver = Math.abs(receiverToCasterZ / lgtToCaster[2]);
                      // distant measured along lgtToCaster
    scale = Math.abs(1/lgtToCaster[2]);
} else {
    // Region 3: projection from a point (point light or spot light position)
    vec3.sub(lgtToCaster, cxf.get3DPosition(), aLight.getPosition());
    lgtToReceiverZ = rxf.getZPos() - (aLight.getPosition())[2];

    if ((lgtToReceiverZ * lgtToCaster[2]) < 0) {
        return false; // caster and receiver on different sides of light in Z
    }

    if ((Math.abs(lgtToReceiverZ) < this.kVerySmall) ||
        ((Math.abs(lgtToCaster[2]) < this.kVerySmall))) {
        // alomst the same Z, can't see shadow
        return false;
    }

    distToCaster = vec3.length(lgtToCaster);
    vec3.scale(lgtToCaster, lgtToCaster, 1/distToCaster);
                      // normalize lgtToCaster

    distToReceiver = Math.abs(receiverToCasterZ / lgtToCaster[2]);
                      // distant measured along lgtToCaster
    scale = (distToCaster + (distToReceiver * this.kReceiverDistanceFudge))
                      / distToCaster;
}
// Region 4: sets the cxf transform
vec3.scaleAndAdd(offset, cxf.get3DPosition(), lgtToCaster,
                  distToReceiver + this.kDistanceFudge);

cxf.setRotationInRad(cxf.getRotationInRad());
cxf.setPosition(offset[0], offset[1]);
cxf.setZPos(offset[2]);
cxf.setWidth(cxf.getWidth() * scale);
cxf.setHeight(cxf.getHeight() * scale);

return true;
};

```

The `aLight` parameter is the casting light source. The goals of this function is 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 8-28, there are two cases to consider: parallel projection for a directional light source or projection from a point for the point or spotlight.

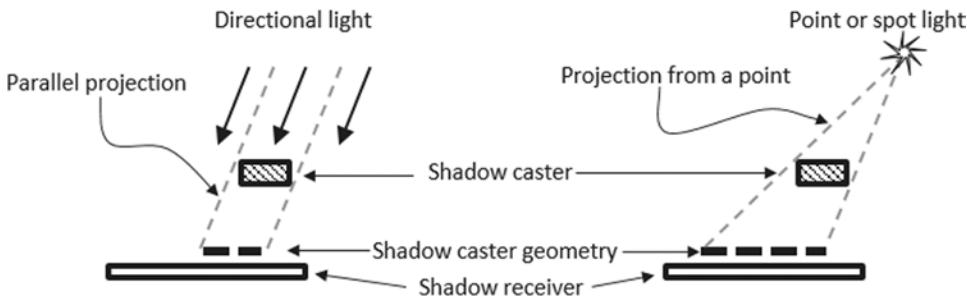


Figure 8-28. Computing the shadow caster geometry

- Region 2:* Computes parallel projection according to the directional light. The if statement 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.
- Region 3:* Computes projection from the point or spotlight position. The two if statements ensure the shadow caster and receiver are on the same side of the light position and that, for the purpose of maintaining mathematical stability, neither is close to the light source.
- Region 4:* Uses the computed `distToReceiver` and `scale` to set the `cxf` transform or the transform of the shadow caster.

Creating the Shadow Receiver Object

The `ShadowReceiver` object implements the outlined shadow simulation algorithm. The actual implementation of this object is separated into two files. The first file implements the core operations of the object, while the second defines the WebGL-specific stencil operations.

Defining the Shadow Receiver Operations

Follow these steps:

- Create a new file in the `src/Engine/Shadows/` folder and name it `ShadowReceiver.js`. Remember to load this new source file in `index.html`.
- Define the constructor to 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`.

```

function ShadowReceiver (theReceiverObject) {
    this.kShadowStencilBit = 0x01;           // The stencil bit to switch on/
    off for shadow
    this.kShadowStencilMask = 0xFF;          // The stencil mask
    this.mReceiverShader = gEngine.DefaultResources.getShadowReceiverShader();

    this.mReceiver = theReceiverObject;

    // To support shadow drawing
    this.mShadowCaster = [];                // array of ShadowCasters
}

```

3. Define the function `addShadowCaster()` to add a game object as a shadow caster for this receiver.

```

ShadowReceiver.prototype.addShadowCaster = function (lgtRenderable) {
    var c = new ShadowCaster(lgtRenderable, this.mReceiver);
    this.mShadowCaster.push(c);
};

```

4. Define the `draw()` function to draw the receiver and all the shadow caster geometries.

```

ShadowReceiver.prototype.draw = function (aCamera) {
    var c;

    // A: draw receiver as a regular renderable
    this.mReceiver.draw(aCamera);

    this._shadowRecieverStencilOn(); // B1
    var s = this.mReceiver.getRenderable().swapShader(this.mReceiverShader);
    this.mReceiver.draw(aCamera); // B2
    this.mReceiver.getRenderable().swapShader(s);
    this._shadowRecieverStencilOff(); // B3

    // C + D: now draw shadow color to the pixels in the stencil that are
    // switched on
    for (c = 0; c < this.mShadowCaster.length; c++)
        this.mShadowCaster[c].draw(aCamera);

    // switch off stencil checking
    this._shadowRecieverStencilDisable();
};

```

This function closely 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 B1 enables the stencil buffer 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, the calls to the `mShadowCaster draw()` function will draw shadow caster geometries only into the pixels that are covered by the receiver.

Defining the Shadow Receiver Stencil Operations

The stencil buffer configuration actually consists of WebGL-specific operations. These operations are gathered in this file for convenience.

1. Create a new file in the `src/Engine/Shadows/` folder and name it `ShadowReceiver_Stencil.js`. Remember to load this new source file in `index.html`.
2. Please refer to the source code file for the WebGL operations to configure the stencil buffer to implement the `_shadowRecieverStencilOn()`, `_shadowRecieverStencilOff()`, and `_shadowRecieverStencilDisable()` functions.

Updating Engine Supporting Objects

With the new objects defined and engine configured, some of the existing engine objects must also be modified to support the new shadow operations.

Modifying the Renderable

Both of the `ShadowCaster` and `ShadowReceiver` objects require the ability to swap the shaders and render the objects for shadow simulation purpose. This function is best realized in the root of the `Renderable` hierarchy. Edit the `Renderable.js` file and define the `swapShader()` function.

```
Renderable.prototype.swapShader = function (s) {
    var out = this.mShader;
    this.mShader = s;
    return out;
};
```

Modifying the SpriteShader

The engine interfaces to the GLSL `ShadowReceiverFS` using a `SpriteShader`, while the engine `ShadowReceiver` may be a reference to any of the `SpriteRenderable`, `LightRenderable`, and `IllumRenderable` objects. Edit the `SpriteShader.js` file to define the following two functions to ensure proper drawing for all `ShadowReceiver` objects:

```
// will be override by LightShader
SpriteShader.prototype.setLights = function (l) { };

// will be override by IllumShader
SpriteShader.prototype.setMaterialAndCameraPos = function(m, p) { };
```

Modifying the Light

The `Light` object should support the ability to switch shadow casting on or off. Edit the `Light.js` file to define the instance variable `mCastShadow` and accessor functions.

```
function Light() {
    // ... code identical to previous
    this.mCastShadow = false;
}
```

```
Light.prototype.isLightCastShadow = function () { return this.mCastShadow; };
Light.prototype.setLightCastShadowTo = function (on) { this.mCastShadow = on; };
```

Modifying the Camera

The Camera WC center must now be located at some z distance away. This is easily implemented by editing the `Camera.js` file and modifying the camera `lookAt()` matrix computation in the `setupViewProjection()` function.

```
mat4.lookAt(this.mViewMatrix,
    [center[0], center[1], this.kCameraZ], // WC center
    [center[0], center[1], 0],
    [0, 1, 0]); // orientation
```

Modifying the Transform Object

The last object that must be modified is the `Transform` utility. Recall that the `Transform` object is defined to implement the transformation operations in 2D. This object must now be updated to support some 3D positioning.

1. Edit the `Transform.js` file and add a z component.

```
function Transform() {
    this.mPosition = vec2.fromValues(0, 0); // this is the translation
    this.mScale = vec2.fromValues(1, 1); // this is the width (x) and height (y)
    this.mZ = 0.0; // must be a positive number,
                      // larger is closer to eye
    this.mRotationInRad = 0.0; // in radians!
}
```

2. Define assessors for the z position.

```
Transform.prototype.get3DPosition = function () {
    return vec3.fromValues(this.getXPos(), this.getYPos(), this.getZPos());
};
Transform.prototype.setZPos = function (d) { this.mZ = d; };
Transform.prototype.getZPos = function () { return this.mZ; };
Transform.prototype.incZPosBy = function (delta) { this.mZ += delta; };
```

3. Define the `cloneTo()` function to duplicate the transform.

```
Transform.prototype.cloneTo = function (aXform) {
    aXform.mPosition = vec2.clone(this.mPosition);
    aXform.mScale = vec2.clone(this.mScale);
    aXform.mZ = this.mZ;
    aXform.mRotationInRad = this.mRotationInRad;
};
```

4. Utilize the z component when computing an object transform.

```
Transform.prototype.getXform = function () {
    // Creates a blank identity matrix
    var matrix = mat4.create();

    // The matrices that WebGL uses are transposed, thus the typical matrix
    // operations must be in reverse.

    // Step A: compute translation, for now z is the mHeight
    mat4.translate(matrix, matrix, this.get3DPosition());
    // Step B: concatenate with rotation.
    mat4.rotateZ(matrix, matrix, this.getRotationInRad());
    // Step C: concatenate with scaling
    mat4.scale(matrix, matrix, vec3.fromValues(this.getWidth(),
        this.getHeight(), 1.0));

    return matrix;
};
```

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 `ShadowReceiver` objects and then add `ShadowCaster` objects to it. The `MyGame_Shadow.js` file defines the `_setupShadow()` function to demonstrate this. The `_setupShadow()` function is called at the end of the `MyGame initialize()` function, when all `GameObject` instances are properly created and initialized. The details of the `MyGame _setupShadow()` function are as follows:

```
MyGame.prototype._setupShadow = function () {
    this.mBgShadow = new ShadowReceiver(this.mBg);
    this.mBgShadow.addShadowCaster(this.mLgtHero);
    this.mBgShadow.addShadowCaster(this.mIllumMinion);
    this.mBgShadow.addShadowCaster(this.mLgtMinion);

    this.mMinionShadow = new ShadowReceiver(this.mIllumMinion);
    this.mMinionShadow.addShadowCaster(this.mIllumHero);
    this.mMinionShadow.addShadowCaster(this.mLgtHero);
    this.mMinionShadow.addShadowCaster(this.mLgtMinion);

    this.mLgtMinionShaodw = new ShadowReceiver(this.mLgtMinion);
    this.mLgtMinionShaodw.addShadowCaster(this.mIllumHero);
    this.mLgtMinionShaodw.addShadowCaster(this.mLgtHero);
};
```

This function demonstrates that three types of Renderable objects can serve as shadow receivers.

- `IllumRenderable`: `mBgShadow` has `mBg` as a receiver, which has a reference to an `IllumRenderable` object.
- `SpriteAnimateRenderable`: `mMinionShadow` has `mIllumMinion` as a receiver, which has a reference to a `SpriteAnimateRenderable` object.
- `LightRenderable`: `mLgtMinionShadow` has `mLgtMinon` as a receiver, which has a reference to a `LightRenderable` object.

The shadow casters for these receivers show that `IllumRenderable`, `SpriteAnimateRenderable`, and `LightRenderable` can all serve as shadow casters.

Drawing the Shadow

In 2D drawings, objects are drawn and overwrite 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 `drawCamera()` function is defined in the `MyGame.js` file:

```
MyGame.prototype.drawCamera = function (camera) {
    // set up the View Projection matrix
    camera.setupViewProjection();

    // always draw shadow receivers first!
    this.mBgShadow.draw(camera);           // also draws the receiver object
    this.mMinionShadow.draw(camera);
    this.mLgtMinionShaodw.draw(camera);

    this.mBlock1.draw(camera);
    this.mIllumHero.draw(camera);
    this.mBlock2.draw(camera);
    this.mLgtHero.draw(camera);

};
```

The rest of the `MyGame` level is largely similar to previous projects and is not listed here. Please refer to the source code for the details.

Observations

You can now run the project and observe the shadows. Notice the effect of the stencil buffer where the shadow from the `mIllumHero` object is cast on the minion and yet not on the background. Press the WASD keys to move both of the Hero objects. Observe how the shadows offer depth and distance cues as they move with the Hero objects. The `mLgtHero` on the right is illuminated by all four lights and thus casts many shadows. Light 1 does not illuminate the background, and thus the `mLgtHero` shadow from light 1 is not visible on the background but visible on the minions. Try selecting and manipulating each of the lights, such as moving or changing the direction or switching the light on/off to observe the effects on the shadows. You can even try changing the color of the shadow (in `ShadowCaster.js`) to something dramatic, such as to bright blue [0, 0, 5, 1], and observe shadows that could never exist in the real world.

Summary

This chapter guided you in developing a variation of the simple yet complete Phong illumination model for your game engine. The examples were organized to follow the three terms of the Phong illumination model: ambient, diffuse, and specular. The light source examples were strategically intermixed because without the lights illumination cannot occur.

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 a critical one because it enabled illumination computation based on simple physical models and simulation of an environment in 3D. The Phong illumination model and the need for a per-object material property were presented in the specular reflection example. The halfway vector variant of the Phong illumination model was implemented to avoid computing the light source reflection vector for each pixel. The light source types example 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 vital visual cues.

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 two chapters, this complementary pattern will be repeated. Chapter 9 will introduce physical behavior simulation, and Chapter 10 will complete the engine development with more advanced support for the camera including tiling and parallax.

Game Design Considerations

The work you did in the “Game Design Consideration” section of Chapter 7 to create a basic well-formed game mechanic will ultimately need to be paired with the other elements of game design to create something that feels satisfying for players. In addition to the basic mechanic or mechanics, you’ll need to think about your game’s systems, setting, and metagame and how they’ll help determine the kinds of levels you design, and you’ll want to begin exploring ideas for visual and audio design as you begin to define the setting.

As with most visual work, games rely in no small measure on lighting to convey setting. A horror game taking place in a graveyard at midnight will typically use a very different lighting model and color palette than a game focusing on upbeat, happy themes. Many people think that lighting applies primarily to games created in 3D engines that are capable of simulating realistic light and shadow, but the notion of lighting applies to most 2D game environments as well; consider the example presented by Playdead studio’s 2D side-scrolling platform game Limbo, as shown in Figure 8-29.



Figure 8-29. 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 programmatic 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 <http://www.playdead.com/limbo> for more information)

Lighting is also often used to help drive a game mechanic in addition to setting the mood; if you've ever played a game where you were navigating a game space in the dark with a virtual flashlight, that's one direct 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 areas with deadly gas, flashing lights on a map can help direct players to important areas, and the like.

In the Simple Global Ambient project, you saw the impact that colored environment lighting has on the game setting. In this project, the hero character moves in front of a background that appears to be made of metallic panels, tubes, and machinery; perhaps it's the exterior of a space ship. The environment light is red, and it can be pulsed. Notice the effect on mood when the intensity is set at a comparatively low 1.5 versus when it's set to something like a super-saturated 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 (especially representational environments of physical spaces) 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's return for a moment to the simple game mechanic project from the "Game Design Consideration" section of Chapter 7 and consider how you might think about adding lighting to the mechanic to make the puzzle more engaging; Figure 8-30 begins with the basic mechanic from the end of the exercise.

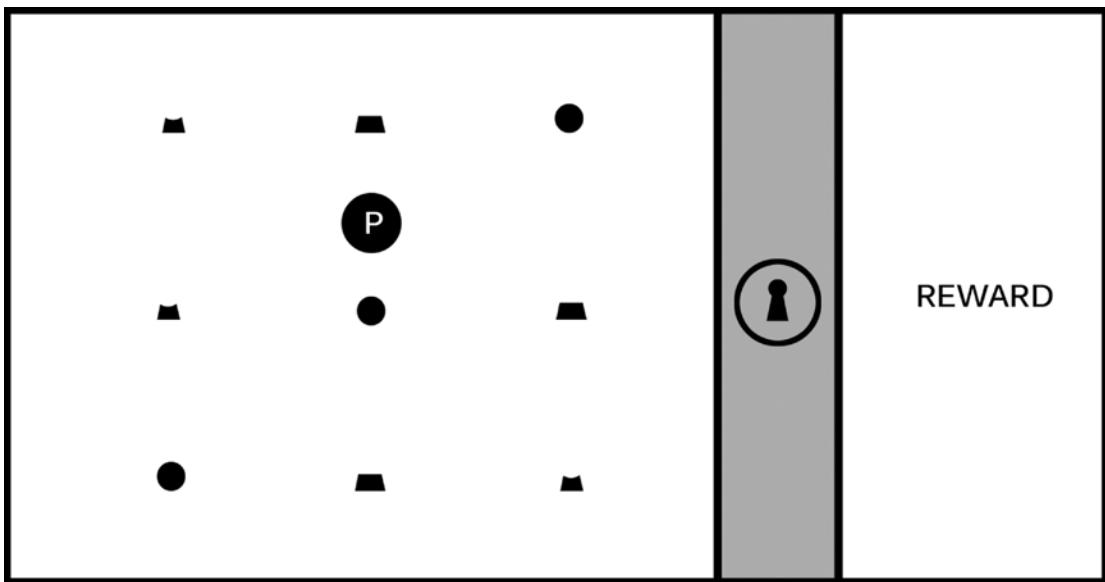


Figure 8-30. The simple game mechanic project, without lighting. Recall that the player controls the circle labeled with a P and must activate each of the three sections of the lock in proper sequence to disengage the barrier and reach the reward

Lighting can be used exclusively to support a game's visual style with no direct impact on gameplay, or as mentioned earlier, it can be integrated more or less directly with the game mechanic; depending on the direction your mechanic follows, either you'll want to begin experimenting with lighting right away (if it's integrated with the game mechanic) or you'll first focus on defining the game setting (if it's purely part of the visual style) because the setting will typically determine the colors, shapes, and lighting you choose.

For the next phase of the simple game mechanic project, how might you integrate light directly into the mechanic so that it becomes part of gameplay? As with the previous exercise, minimizing complexity and limiting yourself to one addition or evolution to the current mechanic at a time will help prevent the design from becoming over-burdened or too complex. Start this phase of the exercise by considering all the different ways that light might impact the current game screen. You might choose to have a dark environment where the player sees only shadowy shapes unless illuminating an area with a flashlight, you might use colored light to change the visible color of illuminated objects, or you might use something like an X-ray or ultraviolet beam to reveal information about the objects that wouldn't be seen with the naked eye. For this example, you'll add one additional dimension to the simple sequence mechanic: a light beam that reveals hidden information about the objects, as shown in Figure 8-31.

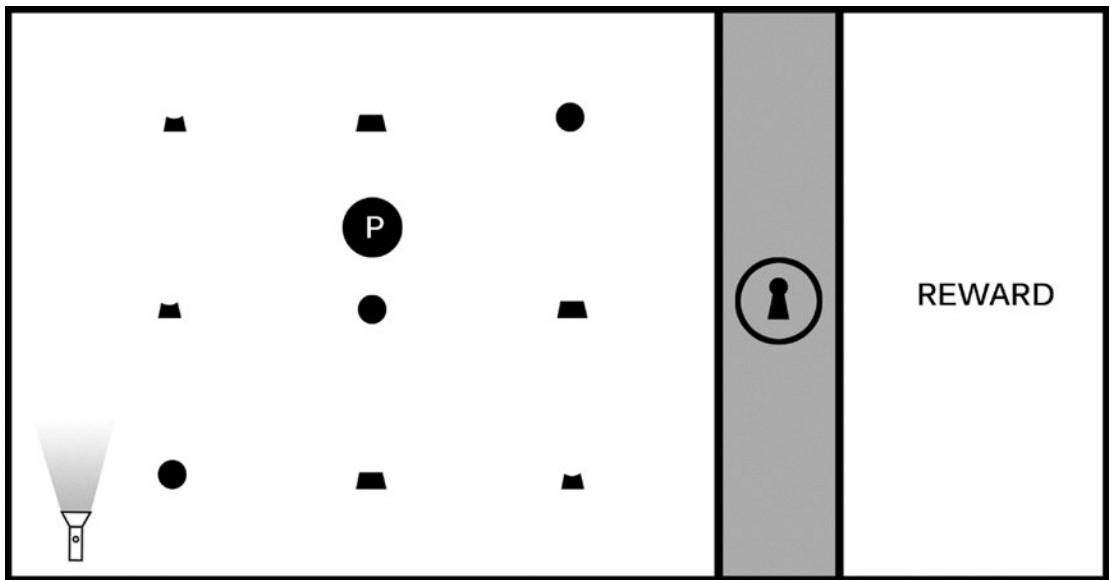


Figure 8-31. The addition of a movable “flashlight” that shines a special beam

In the first iteration of this mechanic, 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 could deduce the proper sequence required to unlock the barrier. Now imagine how the addition of a special light beam might take the mechanic in a new direction. Building upon the basic notion of sequencing, you can create an incrementally cleverer 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 8-32) 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 8-33 shows the flashlight moved to illuminate one of the objects with its beam, revealing a single white dot.

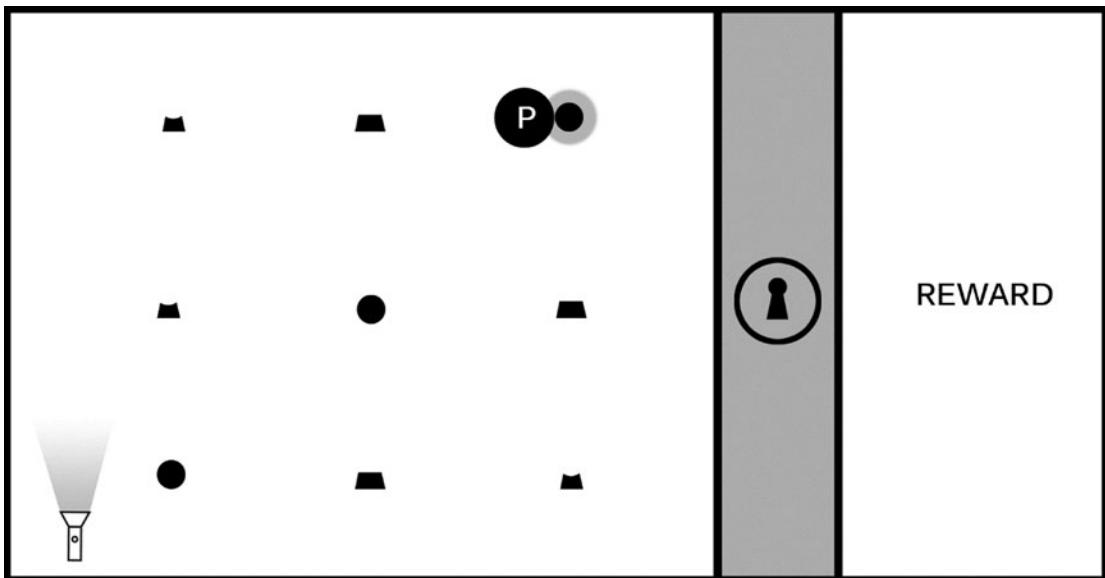


Figure 8-32. The player is able to directly activate the objects as in the first iteration of the mechanic, but the corresponding section of the lock now remains inactive

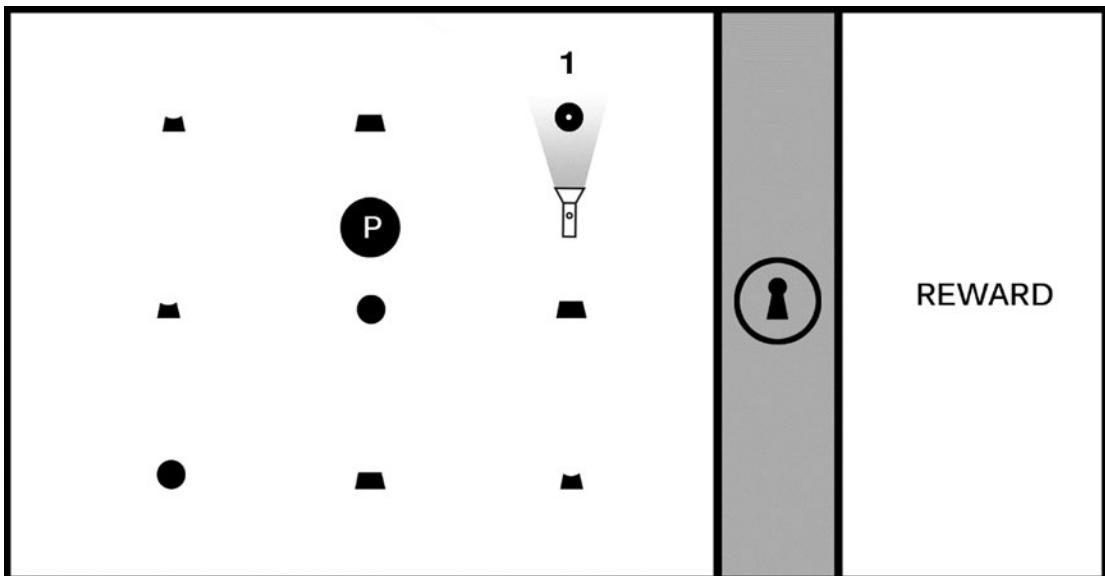


Figure 8-33. The player moves the flashlight under one of the shapes to reveal a hidden clue (#1)

From a gameplay point of view, any object in a game environment can be conscripted into service as a tool; your job as a mechanic designer is to ensure the use of 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 mechanic 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 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 8-34 to 8-36 show the new sequence using the flashlight beam.

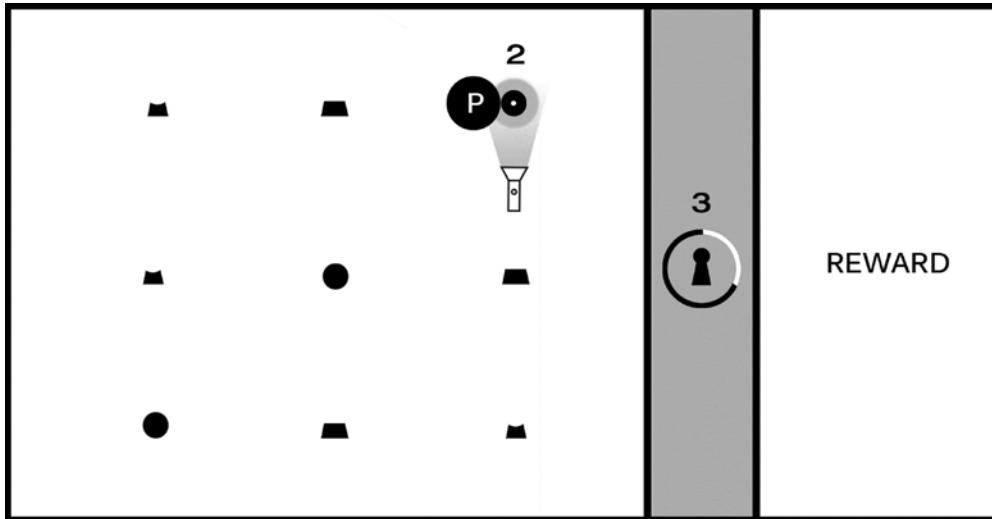


Figure 8-34. With the flashlight revealing the hidden symbol, the player can now activate the object (#2), and a progress bar (#3) on the lock indicates the player is on the right track to complete a sequence

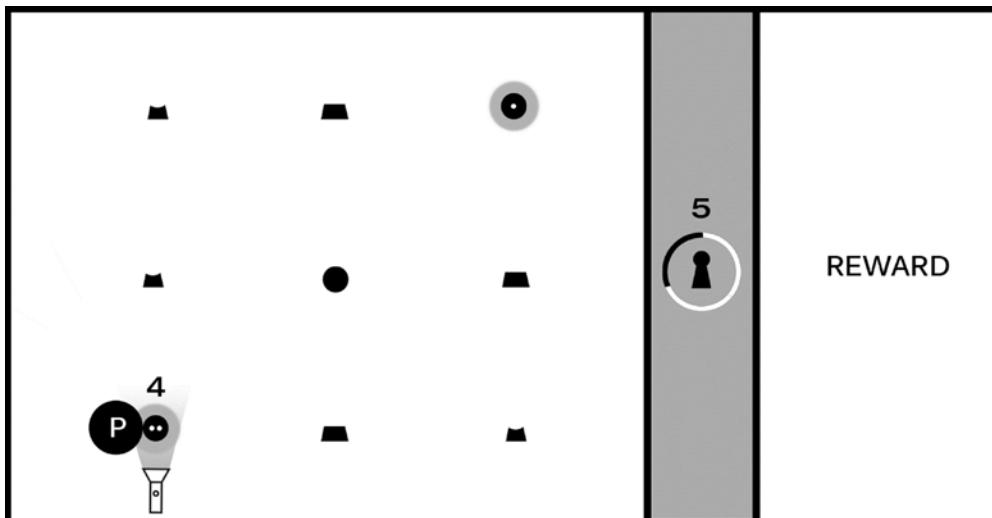


Figure 8-35. The player activates the second of the three top sections in the correct order (#4), and the progress bar confirms the correct sequence by lighting another section (#5). In this implementation, the player would not be able to activate the object with two dots before activating the object with one dot (the rules require activating like objects in order from one to three dots)

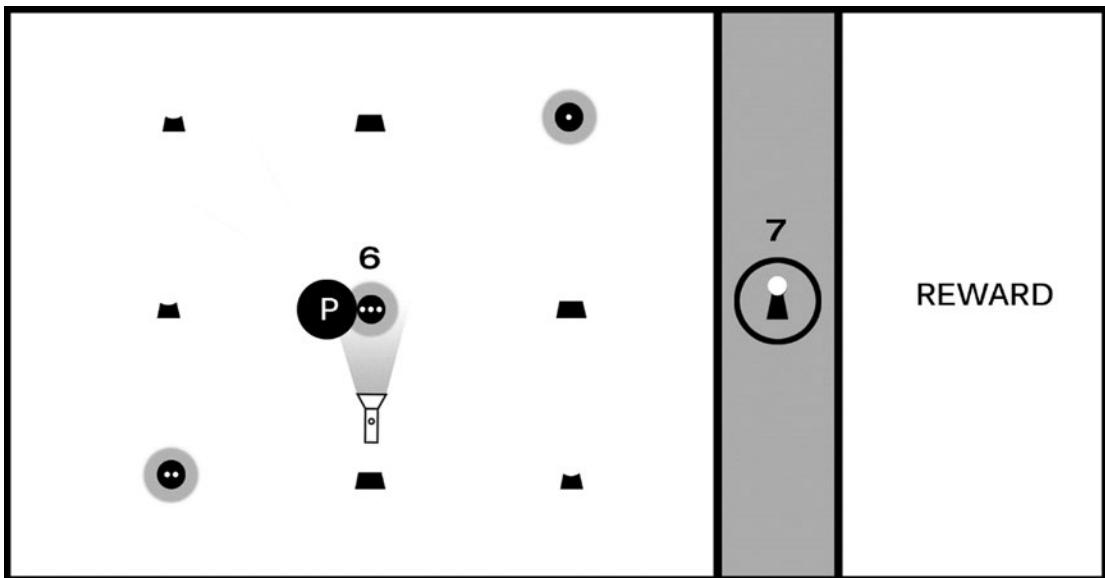


Figure 8-36. The third of the three top sections is revealed with the flashlight beam and activated by the player (#6), thereby activating the top section of the lock (#7). Once the middle and lower sections of the lock have been similarly activated, the barrier is disabled and players can claim the reward

Note that you've changed the feedback players receive slightly from the first iteration of the mechanic. 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 mechanic 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. If 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.

Although somewhat tangential to the focus of this chapter, It's important to be aware that as gameplay complexity increases, so increases the complexity of the interaction model and the importance of providing players with proper audiovisual feedback to help them make sense of their actions (recall from Chapter 1 that the interaction model is the combination of keys, buttons, controller sticks, touch gestures, and the like that the player uses to accomplish game tasks). In the current example, the player is now capable of controlling not just the hero character but also the flashlight. Creating intuitive interaction models is a critical component of game design and often much more complex than designers realize; as one example, consider the difficulty in porting many PC games designed for a mouse and keyboard to a game console using buttons and thumb sticks. Development teams often pour thousands of hours of research and testing into control schemes, yet they still frequently miss the mark. There are many books dedicated to examining interaction design in detail, but for the purposes of this book, you should keep the two golden rules in mind when you design interactions. First, use known and tested patterns when possible unless you have a compelling reason to ask players to learn something new; second, keep the number of unique actions players must remember to a minimum. Decades of user testing have clearly shown that players don't enjoy relearning basic key combinations for tasks that are similar across titles (which is why so many games have standardized on WASD for movement, for example), and similar data is available showing how easily players can become overwhelmed when you ask them to remember more than a few simple unique button

combinations. There are exceptions, of course; many classic arcade fighting games, for example, use dozens of complex combinations, but those genres are targeted to a specific kind of player who considers mastering button combinations to be a fundamental component of the game mechanic. As a general rule, most players prefer to keep interaction complexity as streamlined and simple as possible if it's not an intentional component of play.

There are a number of ways to deal with moving two objects in the current example. It would be reasonable to assign one kind of interaction for the hero character (the circle labeled *P*) and another for all other game objects (in this case, the flashlight). Perhaps the hero character can move around the game screen freely, while other objects are first selected with a left mouse click and can be moved by holding the left mouse button and dragging them into position. There are similarly a number of ways to provide the player with contextual feedback that will help teach the puzzle logic and rules (in this case, using 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 mechanic can evolve once again by applying simple physics to objects in the game world.

CHAPTER 9



Integrating Physics and Particles

After completing this chapter, you will be able to:

- Understand how to approximate integrals with Euler Method and Symplectic Euler Integration
- Approximate Newtonian motion formulation with Symplectic Euler Integration
- Understand the needs for and implement collision detections of bounding rectangles and circles
- Resolve interpenetrating collisions based on a numerically stable relaxation method
- Understand, implement, and work with particle systems

Introduction

In the previous chapter, you experienced building the illumination component by simulating the propagation of light energy in a game scene. Recall that only selected objects participated in the simulation. For instance, in a scene, only `IllumRenderable` objects can be illuminated by the light sources, while others such as `SpriteRenderable` objects are not. In a similar fashion, the physics component you will learn about in this chapter simulates the transfer of energy between selected objects. Just as illumination, 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 physics component with the same approach you used for all the previous game engine components you have implemented. That is, by focusing on the core functionality of the component, you increase the game engine's ability to create a diverse set of 2D games.

Note In games, the functionality of simulating energy transfer is often referred to as *physics*, *physics system*, or *physics component*. This convention is followed in the book.

Physics in 2D games can be seen in a wide variety of genres from simple Pong-like reaction-based games to more advanced Angry Birds-style block destruction games. Both of these games and everything in between (such as platformers) rely on specific physical behavior to implement their core gameplay. You will implement a physics system that can support a large variety of gameplay based on an architecture that supports expansion.

Particles are textured images displayed at positions with no defined dimensions that often participate in the physics simulation. The textured images facilitate representation of interesting effects while positions without dimensions allow simple and efficient physics simulations. A particle system controls and strategizes the generation of particles with specific appearance and behavior. A particle system provides a game engine with the ability to implement dynamic-looking reactions. This includes common effects such as explosions and effects associated with casting spells. Building upon the physics component, you will create a particle system that can support diverse gameplay requirements.

Physics Overview

Physics simulations in games involve three core concepts: movement, collision detection, and collision resolution. The proper implementation based on these concepts enables believable scenarios where objects physically interact with each other in the game world. To satisfy the real-time interaction requirement, the energy transfer simulations are approximated with optimization compromises.

Movement

Movement is the description of how object positions are maneuvered around the game world.

Mathematically, movement can be formulated in many ways. In Chapter 6, you experienced working with movement where you continuously accumulated a velocity to an object's position. Although desired results can be achieved, mathematically this is problematic because a velocity and a position are different types of quantities and strictly speaking cannot be combined. In practice, you have been working with describing movement based on displacements.

- $p_{new} = p_{current} + displacement$

Movements governed by the displacement formulation become restrictive when it is necessary to change the quantity being displaced over time. Newtonian mechanics address this restriction by considering time in movement formulations.

- $v_{new} = v_{current} + \int a(t) dt$

- $p_{new} = p_{current} + \int v(t) dt$

where $v(t)$ is the velocity that describes the change of position over time and $a(t)$ is the acceleration that describes the change of velocity over time.

Notice that both velocity and acceleration are vector quantities encoding the change in 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 via its magnitude and the direction the acceleration is occurring in. Acceleration is changed by the forces acting upon the object. For example, if you were to throw a ball into the air, the gravitational force on the earth would affect the object's acceleration over time, which in turn would change the object's velocity.

Euler method, or Explicit Euler Integration, approximates integrals based on initial values. Though potentially unstable, this is one of the simplest and thus a good beginning point to learn about integration approximation methods. In the case of the Newtonian movement formulation, the new velocity of the object can be approximated as the current velocity plus the current acceleration multiplied by the amount of elapsed time. Similarly, the object's new position can be approximated by the object's current position plus the current velocity multiplied by the amount of elapsed time.

Note An example of a numerically unstable system is one where under gravitational force a bouncing ball slows down but never stops jittering and, in some cases, may even start bouncing again.

- $v_{new} = v_{current} + a_{current} * dt$
- $p_{new} = p_{current} + v_{current} * dt$

In practice, because of system stability concerns, Explicit Euler Integration is seldom implemented. This shortcoming is overcome with the method you will be implementing, known as the Semi-Implicit Euler Integration or Symplectic Euler Integration, where intermediate results are used in subsequent approximations. The following equations show Symplectic Euler Integration. Notice that it is nearly identical to the Euler method except that the new velocity is being used when calculating the new position. This essentially means that the velocity for the next frame is being used to calculate the position of this frame.

- $v_{new} = v_{current} + a_{current} * dt$
- $p_{new} = p_{current} + v_{new} * dt$

Recall that the game loop you have created has a fixed time step, where the `update()` function is called 60 times per second consistently. In typical implementations, the Symplectic Euler Integration method is invoked once for each `update()` function call, and the dt quantity can simply be the fixed time step. This means that the integration approximation occurs once during each time interval. In this way, an object in continuous motion is approximated by a discrete set of positions. Figure 9-1 shows a moving object being approximated by the object at three different time steps. However, in a continuous motion, the approximation places the object in only one of these three positions. This is a consequence of the discrete time step nature of a game engine. The most notable ramifications of this approximation are in detecting collisions. You can see one such problem in Figure 9-1; imagine a thin wall existed in the space between the current and the next update. You would expect the object to collide and stop by the wall in the next update. However, if the wall were thin enough, the object would essentially pass right through it as it jumped from one position to the next. This is a common problem faced in many game engines. A general solution for these types of problems can be algorithmically complex and computationally intensive. It is typically the job of the game designer to mitigate and avoid this problem with well-designed (for example, appropriate size) and well-behaved (for example, appropriate traveling speed) game objects.

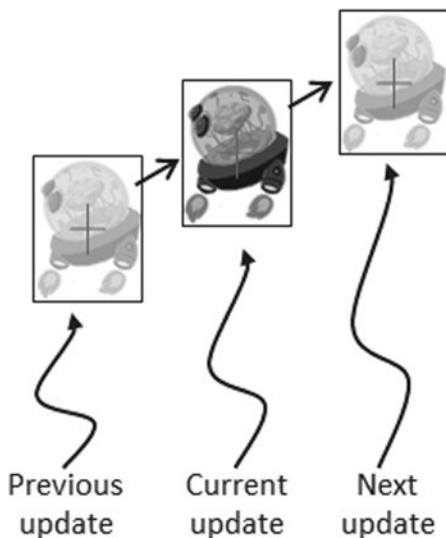


Figure 9-1. Approximating object position with Symplectic Euler Integration over three distinct time steps

Collision Detection

Collision detection is a vital and potentially a costly piece of physics simulation that can impact performance. For example, if you want to detect the collisions between five enemies, in the worst case you must perform four detection computations for the first enemy, followed by three computations for the second, two for the third, and one for the forth. In general, without dedicated optimizations, in the worst case you must perform $O(N^2)$ operations to detect the collisions between N objects.

Note Optimizations for collision detection typically exploit the proximity of game objects in the scene by organizing these objects either with a spatial structure such as uniform grid or quad-tree or into coherent groups such as hierarchies of bounding colliders. To focus on the core concepts and avoid unnecessary complications, these optimizations will not be covered.

You have experienced computing the collisions between objects from Chapters 6 with the `BoundingBox` utility class and the per-pixel collision detection algorithm. Although those methods can successfully detect the situation when objects overlap in space, they are not capable of computing the details of the collision. The details of a collision become important when it is necessary to resolve undesirable collision results.

Figure 9-2 shows two objects colliding after a time step. Before the time step, the objects are not touching. However, after the time step, the results of the numerical integration place the two objects over each other. This is another example ramification of the `update()` function being called at discrete time intervals. In the real world, given that the objects were solid, the two would never interpenetrate. This is where details of a collision must be computed such that the interpenetrating situation can be properly resolved.

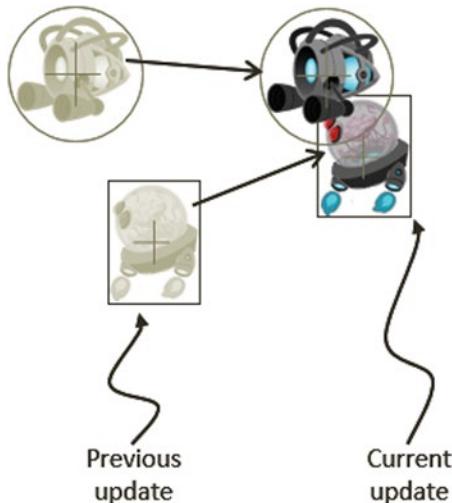


Figure 9-2. Interpenetration of objects after an update

Collision Resolution

In the context of game engines, collision resolution refers to the process that determines how objects respond after a collision, including strategies to resolve the potential interpenetration situation. Notice that there are no collision resolution processes in the real world where collisions are strictly governed by the law of physics and interpenetration of rigid objects would never occur. Resolutions of collisions are relevant only in the virtual game world, where movements are approximated and impossible conditions may occur but can be resolved in ways that are desirable to the designer.

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 the object's position so that it is no longer penetrating the other. While this is simple to calculate and implement, it lacks stability when many objects are in proximity and resting upon one another. The simple resolving of one pair of interpenetration can result in new penetrations with other close objects. However, this is still a common method for simple engines or games with simple object interaction rules. For example, in the Pong game, the ball never comes to rest on the paddles or walls and continuously remains in motion by bouncing off any object it collides with. The Projection Method is perfect for resolving collisions for these types of simple object interactions. The second method is known as the Impulse Method, which uses object velocities to compute and apply impulses to initiate the objects to move in the opposite directions at the point of collision. This method tends to slow down colliding objects rapidly and converges to stable solutions. This is because impulses are computed based on the transfer of momentum, which in turn has a damping effect on the velocities of the colliding objects. The third method is known as the Penalty Method, which models the depth of object interpenetration as the degree of compression of a spring and approximates an acceleration to apply forces to separate the objects. This last method is the most complex and challenging to implement.

For your engine, you will be combining the strengths of the Projection and Impulse Methods. The Projection Method will be used to separate the interpenetrating objects, while the Impulse Method will be used to apply small impulses to reduce the object velocities in the direction that caused the interpenetration. As described, the simple Projection Method can result in an unstable system, such as objects that sink into each other when stacked. You will overcome this instability by implementing relaxation where interpenetrated objects are separated incrementally via repeated applications of the Projection Method in a single update cycle. With relaxation, the number of times that the Projection Method is applied is referred to as *relaxation iterations*. For example, by default the engine sets relaxation iterations to 15. This means that within one `update()` function call, after the movement integration approximation, the collision detection and resolution procedures will be executed 15 times.

Note The Impulse Method was popularized and championed by Erin Catto with the popular Box2D physics game engine component. You can view the source code at <http://box2d.org/>.

Detecting Collisions

You have already worked with axis-aligned bounding boxes in Chapter 6. For flexibility and to better bound objects of different shapes, you will also support bounding circles in your game engine. The collision between game objects will be determined based on the collision results between bounding boxes and circles. You can position a bounding shape or shapes on your object regardless of the object's center position, and you can use multiple bounding shapes for detecting collisions with the object. Focusing on the core concepts and clarity of implementation, you will implement the straightforward system with one bounding shape centered on each game object. You can expand and improve this basic system after mastering the concepts involved.

Note This book covers the collision detection only between axis-aligned rectangles and circles. There are algorithms, such as the popular Separating Axis Theorem (SAT) and the Gilbert-Johnson-Keerthi (GJK), that are capable of detecting the collisions between any general convex shapes in 2D. These algorithms are computationally intensive and typically use simple bounding colliders such as bounding boxes or circles as a first pass.

The Rigid Shape Bounds Project

This project demonstrates how to detect collisions between bounding boxes and circles. For debugging purposes, the new bounding shapes can be drawn with the `RenderableLine` class. You can see an example of this project running in Figure 9-3. The source code to this project is defined in the Chapter9/9.1.RigidBodyBounds folder.



Figure 9-3. Running the Rigid Shape Bounds project

The controls of the project are as follows:

- **H/M/P keys:** Select one of Hero, Minion, or Platform. The selected object will not be drawn; instead, only its bounds will be drawn for better visual inspection of collision detection results.
- **Left mouse button press and move in the game window:** Drags the selected object (with only the bounding shape drawn) for collision detection. A detected collision causes the bounding shapes to be drawn in red.

The goals of the project are as follows:

- To derive the basic infrastructure to support collision detection between rigid shapes
- To understand and implement collision detection algorithms between bounding boxes and circles
- To lay the foundation for building a physics component

You can find the following external resources in the assets folder: the fonts folder that contains the default system fonts and two texture images (`minion_sprite.png`, which defines the sprite elements for the heroes and the minions, and `platform.png`, which defines the platforms).

Creating the Rigid Shape Base Class

You will begin by defining a base class for the bounding rectangle and circle. The base class will define all functionality that is common to the two bounding shapes. Note that a new class, `RigidRectangle`, is created for detecting collisions with bounding circles. Although similar, the existing functionality of the `BoundingBox` class is left unchanged to avoid affecting the rest of the engine that currently utilizes the `BoundingBox` class.

1. Start by creating a new subfolder called `Physics` under the `src/Engine` folder. In the `Physics` folder, create a new file called `RigidBody.js`.
2. Edit `RigidBody.js` and define the `eRigidType` enumerated data type to distinguish between the different types of rigid shapes.

```
RigidBody.eRigidType = Object.freeze({
    eRigidAbstract: 0,
    eRigidCircle: 1,
    eRigidRectangle: 2
});
```

3. Define the constructor for the rigid shape, which takes the transform and contains a `LineRenderable` object for drawing the center position of the rigid shape. The `mDrawBounds` variable defines whether the bounds should be drawn.

```
function RigidBody(xform) {
    this.mXform = xform; // this is typically from gameObject
    this.kPadding = 0.25 // size of the position mark
    this.mPositionMark = new LineRenderable();
    this.mDrawBounds = false;
}
```

4. Create a `draw()` function to draw an X at the center position of the rigid shape by utilizing the transform and the constant padding created in the constructor. Notice that the same `LineRenderable` is used for both pieces of the X to save memory. This may seem like an extraneous detail; however, when drawing many bounding boxes and circles, the savings can quickly add up.

```
RigidBody.prototype.draw = function (aCamera) {
    if (!this.mDrawBounds) {
        return;
    }
```

```

//calculation for the X at the center of the shape
var x = this.mXform.getXPos();
var y = this.mXform.getYPos();

this.mPositionMark.setFirstVertex(x - this.kPadding, y + this.kPadding);
                                         //TOP LEFT
this.mPositionMark.setSecondVertex(x + this.kPadding, y - this.kPadding);
                                         //BOTTOM RIGHT
this.mPositionMark.draw(aCamera);

this.mPositionMark.setFirstVertex(x + this.kPadding, y + this.kPadding);
                                         //TOP RIGHT
this.mPositionMark.setSecondVertex(x - this.kPadding, y - this.kPadding);
                                         //BOTTOM LEFT
this.mPositionMark.draw(aCamera);
};

```

5. Create an empty update() function for subclasses to override and add get and set accessors.

```

RigidShape.prototype.update = function () {};

RigidShape.prototype.rigidType = function () {
    return RigidShape.eRigidType.eRigidAbstract;
};

RigidShape.prototype.getPosition = function() {
    return this.mXform.getPosition();
};
RigidShape.prototype.setPosition = function(x, y ) {
    this.mXform.setPosition(x, y);
};
RigidShape.prototype.getXform = function () { return this.mXform; };
RigidShape.prototype.setXform = function (xform) { this.mXform = xform; };
RigidShape.prototype.setColor = function (color) {
    this.mPositionMark.setColor(color); };
RigidShape.prototype.getColor = function () {
    return this.mPositionMark.getColor(); };
RigidShape.prototype.setDrawBounds = function(d) { this.mDrawBounds = d; };
RigidShape.prototype.getDrawBounds = function() { return this.mDrawBounds; };

```

Creating the Rigid Rectangle Object

With the base abstract class for rigid shapes defined, you can now create your first rigid object, the rigid rectangle.

1. Under the Physics folder, create a new file called `RigidRectangle.js`.
2. Edit this file to create a constructor that initializes a width property and a height property and a `LineRenderable` object for drawing the sides of the rectangle. Make sure to inherit from the `RigidShape` base class.

```

function RigidRectangle(xform, w, h) {
    RigidShape.call(this, xform);
    this.mSides = new LineRenderable();
}

```

```

        this.mWidth = w;
        this.mHeight = h;
    }
gEngine.Core.inheritPrototype(RigidRectangle, RigidShape);

```

- Override the `draw()` function to draw the bounds of the rectangle when `mDrawBounds` is true. Remember to call the superclass `draw()` function to draw the center position of the shape.

```

RigidRectangle.prototype.draw = function (aCamera) {
    if (!this.mDrawBounds) {
        return;
    }
    RigidShape.prototype.draw.call(this, aCamera);
    var x = this.getPosition()[0];
    var y = this.getPosition()[1];
    var w = this.mWidth/2;
    var h = this.mHeight/2;

    this.mSides.setFirstVertex(x - w, y + h); //TOP LEFT
    this.mSides.setSecondVertex(x + w, y + h); //TOP RIGHT
    this.mSides.draw(aCamera);
    this.mSides.setFirstVertex(x + w, y - h); //BOTTOM RIGHT
    this.mSides.draw(aCamera);
    this.mSides.setSecondVertex(x - w, y - h); //BOTTOM LEFT
    this.mSides.draw(aCamera);
    this.mSides.setFirstVertex(x - w, y + h); //TOP LEFT
    this.mSides.draw(aCamera);
};

```

- Define the get and set accessors and override the `rigidType()` function to return the appropriate value.

```

RigidRectangle.prototype.rigidType = function () {
    return RigidShape.eRigidType.eRigidRectangle;
};
RigidRectangle.prototype.getWidth = function () { return this.mWidth; };
RigidRectangle.prototype.getHeight = function () { return this.mHeight; };
RigidRectangle.prototype.setColor = function (color) {
    RigidShape.prototype.setColor.call(this, color);
    this.mSides.setColor(color);
};

```

Creating the Rigid Circle Object

You can now implement the rigid circle object based on a similar overall structure.

- Under the Physics folder, create a new file called `RigidCircle.js`.
- Edit this file to create a constructor that initializes a `radius` property and create a `LineRenderable` for drawing the bounds of the circle. This class must also inherit from the `RigidShape` base class.

```

function RigidCircle(xform, r) {
    RigidBody.call(this, xform);
    this.kNumSides = 16;
    this.mSides = new LineRenderable();
    this.mRadius = r;
}
gEngine.Core.inheritPrototype(RigidCircle, RigidBody);

```

3. Override the `draw()` function to draw the circumference of the circle when `mDrawBounds` is true. Take note that the circumference of the circle is drawn as 16 individual line segments.

```

RigidCircle.prototype.draw = function (aCamera) {
    if (!this.mDrawBounds) {
        return;
    }
    RigidBody.prototype.draw.call(this, aCamera);

    // kNumSides forms the circle.
    var pos = this.getPosition();
    var prevPoint = vec2.clone(pos);
    var deltaTheta = (Math.PI * 2.0) / this.kNumSides;
    var theta = deltaTheta;
    prevPoint[0] += this.mRadius;
    var i, x, y;
    for (i = 1; i <= this.kNumSides; i++) {
        x = pos[0] + this.mRadius * Math.cos(theta);
        y = pos[1] + this.mRadius * Math.sin(theta);

        this.mSides.setFirstVertex(prevPoint[0], prevPoint[1]);
        this.mSides.setSecondVertex(x, y);
        this.mSides.draw(aCamera);

        theta = theta + deltaTheta;
        prevPoint[0] = x;
        prevPoint[1] = y;
    }
};

```

4. Define the get and set accessors and override the `rigidType()` function to return the appropriate value.

```

RigidCircle.prototype.rigidType = function () {
    return RigidBody.eRigidType.eRigidCircle;
};
RigidCircle.prototype.getRadius = function () {
    return this.mRadius;
};
RigidCircle.prototype.setColor = function (color) {
    RigidBody.prototype.setColor.call(this, color);
    this.mSides.setColor(color);
};

```

Detecting Collisions Between the Rigid Objects

Thus far you have implemented simple shapes that can be drawn on the canvas. You can now add collision detection behaviors to these shapes to determine whether they are colliding. You need to define functionality to detect collisions between two rectangles, between two circles, and between a rectangle and a circle. You will also add functions for determining whether a position is within the bounds of a shape.

Collision Detection Between a Rectangle and a Circle

Follow these steps:

1. In the Physics folder, create a new file called `RigidShape_Collision.js`. This file defines collision-related functions of the `RigidShape` class. The separate source code files ensure readability.
2. Create the `collidedRectCirc()` function to compute the collision between a rectangle and a circle. As illustrated in Figure 9-4, there are three distinct cases when a rectangle and a circle collide. The left and middle sketches of Figure 9-4 illustrate the first two cases where either the center of the rectangle is within the bounds of the circle or the center of circle is within the bounds of the rectangle. The third case on the right of Figure 9-4 illustrates two colliding shapes with centers outside the bounds of the other shape.

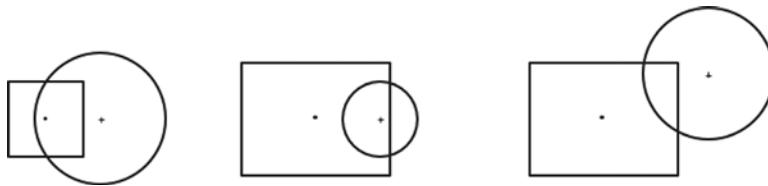


Figure 9-4. The three cases of rectangle and circle interpenetration

You can see this logic reflected in the following code where the first two simple cases are tested in the beginning of the function. For the third and more general case, Figure 9-5 uses two circles with centers at position V_{2a} and V_{2b} that interpenetrate the rectangle with a center at V_1 to illustrate two cases of this computation. Circle V_{2a} collides the rectangle from the top side, and V_{2b} collides from the right side. In both cases, the `vFrom1to2` vector connects the corresponding centers, and the size of the normal vector determines the collision condition: the collision is true when the length of the normal vector is less than the corresponding circle radius. The normal vector can be computed by subtracting `vec` from `vFrom1to2`. The `vec` vector is the result of clamping the corresponding component of the `vFrom1to2` vector by the collision side of the rectangle. For example, in the case of V_{2a} the y-component of the `vFrom1to2_a` vector is clamped by the y-value of the top side of the rectangle to form `vec_a`. Similarly, the x-component of the `vFrom1to2_b` vector is clamped by the x-value of the right side of the rectangle to form `vec_b`.

```
RigidShape.prototype.collidedRectCirc = function(rect1Shape, circ2Shape) {
    var rect1Pos = rect1Shape.getPosition();
    var circ2Pos = circ2Shape.getPosition();

    if (rect1Shape.containsPos(circ2Pos) || (circ2Shape.containsPos(rect1Pos))) {
        return true;
    }
```

```

var vFrom1to2 = [0, 0];
vec2.subtract(vFrom1to2, circ2Pos, rect1Pos);
var vec = vec2.clone(vFrom1to2);

var alongX = rect1Shape.getWidth() / 2;
var alongY = rect1Shape.getHeight() / 2;

vec[0] = this.clamp(vec[0], -alongX, alongX);
vec[1] = this.clamp(vec[1], -alongY, alongY);

var normal = [0, 0];
vec2.subtract(normal, vFrom1to2, vec);

var distSqr = vec2.squaredLength(normal);
var rSqr = circ2Shape.getRadius() * circ2Shape.getRadius();

return (distSqr < rSqr);
};

```

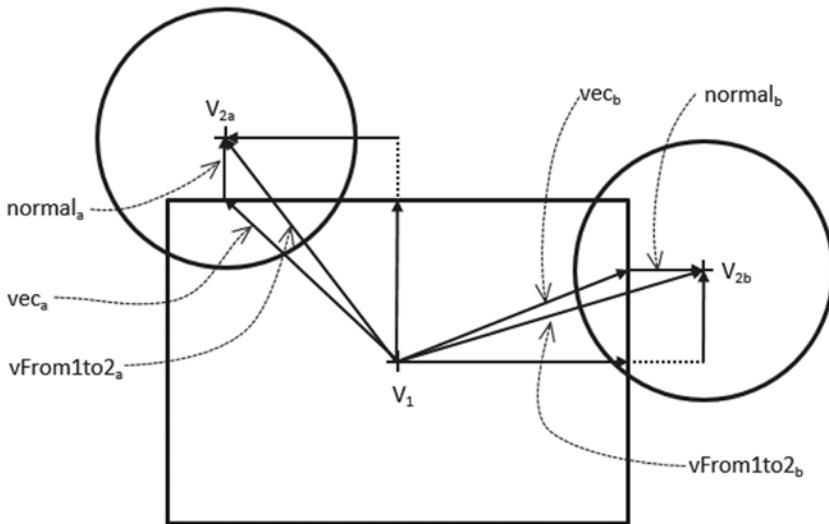


Figure 9-5. Two examples of general case of rectangle/circle collision

3. Define the `clamp()` function to clamp a value between minimum and maximum values.

```

RigidShape.prototype.clamp = function (value, min, max) {
    return Math.min(Math.max(value, min), max);
};

```

Collision Detection Between Two Rectangles

Follow these steps:

1. In the Physics folder, create a new file, `RigidRectangle_Collision.js`, to define the collision-related functions for the `RigidRectangle` object.
2. Create the `collidedRectRect()` function to receive two `RigidRectangle` objects and return whether the two have collided. Since the rectangles are axis-aligned, the collision condition can be determined by X and Y bounds check, similar to the `boundCollideStatus()` function of the `BoundingBox` class.

```
RigidRectangle.prototype.collidedRectRect = function(r1, r2) {
    var r1Pos = r1.getPosition();
    var r1MinX = r1Pos[0] - r1.getWidth() / 2;
    var r1MaxX = r1Pos[0] + r1.getWidth() / 2;
    var r1MinY = r1Pos[1] - r1.getHeight() / 2;
    var r1MaxY = r1Pos[1] + r1.getHeight() / 2;

    var r2Pos = r2.getPosition();
    var r2MinX = r2Pos[0] - r2.getWidth() / 2;
    var r2MaxX = r2Pos[0] + r2.getWidth() / 2;
    var r2MinY = r2Pos[1] - r2.getHeight() / 2;
    var r2MaxY = r2Pos[1] + r2.getHeight() / 2;

    return ((r1MaxX > r2MinX) && (r1MinX < r2MaxX) &&
            (r1MaxY > r2MinY) && (r1MinY < r2MaxY));
};
```

3. Define a `collided()` function for the `RigidRectangle` class to detect collisions between two arbitrary rigid shapes.

```
RigidRectangle.prototype.collided = function(otherShape) {
    var status = false;
    switch (otherShape.rigidType()) {
        case RigidShape.eRigidType.eRigidCircle:
            status = this.collidedRectCirc(this, otherShape);
            break;
        case RigidShape.eRigidType.eRigidRectangle:
            status = this.collidedRectRect(otherShape, this);
            break;
    }
    return status;
};
```

4. Create the `containsPos()` function to determine whether a given position is within the bounds of the rectangle.

```
RigidRectangle.prototype.containsPos = function (pos) {
    var rPos = this.getPosition();
    var rMinX = rPos[0] - this.getWidth() / 2;
    var rMaxX = rPos[0] + this.getWidth() / 2;
```

```

var rMinY = rPos[1] - this.getHeight() / 2;
var rMaxY = rPos[1] + this.getHeight() / 2;

return ((rMinX < pos[0]) && (rMaxX > pos[0]) &&
        (rMinY < pos[1] && rMaxY > pos[1]));
};

```

Collision Detection Between Two Circles

Follow these steps:

1. In the Physics folder, create a new file, `RigidCircle_Collision.js`, to define the collision-related functions for the `RigidCircle` object.
2. Create the `collidedCircCirc()` function to receive two `RigidCircle` objects and return whether the two have collided. The collision condition is true when the distance between the two circle centers is less than the sum of their radii. To avoid the expensive square root function when computing a vector magnitude, the squared values are compared instead.

```

RigidCircle.prototype.collidedCircCirc = function(c1, c2) {
    var vecToCenter = [0, 0];
    vec2.sub(vecToCenter, c1.getPosition(), c2.getPosition());
    var rSum = c1.getRadius() + c2.getRadius();
    return (vec2.squaredLength(vecToCenter) < (rSum * rSum));
};

```

3. As in the case for `RigidBody`, define a `collide()` function to support collisions with an arbitrary rigid shape.

```

RigidCircle.prototype.collided = function(otherShape) {
    var status = false;
    switch (otherShape.rigidType()) {
        case RigidShape.eRigidType.eRigidCircle:
            status = this.collidedCircCirc(this, otherShape);
            break;
        case RigidShape.eRigidType.eRigidRectangle:
            status = this.collidedRectCirc(otherShape, this);
            break;
    }
    return status;
};

```

4. Create the `containsPos()` function to determine whether a given position is within the bounds of the circle.

```

RigidCircle.prototype.containsPos = function(pos) {
    var dist = vec2.distance(this.getPosition(), pos);
    return (dist < this.getRadius());
};

```

Modifying the Game Object

You can now modify the `GameObject` class to begin building the support for the eventual physics component of the game engine.

1. Edit `GameObject.js` and declare a new variable: `mPhysicsComponent` in the constructor.

```
function GameObject(renderableObj) {
    this.mRenderComponent = renderableObj;
    this.mVisible = true;
    this.mCurrentFrontDir = vec2.fromValues(0, 1); // the current front direction
    this.mSpeed = 0;
    this.mPhysicsComponent = null;
}
```

2. Define simple get and set accessors for the physics component.

```
GameObject.prototype.setPhysicsComponent = function (p)
    { this.mPhysicsComponent = p; };
GameObject.prototype.getPhysicsComponent = function ()
    { return this.mPhysicsComponent; };
```

3. Modify the `update()` and `draw()` functions to support the physics component if it exists.

```
GameObject.prototype.update = function () {
    // simple default behavior
    var pos = this.getXform().getPosition();
    vec2.scaleAndAdd(pos, pos, this.getCurrentFrontDir(), this.getSpeed());

    if (this.mPhysicsComponent !== null) {
        this.mPhysicsComponent.update();
    }
};

GameObject.prototype.draw = function (aCamera) {
    if (this.isVisible()) {
        this.mRenderComponent.draw(aCamera);
    }
    if (this.mPhysicsComponent !== null) {
        this.mPhysicsComponent.draw(aCamera);
    }
};
```

Testing the Collision Detection

It is important to create objects with both rectangle and circle bounds to verify the correctness of collision detection functionality.

Modifying the Game Objects

Modify the `Hero` and `Minion` objects to define rigid objects as their physics component.

1. Modify the `Hero.js` file to define a `RigidCircle` to be its physics component.

```
function Hero(spriteTexture, atX, atY) {
    // ... identical code to previous project

    var r = new RigidCircle(this.getXform(), 9);
    r.setColor([0, 1, 0, 1]);
    r.setDrawBounds(true);
    this.setPhysicsComponent(r);
}
```

2. Edit the `Minion.js` file to define either a `RigidCircle` or a `RigidRectangle` to be its physics component.

```
function Minion(spriteTexture, atX, atY) {
    // ... identical code to previous project

    var r;
    if (Math.random() > 0.5) {
        r = new RigidCircle(this.getXform(), 7);
    } else {
        r = new RigidRectangle(this.getXform(), 17, 14);
    }
    r.setColor([0, 1, 0, 1]);
    r.setDrawBounds(true);
    this.setPhysicsComponent(r);
}
```

3. Create a new `GameObject` type, `Platform`, with a `RigidRectangle` defined as its physics component. Because of the similarity with the previous code, the details are not shown here.

Modifying the MyGame Object

To properly test and observe the results of your collision detection implementation, begin with a new `MyGame.js` file and implement the usual necessary functions of the constructor: `initialize()`, `update()`, and `draw()`. Please refer to the `MyGame.js` file for the details of the implementation. The interesting elements of this example are in the `_detectCollision()` function. This function is called by the `update()` function to compute the collision between the selected object against all other objects in the scene. The colliding objects will have their corresponding `RigidBody` objects drawn in red.

```
MyGame.prototype._detectCollision = function () {
    var i, obj;
    this.mCollidedObj = null;
    var selectedRigidBody = this.mSelectedObj.getPhysicsComponent();
```

```

for (i = 0; i<this.mAllObjects.size(); i++) {
    obj = this.mAllObjects.getObjectAt(i);
    if (obj != this.mSelectedObj) {
        if (selectedRigidShape.collided(obj.getPhysicsComponent())) {
            this.mCollidedObj = obj;
            this.mCollidedObj.getPhysicsComponent().setColor(this.kCollideColor);
        } else {
            obj.getPhysicsComponent().setColor(this.kNormalColor);
        }
    }
}
};


```

Observations

Run the project to test your implementation. Notice that the texture for the selected object is not drawn. This is to ensure that you can clearly observe and verify that the edges of colliding bounds are touching when the bound color changes to red. Cycle through the H, M, and P keys to observe the same three objects of the many instances are always selected. Now you can move the mouse with a left button click to move the bounds of the selected object. Notice that you can simultaneously collide with the bounds of multiple objects. Lastly, take note that the bounding shapes may not completely enclose the `GameObject` textures, as in the case of the Hero object where part of Dye's head and leg are outside of the bounding circle. How tightly a bounding shape should enclose an object should be determined by the gameplay requirement and not by the geometries.

Resolving Collisions

With a functioning collision detection system, you can now begin implementing collision resolution and support more natural and realistic behaviors. Focusing on the core concepts and building a stable system, you will continue to work with axis-aligned rigid rectangles and circles and avoid the complications involved in computing angular impulse resolutions and detecting collisions between rotated rectangles. For this reason, in your implementation, objects will not rotate as a response to collisions. However, the concepts and implementation approach described in this section can be generalized to support object rotation collision responses.

Note Interested readers can refer to the following links on how rotation for impulse resolution can be supported: <https://research.ncl.ac.uk/game/mastersdegree/gametechnologies/> or <http://gamedevelopment.tutsplus.com/series/how-to-create-a-custom-physics-engine--gamedev-12715>.

The Rigid Shape Impulse Project

This project demonstrates how to combine and implement projection and impulse-based collision resolution for collided rigid rectangles and circles. Additionally, this project implements the Symplectic Euler Integration for object movements. You can see an example of this project running in Figure 9-6. The source code of this project is located in the Chapter9/9.2.RigidShapeImpulse folder.

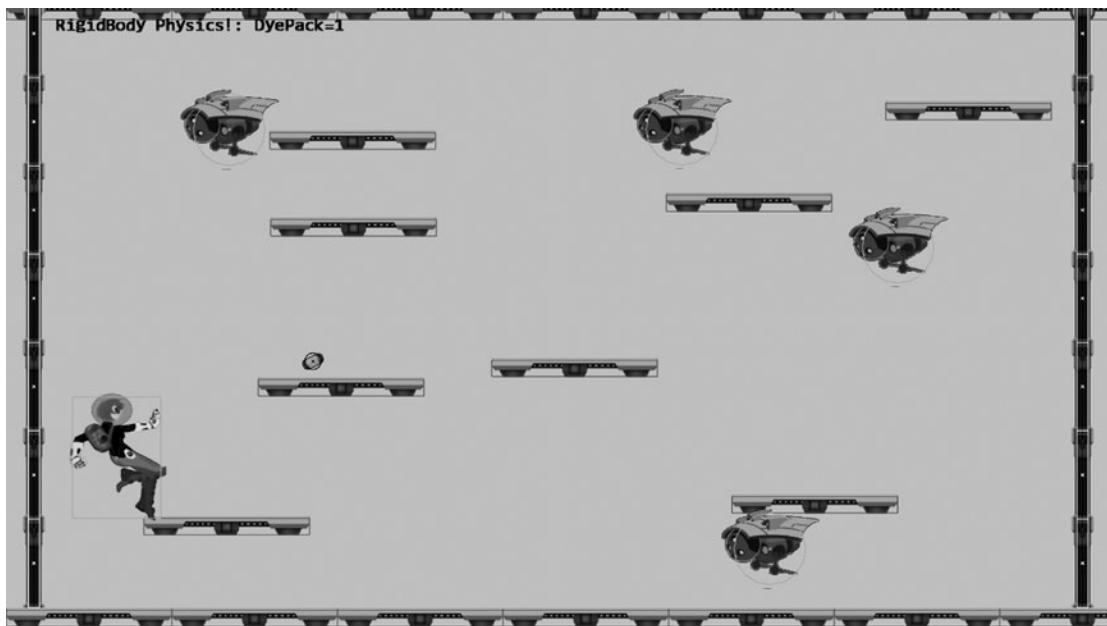


Figure 9-6. Running the Rigid Shape Impulse project

The controls of the project are as follows:

- *WASD keys:* Moves the Hero object
- *Left mouse button click in the game window:* Sends a DyePack chasing after the Hero
- *For testing the stability of collision resolution:*
 - *Z key:* Creates dumb minions with white bounds under the current mouse pointer position
 - *X key:* Selects the dumb minion under the current mouse pointer position; moving the mouse with the X key pressed will move the selected dumb minion
 - *C key:* Agitates all dumb minions with a random velocity; press and hold the C key to increase the agitation
 - *V key:* Clears all dumb minions

The goals of the project are as follows:

- To experience implementing movements based on Symplectic Euler Integration
- To understand the need for and experience the implementation of collecting collision information
- To build a physics component with relaxation support

You can find the following external resources in the assets folder: the fonts folder that contains the default system fonts and four texture images (`minion_sprite.png`, which defines the sprite elements for the hero and the minions; `platform.png`, which defines the platforms and floor and ceiling tiles; `wall.png`, which defines the walls; and `dye_pack.png`, which defines the dye packs).

Implementing Symplectic Euler Integration

For objects to collide and the resulting collisions resolved, their movements must be properly simulated. Recall the discussion from the beginning of this chapter that such movements can be described by the Newtonian motion equations, and the solutions can be approximated by the Symplectic Euler Integration.

- $v_{new} = v_{current} + a_{current} * dt$
- $p_{new} = p_{current} + v_{new} * dt$

The fixed time step update() function architecture of the game engine allows the *dt* quantity to be implemented as the update time interval and the integral to be evaluated once per update cycle. You will modify the RigidBody class for this implementation.

Modifying the RigidShape Class

Modify the RigidShape class to include variables for mass, restitution (bounciness), velocity, friction, and acceleration, as shown in the following code. Notice that the inverse of the mass value is actually stored for computation efficiency.

```
function RigidShape(xform) {
    this.mXform = xform; // this is typically from gameObject
    this.kPadding = 0.25; // size of the position mark
    this.mPositionMark = new LineRenderable();
    this.mDrawBounds = false;

    // physical properties
    this.mInvMass = 1;
    this.mRestitution = 0.8;
    this.mVelocity = vec2.fromValues(0, 0);
    this.mFriction = 0.3;
    this.mAcceleration = gEngine.Physics.getSystemAcceleration();
}
```

Creating the Rigid Shape Behavior

You can now add the behavior to the rigid shape object for numerical integration.

1. Under the src/Engine/Physics folder, add a new file and name it `RigidShapeBehavior.js`. Remember to load this new source file in `index.html`.
2. Define an `update()` function to apply Symplectic Euler Integration to the rigid shape where the updated velocity is used for computing the new position. Notice that the use of the inverse mass in computing the velocity is a deviation from classical physics. This is a convenient trick that affords you the ability to switch off an object's motion by initializing both the velocity and inverse mass to zero.

```
RigidBody.prototype.update = function () {
    var dt = gEngine.GameLoop.getUpdateIntervalInSeconds();

    // Symplectic Euler
    //   v += (1/m * a) * dt
    //   x += v * dt
    var v = this.getVelocity();
    vec2.scaleAndAdd(v, v, this.mAcceleration, (this.getInvMass() * dt));

    var pos = this.getPosition();
    vec2.scaleAndAdd(pos, pos, v, dt);
};
```

3. Create get and set accessors for each of the newly added variables. Note that the `setMass()` function actually computes the inverse value or initializes `mInvMass` to zero.

```
RigidBody.prototype.getInvMass = function () { return this.mInvMass; };
RigidBody.prototype.setMass = function (m) {
    if(m > 0) {
        this.mInvMass = 1/m;
    } else {
        this.mInvMass = 0;
    }
};
RigidBody.prototype.getVelocity = function () { return this.mVelocity; };
RigidBody.prototype.setVelocity = function (v) { this.mVelocity = v; };
RigidBody.prototype.getRestitution = function () { return this.mRestitution; };
RigidBody.prototype.setRestitution = function (r) { this.mRestitution = r; };
RigidBody.prototype.getFriction = function () { return this.mFriction; };
RigidBody.prototype.setFriction = function (f) { this.mFriction = f; };
RigidBody.prototype.getAcceleration = function () { return this.mAcceleration; };
RigidBody.prototype.setAcceleration = function (g) { this.mAcceleration = g; };
```

Extracting Collision Information

To support proper resolution, at each collision you need to compute a collision depth and the corresponding collision normal. The collision depth is the smallest amount that the objects interpenetrated where the collision normal is the direction along which the collision depth is measured. For example, in Figure 9-5 the distance defined by subtracting the length of the normal vector from the circle radius is the collision depth, and the normal vector is the collision normal because the collision depth is measured along this direction. It is always the case that any interpenetration can be resolved by moving the objects along the collision normal by collision depth distance. The `RigidBody` collision functions must be modified to compute for this information.

Creating the CollisionInfo object

For convenience, create a new object for recording the collision information.

- Under the `src/Engine/Utils` folder, add a new file and name it `CollisionInfo.js`. Remember to load this new source file in `index.html`.
- Define the constructor and the accessors for the collision depth and normal.

```
function CollisionInfo() {
    this.mDepth = 0;
    this.mNormal = vec2.fromValues(0, 0);
}
CollisionInfo.prototype.setDepth = function (s) { this.mDepth = s; };
CollisionInfo.prototype.setNormal = function (s) { this.mNormal = s; };

CollisionInfo.prototype.getDepth = function () { return this.mDepth; };
CollisionInfo.prototype.getNormal = function () { return this.mNormal; };
```

Modifying the RigidBody Collision

You can now edit the `RigidBody_Collision.js` file to extract the collision information when a rectangle and circle collide. Figure 9-5 illustrates the general strategy of computing the collision normal, normal, and collision depth (subtracting the length of `normal` from the radius). As discussed previously, the `normal` vector is derived from `vec`, the result of clamping the components of the `vFrom1to2` vector by the colliding side of the rectangle. As illustrated in Figure 9-7, the only slight complication is when the circle center is inside the bounds of the rectangle, where instead of clamping, you must extend corresponding `vFrom1to2` components to compute the `vec` vector. For example, in Figure 9-7, the x-component of `vFrom1to2` must be extended to reach the x-value of the rectangle right boundary to form the `vec` vector. Notice that in all cases, the computed `normal` is a vector from the collision rectangle bound toward the circle center. For this reason, in the case of Figure 9-7, the `normal` vector must be reversed to represent collision normal.

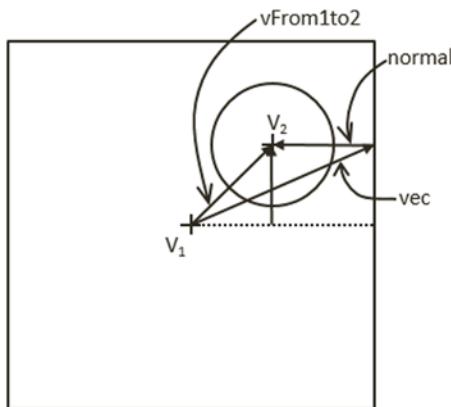


Figure 9-7. Circle center in the bounds of the colliding rectangle

```

RigidBody.prototype.collidedRectCirc = function(rect1Shape, circ2Shape, collisionInfo) {
    var rect1Pos = rect1Shape.getXform().getPosition();
    var circ2Pos = circ2Shape.getXform().getPosition();
    var vFrom1to2 = [0, 0];
    vec2.subtract(vFrom1to2, circ2Pos, rect1Pos);

    var vec = vec2.clone(vFrom1to2);

    var alongX = rect1Shape.getWidth() / 2;
    var alongY = rect1Shape.getHeight() / 2;

    vec[0] = this.clamp(vec[0], -alongX, alongX);
    vec[1] = this.clamp(vec[1], -alongY, alongY);

    var isInside = false;
    if (rect1Shape.containsPos(circ2Pos)) {
        isInside = true;
        // Find closest axis
        if (Math.abs(vFrom1to2[0] - alongX) < Math.abs(vFrom1to2[1] - alongY)) {
            // Clamp to closest side
            if (vec[0] > 0) {
                vec[0] = alongX;
            } else {
                vec[0] = -alongX;
            }
        } else { // y axis is shorter
            // Clamp to closest side
            if (vec[1] > 0) {
                vec[1] = alongY;
            } else {
                vec[1] = -alongY;
            }
        }
    }

    var normal = [0, 0];
    vec2.subtract(normal, vFrom1to2, vec);

    var distSqr = vec2.squaredLength(normal);
    var rSqr = circ2Shape.getRadius() * circ2Shape.getRadius();

    if (distSqr > rSqr && !isInside) {
        return false; //no collision exit before costly square root
    }

    var len = Math.sqrt(distSqr);
    var depth;
}

```

```

    vec2.scale(normal, normal, 1/len); // normalize normal
    if (isInside) { //flip normal if inside the rect
        vec2.scale(normal, normal, -1);
        depth = circ2Shape.getRadius() + len;
    } else {
        depth = circ2Shape.getRadius() - len;
    }

    collisionInfo.setNormal(normal);
    collisionInfo.setDepth(depth);
    return true;
};

}

```

Modifying the RigidRectangle Collision

Edit the `RigidRectangle_Collision.js` file to compute for collision information when a collision occurs.

1. Modify the `collidedRectRect()` function to calculate the collision depth and normal. In the following code, the collision normal is chosen to be along either the x- or y-direction with the corresponding depth.

```

RigidRectangle.prototype.collidedRectRect = function(r1, r2, collisionInfo) {
    var vFrom1to2 = vec2.fromValues(0, 0);
    vec2.sub(vFrom1to2, r2.getPosition(), r1.getPosition());
    var xDepth = (r1.getWidth() / 2) + (r2.getWidth() / 2) - Math.abs(vFrom1to2[0]);
    if (xDepth > 0) {
        var yDepth = (r1.getHeight() / 2) + (r2.getHeight() / 2) -
                    Math.abs(vFrom1to2[1]);
        if (yDepth > 0) {
            //axis of least penetration
            if (xDepth < yDepth) {
                if (vFrom1to2[0] < 0) {
                    collisionInfo.setNormal([-1, 0]);
                } else {
                    collisionInfo.setNormal([1, 0]);
                }
                collisionInfo.setDepth(xDepth);
            } else {
                if (vFrom1to2[1] < 0) {
                    collisionInfo.setNormal([0, -1]);
                } else {
                    collisionInfo.setNormal([0, 1]);
                }
                collisionInfo.setDepth(yDepth);
            }
            return true;
        }
    }
    return false;
};

```

2. Modify the `collided()` function to include the `collisionInfo` parameter. Take note that the default collision depth value is initialized to zero.

```
RigidRectangle.prototype.collided = function(otherShape, collisionInfo) {
    var status = false;
    collisionInfo.setDepth(0);
    switch (otherShape.rigidType()) {
        case RigidShape.eRigidType.eRigidCircle:
            status = this.collidedRectCirc(this, otherShape, collisionInfo);
            break;
        case RigidShape.eRigidType.eRigidRectangle:
            status = this.collidedRectRect(this, otherShape, collisionInfo);
            break;
    }
    return status;
};
```

Modifying the Rigid Circle Collision

Similarly to the `RigidRectangle` collision, the `RigidCircle_Collision.js` file must also be edited for extracting the collision information.

1. Modify the `collidedCircCirc()` function to calculate the collision depth and normal. In the following code, `vFrom1to2` defines the vector between the two circle centers. The collision depth is simply the difference between the length of the `vFrom1to2` vector and the sum of the circle radii, while the collision normal is simply `vFrom1to2`. Notice that when the two circles overlap completely, collision normal is defined to be along the positive y-direction.

```
RigidCircle.prototype.collidedCircCirc = function(c1, c2, collisionInfo) {
    var vFrom1to2 = [0, 0];
    vec2.sub(vFrom1to2, c2.getPosition(), c1.getPosition());
    var rSum = c1.getRadius() + c2.getRadius();
    var sqLen = vec2.squaredLength(vFrom1to2);
    if (sqLen > (rSum * rSum)) {
        return false;
    }
    var dist = Math.sqrt(sqLen);

    if (dist !== 0) { // overlapping
        vec2.scale(vFrom1to2, vFrom1to2, 1/dist);
        collisionInfo.setNormal(vFrom1to2);
        collisionInfo.setDepth(rSum - dist);
    }
    else // same position
    {
        collisionInfo.setDepth(rSum / 10);
        collisionInfo.setNormal([0, 1]);
    }
    return true;
};
```

2. Modify the `collided()` function to support the new `collisionInfo` parameter. Once again, note that the depth of the collision is initialized to zero by default and that the collision normal returned from `collidedRectCirc()` must be reversed for the `RigidCircle` object.

```
RigidCircle.prototype.collided = function(otherShape, collisionInfo) {
    var status = false;
    var n;
    collisionInfo.setDepth(0);
    switch (otherShape.rigidType()) {
        case RigidShape.eRigidType.eRigidCircle:
            status = this.collidedCircCirc(this, otherShape, collisionInfo);
            break;
        case RigidShape.eRigidType.eRigidRectangle:
            status = this.collidedRectCirc(otherShape, this, collisionInfo);
            n = collisionInfo.getNormal();
            n[0] = -n[0];
            n[1] = -n[1];
            break;
    }
    return status;
};
```

Defining the Engine Physics Component

A new physics engine component can now be defined to coordinate collision detections and collision responses via the relaxation iteration loops. To begin, follow the pattern of defining an engine component.

1. In the `src/Engine/Core` folder, create a new file and name it `Engine_Physics.js`. This file will implement the physics engine component. Remember to load this new source file in `index.html`.
2. Define the physics component following the JavaScript Module Pattern as follows:

```
var gEngine = gEngine || {};
// initialize the variable while ensuring it is not redefined

gEngine.Physics = (function () {
    var mPublic = { };
    return mPublic;
}());
```

3. Define variables for relaxation count, current relaxation loop count, has collided collision check, position correction rate, system acceleration, and collision information. In addition, create an initialization function to allocate and initialize the shared `mCollisionInfo` object.

```

var mRelaxationCount = 15;           // number of relaxation iteration
var mRelaxationOffset = 1/mRelaxationCount; // proportion to apply when scaling friction
var mPosCorrectionRate = 0.8;        // percentage of separation to project objects
var mSystemAcceleration = [0, -50];   // system-wide default acceleration

var mRelaxationLoopCount = 0;         // the current relaxation count
var mHasOneCollision = false;        // detect the first collision

var mCollisionInfo = null;           // information of the current collision

var initialize = function() {
    mCollisionInfo = new CollisionInfo(); // to avoid allocating this constantly
};

```

Resolving Collisions and Applying Friction

Define functions to resolve interpenetrating objects by the Projection Method and to slow down the moving objects by applying friction.

1. Define the function `_positionalCorrection()` to apply the Projection Method to correct the positions of the colliding objects. Calculate the `correctionAmount` in the collision normal direction as a function of the collision depth and mass of the respective objects. Note that the actual correction amount is inversely proportional to the mass of the object and scaled by the `mPosCorrectionRate`. For this reason, an object with infinite mass (`invMass` of 0) will not be positional corrected. In other words, this object will be unmovable (for example, a platform). The scaling by `mPosCorrectionRate` means that the interpenetration will not be resolved completely. Rather, it is expected that, through relaxation, the `_positionalCorrection()` function will be invoked multiple times in each game loop update.

```

var _positionalCorrection = function (s1, s2, collisionInfo) {
    var s1InvMass = s1.getInvMass();
    var s2InvMass = s2.getInvMass();
    var num = collisionInfo.getDepth() / (s1InvMass + s2InvMass) * mPosCorrectionRate;
    var correctionAmount = [0, 0];
    vec2.scale(correctionAmount, collisionInfo.getNormal(), num);

    var ca = [0, 0];
    vec2.scale(ca, correctionAmount, s1InvMass);
    var s1Pos = s1.getPosition();
    vec2.subtract(s1Pos, s1Pos, ca);

    vec2.scale(ca, correctionAmount, s2InvMass);
    var s2Pos = s2.getPosition();
    vec2.add(s2Pos, s2Pos, ca);
};

```

2. Create a function for applying friction to a colliding rigid shape. As illustrated in Figure 9-8, friction is simulated by applying a deceleration in the tangent direction, \hat{T} , of the collision. The actual deceleration size is scaled by the `mRelaxationOffset` to account for multiple invocations from relaxation.

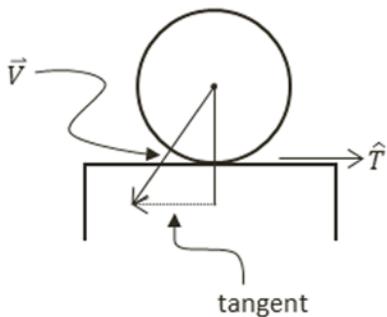


Figure 9-8. The tangent direction of a collision

```
// n is the collision normal
// v is the velocity
// f is the friction
// m is the invMass
var _applyFriction = function(n, v, f, m) {
    var tangent = vec2.fromValues(n[1], -n[0]); // perpendicular to n
    var tComponent = vec2.dot(v, tangent);
    if (Math.abs(tComponent) < 0.01)
        return;

    f *= m * mRelaxationOffset;
    if (tComponent < 0) {
        vec2.scale(tangent, tangent, -f);
    } else {
        vec2.scale(tangent, tangent, f);
    }
    vec2.sub(v, v, tangent);
};
```

3. You can now create the `resolveCollision()` function to implement the collision resolution between two colliding objects. The resolution procedure requires access to the two objects and the corresponding collision information. Notice that the function is defined based on the abstract `RigidBody` and thus is able to resolve the collisions between any combinations of `RigidCircle` and `RigidRectangle`. The following are the steps to resolve the collision between two `RigidShapes`:
 - a. Steps A sets `mHasOneCollision` to true to ensure that that relaxation loop will continue.
 - b. Steps B calls to `_positionalCorrection()` to apply positional correction to the objects to push them apart by 80 percent (by default) of the collision depth.
 - c. Step C calls to `_applyFriction()` to dampen the tangent component of the object velocities.

- d. Step D calculates the relative velocity between the two objects by subtracting them. This relative velocity is important for computing the impulse that pushes the objects apart.
- e. Step E computes `rVelocityInNormal`, the component of the relative velocity vector that is in the collision normal direction. This component indicates how rapidly the two objects are moving toward or away from each other. If `rVelocityInNormal` is greater than zero, then the objects are moving away from each other and impulse response will not be necessary.
- f. Step F computes the impulse magnitude, j , based on `rVelocityInNormal`, restitution (bounciness), and the masses of the colliding objects. This impulse magnitude value will be used to modify both velocities to push them apart.

Figure 9-9 depicts the details of step F. S1 and S2 are the two colliding `RigidShapes`, where the normal vector is the collision normal. Note that this vector always points from object S1 toward S2. The vectors `s1V-before` and `s2V-before` are the velocity vectors of the corresponding objects before step F, and the `s1V-after` and `s2V-after` are the impulse modified velocities after the step. Notice the two -after velocities would bring the corresponding objects away from the interpenetration.

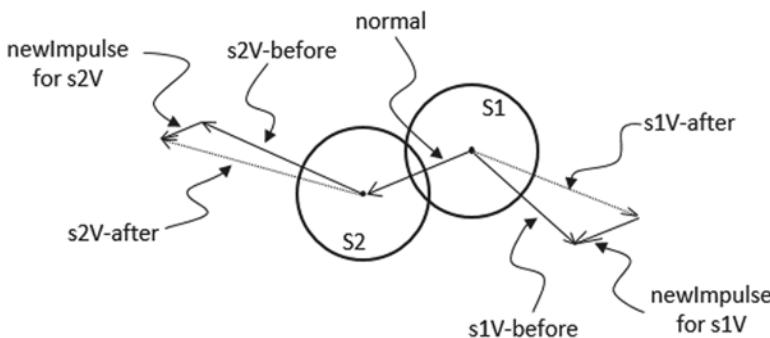


Figure 9-9. Computing new object velocities based on collision normal and impulse

```
var resolveCollision = function (s1, s2, collisionInfo) {
    // Step A: one collision has been found
    mHasOneCollision = true;

    // Step B: correct positions
    _positionalCorrection(s1, s2, collisionInfo);

    // Step C: apply friction
    var s1V = s1.getVelocity(); // collision normal direction is _against_ s2
    var s2V = s2.getVelocity();
    var n = collisionInfo.getNormal();
    _applyFriction(n, s1V, s1.getFriction(), s1.getInvMass());
    _applyFriction(n, s2V, -s2.getFriction(), s2.getInvMass());
```

```

// Step D: compute relatively velocity of the colliding objects
var relativeVelocity = [0, 0];
vec2.sub(relativeVelocity, s2V, s1V);

// Step E: examine the component in the normal direction
var rVelocityInNormal = vec2.dot(relativeVelocity, n);
if (rVelocityInNormal > 0) { //if objects moving apart ignore
    return;
}

// Step F: compute and apply response impulses for each object
var newRestituation = Math.min(s1.getRestitution(), s2.getRestitution());
var j = -(1 + newRestituation) * rVelocityInNormal;
j = j / (s1.getInvMass() + s2.getInvMass());

var impulse = [0, 0];
vec2.scale(impulse, collisionInfo.getNormal(), j);

var newImpulse = [0, 0];
vec2.scale(newImpulse, impulse, s1.getInvMass());
vec2.sub(s1V, s1V, newImpulse);

vec2.scale(newImpulse, impulse, s2.getInvMass());
vec2.add(s2V, s2V, newImpulse);
};


```

Adding the Relaxation Loop for Stability

With all the components of collision resolutions defined, you are now ready to define the relaxation support. Recall relaxation allows the collision resolution process to resolve the interpenetrations gradually over a predetermined number of times, in this case a default of 15. In general, larger relaxation iterations would result with a more natural appearance and more stable collision system. The associated computation cost is the main drawback with large relaxation iterations.

1. Continue with editing the `Engine_Physics.js` file and define the `beginRelaxation()` and `continueRelaxation()` functions to manage the relaxation iteration loop. Relaxation iterations can terminate when there are no more collisions detected.

```

var beginRelaxation = function() {
    mRelaxationLoopCount = mRelaxationCount;
    mHasOneCollision = true;
};
var continueRelaxation = function() {
    var oneCollision = mHasOneCollision;
    mHasOneCollision = false;
    mRelaxationLoopCount = mRelaxationLoopCount - 1;
    return ((mRelaxationLoopCount > 0) && oneCollision);
};

```

2. Define a `processObjObj()` function to support the relaxation of resolving the collision between two individual `GameObject` instances. The relaxation loop will continue only if the two objects continue to collide.

```
// Rigid Shape interactions: two game objects
var processObjObj = function(obj1, obj2) {
    var s1 = obj1.getPhysicsComponent();
    var s2 = obj2.getPhysicsComponent();
    if (s1 === s2)
        return;
    beginRelaxation();
    while (continueRelaxation()) {
        if (s1.collided(s2, mCollisionInfo)) {
            resolveCollision(s1, s2, mCollisionInfo);
        }
    }
};
```

3. Define a `processObjSet()` function to process the collision resolution between a `GameObject` instance and a `GameObjectSet`. Notice the nested loops. In the worst case, with the outer loop iterating at constant 15 times, the run time of this function is $O(N)$, where N is the size of the `RigidBody` set. It is important to be mindful of the cost of collision resolution.

```
// Rigid Shape interactions: a game object and a game object set
var processObjSet = function(obj, set) {
    var s1 = obj.getPhysicsComponent();
    var i, s2;
    beginRelaxation();
    while (continueRelaxation()) {
        for (i=0; i<set.size(); i++) {
            s2 = set.getObjectAt(i).getPhysicsComponent();
            if ((s1 !== s2) && (s1.collided(s2, mCollisionInfo))) {
                resolveCollision(s1, s2, mCollisionInfo);
            }
        }
    }
};
```

4. Define the `processSetSet()` function to process the collision resolution between two instances of `GameObjectSet`. Notice the triple nested loops. In the worst case, even with the outer loop iterating at constant 15 times, the run time of this function is $O(NM)$, where N and M are the sizes of the respective `GameObjectSet`. In other words, this function runs in quadratic times, and any large numbers of N or M will quickly degrade the performance of the system to non-real-time. It is important to design your system to avoid large set-to-set collisions.

```
// Rigid Shape interactions: two game object sets
var processSetSet = function(set1, set2) {
    var i, j, s1, s2;
    beginRelaxation();
```

```

        while (continueRelaxation()) {
            for (i=0; i<set1.size(); i++) {
                s1 = set1.getObjectAt(i).getPhysicsComponent();
                for (j=0; j<set2.size(); j++) {
                    s2 = set2.getObjectAt(j).getPhysicsComponent();
                    if ((s1 != s2) && (s1.collided(s2, mCollisionInfo))) {
                        resolveCollision(s1, s2, mCollisionInfo);
                    }
                }
            }
        };
    }
};

```

5. Create get and set accessors. Notice that since the relaxation rate is a percentage, it should remain between zero and one, and note that the relaxation count should never be below zero.

```

var getSystemAcceleration = function() { return mSystemAcceleration; };
var setSystemAcceleration = function(g) { mSystemAcceleration = g; };
var getRelaxationCorrectionRate = function() { return mPosCorrectionRate; };
var setRelaxationCorrectionRate = function(r) {
    if ((r <= 0) || (r>=1)) {
        r = 0.8;
    }
    mPosCorrectionRate = r;
};
var getRelaxationLoopCount = function() { return mRelaxationCount; };
var setRelaxationLoopCount = function(c) {
    if (c <= 0)
        c = 1;
    mRelaxationCount = c;
    mRelaxationOffset = 1/mRelaxationCount;
};

```

6. Provide public access to the functions.

```

var mPublic = {
    initialize: initialize,
    resolveCollision: resolveCollision,
    beginRelaxation: beginRelaxation,
    continueRelaxation: continueRelaxation,
    getSystemAcceleration: getSystemAcceleration,
    setSystemAcceleration: setSystemAcceleration,
    getRelaxationCorrectionRate: getRelaxationCorrectionRate,
    setRelaxationCorrectionRate: setRelaxationCorrectionRate,
    getRelaxationLoopCount: getRelaxationLoopCount,
    setRelaxationLoopCount: setRelaxationLoopCount,
    processObjObj: processObjObj,
    processObjSet: processObjSet,
    processSetSet: processSetSet
};
return mPublic;

```

Supporting the Engine Physics Component

A few minor modifications are required to integrate the physics component into the engine.

1. Edit the `Engine_Core.js` file by adding the following code to the `initializeEngineCore()` function to initialize the physics component of the engine:

```
gEngine.Physics.initialize();
```

2. Edit the `Engine_GameLoop.js` file by adding the following variables and functions to gain access to the time interval between frames. This is the delta time used in your numerical integration.

```
var kFrameTime = 1 / kFPS;
var kMPF = 1000 * kFrameTime; // Milliseconds per frame.
```

```
var getUpdateIntervalInSeconds = function () {
    return kFrameTime;
};

var mPublic = {
    start: start,
    stop: stop,
    getUpdateIntervalInSeconds: getUpdateIntervalInSeconds
};
```

Testing Impulse Resolution

The implementation should be tested in two ways, first to ensure moving objects collide and behave naturally and second to ensure the collision resolution system is stable when there are many objects in proximity. The implementation of the testing system has three interesting details.

- A new object type, `Wall`, is defined in the `Wall.js` file to represent the bounding walls of the world.
- It is important to take note to call the `RigidShape.setMass()` function to set the mass of stationary objects to zero, such as the platform and the wall.
- The new physics engine functionality is invoked from the `MyGame._physicsSimulation()` function defined in the `MyGame_Physics.js` file.

The rest of the testing system is largely similar to previous projects, and the detailed code listing will not be presented. Please refer to the source code files for the details of the implementations.

Observations

Run the project to test your results. Notice that right off the bat the hero character falls gradually to the floor and stops with a slight rebound. This is a clear indication that the base case for numerical integration, collision detection, and resolution all are operating as expected. Now notice the wandering Minion objects; they interact properly with the platforms and the walls of the game world with soft bounces and no apparent interpenetrations. Now, use the WASD keys to navigate the Hero object; make

sure to collide the Hero into the Minion objects and observe the apparent transfer of energies. A right-click releases home-in DyePack objects. This is to verify the proper functioning of the rest of the GameObject class. Notice that when the DyePack objects collide with the Platforms or with the Minions, they properly skirt the edges of the RigidBody. Additionally, observe that the relatively small mass of a DyePack object resulting in the DyePack has little effect on the Minion's trajectory when they collide. However, when many DyePack objects collide with the same Minion object at the same time, the results can have a significant effect on a Minion's movement.

The stability of the system can be tested by pressing the Z key, where many dumb Minion objects with white RigidBody bounds will be spawned. Take note that these white-bounded Minion objects are meant for testing system stability and will collide only with the Platform objects and among themselves. Notice how the white-bounded Minion objects settle and stack at the bottom of the scene; the lack of jitter in the objects here demonstrates the stability of the system. Now, press the C key a few times to excite the white-bounded Minions and watch the system settle back into a stable state. Alternatively, you can press the X key and select and move a white-bounded Minion to observe how the system settles back into a stable state after a local disturbance. Be careful with the number of white-bounded Minions you spawn. The quadratic runtime means the system performance can degrade quickly with even a modest number of Minions. You can press the C key to clear all the white-bounded Minions.

Particles and Particle Systems

A particle is a textured position with no defined dimensions. This description may seem contradictory because you have learned that a texture is an image and images are always defined by a width and height and will definitely occupy areas. The important clarification is that the game engine logic processes a particle as a position with no area, and the drawing system displays the particle as a texture. In this way, even though an actual displayed area is shown, the width and height dimensions of the texture are ignored by the underlying logic.

In addition to a position, a particle also has properties such as life span, color (for tinting the texture), and size (for scaling the texture). A particle system is a collection of particles with the same set of properties. By strategically spawning, manipulating the properties of, and removing particles in a particle system, interesting visual effects such as fire or explosions can be simulated.

In this section, you will create a simple and flexible particle system that includes the basic functionality required to achieve common effects, such as explosions and spell effects. Additionally, you will implement a particle shader to properly blend your particles within your scenes. The particles will collide and interact accordingly with the RigidBody objects.

The Particles Project

This project demonstrates how to implement a particle system to simulate explosion-like effects and to interact with the RigidBody object within the scene. You can see an example of this project running in Figure 9-10. The source code of this project is located in the Chapter9/9.3.Particles folder.



Figure 9-10. Running the Particles project

The controls of the project are as follows:

- *WASD keys:* Moves the Hero object
- *Left mouse button click in the game window:* Sends a DyePack chasing after the Hero
- *Z key:* Spawns particles at the current mouse position

The goals of the project are as follows:

- To understand the details of how to draw a particle and define its behaviors
- To experience implementing a particle system
- To build a particle engine component that supports interaction with RigidShape

You can find the following external resources in the assets folder: the fonts folder that contains the default system fonts and four texture images (`minion_sprite.png`, which defines the sprite elements for the hero and the minions; `platform.png`, which defines the platforms and floor and ceiling tiles; `wall.png`, which defines the walls; `dye_pack.png`, which defines the dye packs; and `particle.png`, which is used to draw one particle).

In the following exercise, 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. Lastly, you will create a new game engine component to manage the interaction of the particles with the `RigidShape` objects.

A particle is a new type of object for your game engine and requires the entire drawing system support, including custom GLSL shaders, default sharable shader instance, and a new Shader/Renderable pair.

Creating GLSL Particle Fragment Shader

Particles are textured positions with no areas. However, as discussed, your engine will draw each particle as a textured rectangle. For this reason, you can simply reuse the existing texture vertex shader `TextureVS.gls1`. When it comes to the actual computation of each pixel color, a new fragment shader, `ParticleFS.gls1`, must be created to properly simulate the intense brightness of fire and explosions. The new fragment shader aggressively composites colors to create oversaturation, where the resulting values of the RGB components quickly become the maximum displayable value of 1.0.

1. Under the `src/GLSLShaders` folder, add a new file and name it `ParticleFS.gls1`.
2. Similar to the texture fragment shader defined in `TextureFS.gls1`, you need to declare `uPixelColor` and `vTexCoord` to receive these values from the game engine and define the `uSampler` to sample the texture.

```
precision mediump float;
// sets the precision for floating point computation

// The object that fetches data from texture.
// Must be set outside the shader.
uniform sampler2D uSampler;

// Color of pixel
uniform vec4 uPixelColor;

// The "varying" keyword is for signifying that the texture coordinate will be
// interpolated and thus varies.
varying vec2 vTexCoord;
```

3. Implement the main function to accumulate colors aggressively.

```
void main(void) {
    // texel color look up based on interpolated UV value in vTexCoord
    vec4 c = texture2D(uSampler, vec2(vTexCoord.s, vTexCoord.t));

    vec3 r = vec3(c) * c.a * vec3(uPixelColor);
    vec4 result = vec4(r, uPixelColor.a);

    gl_FragColor = result;
}
```

Defining a Default ParticleShader Instance

You can now modify the engine to support the initializing, loading, and unloading of a new `ParticleShader`.

1. Begin by adding a variable for the particle shader in the `Engine_DefaultResources.js` file located in the `src/Engine/Core/Resources` folder. Also, define an accessor function as shown here:

```
var kParticleFS = "src/GLSLShaders/ParticleFS.gls1";
var mParticleShader = null;
var getParticleShader = function () { return mParticleShader };
```

2. Now instantiate a new particle shader in the `_createShaders()` function, as shown here:

```
var _createShaders = function(callBackFunction) {
    gEngine.ResourceMap.setLoadCompleteCallback(null);
    // ... identical to previous projects
    mParticleShader = new TextureShader(kTextureVS, kParticleFS);
    callBackFunction();
};
```

3. In the `initialize()` function, add the following code to properly load the file:

```
gEngine.TextFileLoader.loadTextFile(kParticleFS,
    gEngine.TextFileLoader.eTextFileType.eTextFile);
```

4. In the `cleanUp()` function, add the following line of code to unload the file when it is no longer needed:

```
gEngine.TextFileLoader.unloadTextFile(kParticleFS);
```

5. Don't forget to add the `get` accessor function to `mPublic()` so that it can be accessed.

```
getParticleShader: getParticleShader,
```

Creating the ParticleRenderable Object

Recall that a Shader/Renderable pair of objects must be defined to interface the GLSL shader to the game engine. With the default `ParticleShader` object defined to interface to the GLSL `ParticleFS` shader, you can now create a new Renderable object to support the drawing of particles. Fortunately, the detailed behaviors of a particle, or a textured position, is identical to that of a `TextureRenderable` with the exception of the different shader. As such, the definition of the `ParticleRenderable` object is trivial.

1. Under the `src/Engine/Renderables` folder, add a new file and name it `ParticleRenderable.js`. Remember to load this new source file in `index.html`.
2. Define the new renderable object as a simple subclass of the `TextureRenderable`, with the exception of a different default shader.

```
function ParticleRenderable(myTexture) {
    TextureRenderable.call(this, myTexture);
    Renderable.prototype._setShader.call(this,
        gEngine.DefaultResources.getParticleShader());
}
gEngine.Core.inheritPrototype(ParticleRenderable, TextureRenderable);
```

Defining the Particle and Particle Game Object

With the infrastructure for drawing a particle object defined, it is time to create the actual particle and define its behaviors.

Creating a Particle

To support moving particles around the scene, their movements can be approximated with the Symplectic Euler Integration.

1. Begin by creating a new subfolder called `Particles` under the `src/Engine` folder. In this folder, create a new file called `Particle.js`. Remember to load this new source file in `index.html`.
2. Create the constructor and add variables for position, velocity, acceleration, drag, and drawing variables for marking its position (for debugging).

```
function Particle(pos) {
    this.kPadding = 0.5;    // for drawing particle bounds

    this.mPosition = pos;  // this is likely to be a reference to xform.mPosition
    this.mVelocity = vec2.fromValues(0, 0);
    this.mAcceleration = gEngine.Particle.getSystemAcceleration();
    this.mDrag = 0.95;

    this.mPositionMark = new LineRenderable();
    this.mDrawBounds = false;
}
```

3. Create a `draw()` function to draw the position of the particle with a simple X. This function is to support the debugging of a particle system.

```
Particle.prototype.draw = function (aCamera) {
    if (!this.mDrawBounds) {
        return;
    }
    //calculation for the X at the particle position
    var x = this.mPosition[0];
    var y = this.mPosition[1];
    this.mPositionMark.setFirstVertex(x - this.kPadding, y + this.kPadding);           //TOP LEFT
    this.mPositionMark.setSecondVertex(x + this.kPadding, y - this.kPadding);          //BOTTOM RIGHT
    this.mPositionMark.draw(aCamera);
    this.mPositionMark.setFirstVertex(x + this.kPadding, y + this.kPadding);           //TOP RIGHT
    this.mPositionMark.setSecondVertex(x - this.kPadding, y - this.kPadding);          //BOTTOM LEFT
    this.mPositionMark.draw(aCamera);
};
```

4. You can now implement the `update()` function for calculating the particle's position. This is a straightforward implementation of the Symplectic Euler Integration. The `mDrag` variable simulates drags on the particles.

```
Particle.prototype.update = function () {
    var dt = gEngine.GameLoop.getUpdateIntervalInSeconds();
    // Symplectic Euler
    //   v += a * dt
    //   x += v * dt
    var 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);
};
```

5. Define simple get and set accessors. These functions are straightforward and are not listed here.

Creating the ParticleGameObject

With the actual particle object defined, you can now create the `ParticleGameObject` class to abstract the particle's behavior and to support the drawing of the particle.

1. Under the `src/Engine/Particles` folder, add a new file and name it `ParticleGameObject.js`. Remember to load this new source file in `index.html`.
2. Define a constructor to initialize variables for manipulating the color, size, and life span of the particle.

```
function ParticleGameObject(texture, atX, atY, cyclesToLive) {
    var renderableObj = new ParticleRenderable(texture);
    var xf = renderableObj.getXform();
    xf.setPosition(atX, atY);
    GameObject.call(this, renderableObj);

    var p = new Particle(xf.getPosition());
    this.setPhysicsComponent(p);

    this.mDeltaColor = [0, 0, 0, 0];
    this.mSizeDelta = 0;
    this.mCyclesToLive = cyclesToLive;
}
gEngine.Core.inheritPrototype(ParticleGameObject, GameObject);
```

3. Create a function for setting the final color of the particle. This function computes the required rate of change such that the color of the particle will be constantly changing during each update cycle until the particle expires.

```
ParticleGameObject.prototype.setFinalColor = function(f) {
    vec4.sub(this.mDeltaColor, f, this.mRenderComponent.getColor());
    if (this.mCyclesToLive !== 0) {
        vec4.scale(this.mDeltaColor, this.mDeltaColor, 1/this.mCyclesToLive);
    }
};
```

4. Define accessor functions to set the particle size change rate and to test whether the particle has expired. An expired particle should be removed from the system.

```
ParticleGameObject.prototype.setSizeDelta = function(d) { this.mSizeDelta = d; };
ParticleGameObject.prototype.hasExpired = function() {
    return (this.mCyclesToLive < 0); };
```

5. Add an update() function to change the color and size of the particle.

```
ParticleGameObject.prototype.update = function () {
    GameObject.prototype.update.call(this);

    this.mCyclesToLive--;
    var c = this.mRenderComponent.getColor();
    vec4.add(c, c, this.mDeltaColor);

    var xf = this.getXform();
    var s = xf.getWidth() * this.mSizeDelta;
    xf.setSize(s, s);
};
```

Creating the ParticleGameObjectSet

To work with a collection of particles, you can now create the `ParticleGameObjectSet` to support conveniently looping over `ParticleGameObject`s.

1. Under the `src/Engine/Particles` folder, add a new file and name it `ParticleGameObjectSet.js`. Remember to load this new source file in `index.html`.
2. Define `ParticleGameObjectSet` to be a subclass of `GameObjectSet`.

```
function ParticleGameObjectSet() {
    GameObjectSet.call(this);
}
gEngine.Core.inheritPrototype(ParticleGameObjectSet, GameObjectSet);
```

3. Override the `draw()` function of `GameObjectSet` to ensure particles are drawn with additive blending. Recall that the default `gl.blendFunc()` setting utilizes the alpha channel value to implement transparency. This is referred to as *alpha blending*. In this case, the `gl.blendFunc()` setting results in a simple accumulation of colors without considering the alpha channel. This is referred to as additive blending. As mentioned, to properly simulate the intense brightness of fire and explosions, it is desirable to aggressively accumulate colors such that the RGB components can quickly reach the maximum displayable value of 1.0. Additive blending facilitates the aggressive accumulation.

```
ParticleGameObjectSet.prototype.draw = function (aCamera) {
    var gl = gEngine.Core.getGL();
    gl.blendFunc(gl.ONE, gl.ONE); // for additive blending!
    GameObjectSet.prototype.draw.call(this, aCamera);
    gl.blendFunc(gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA); // restore alpha blending
};
```

4. Override the update() function to ensure expired particles are removed.

```
ParticleGameObjectSet.prototype.update = function () {
    GameObjectSet.prototype.update.call(this);

    // Cleanup Particles
    var i, obj;
    for (i=0; i<this.size(); i++) {
        obj = this.getObjectAt(i);
        if (obj.hasExpired()) {
            this.removeFromSet(obj);
        }
    }
};
```

Defining the Engine Particle Component

With the drawing and behavior infrastructures defined, you can now define the engine component to manage the particle system.

1. In the `src/Engine/Core` folder, add a new file and name it `Engine_Particle.js`. Remember to load this new source file in `index.html`.
2. Define the particle component following the JavaScript Module Pattern as follows:

```
var gEngine = gEngine || { };
// initialize the variable while ensuring it is not redefined

gEngine.Particle = (function () {
    var mPublic = { };
    return mPublic;
}());
```

3. Define variables for system acceleration and a common calculation space for heavily used variables.

```
var mSystemAcceleration = [0, -50.0];

// the follows are scratch workspace for vec2
var mFrom1to2 = [0, 0];
var mVec = [0, 0];
var mNormal = [0, 0];
```

4. Add a function to resolve the collision between a circle and a particle. This can be accomplished by a simple radius check. Particles inside a circle are projected out. Notice that, in this case, with a zero-area particle, collision resolution is straightforward to handle and that there is no need for relaxation.

```
var resolveCirclePos = function (circShape, particle) {
    var collided = false;
    var pos = particle.getPosition();
    var cPos = circShape.getPosition();
    vec2.subtract(mFrom1to2, pos, cPos);
    var dist = vec2.length(mFrom1to2);
    if (dist < circShape.getRadius()) {
        vec2.scale(mFrom1to2, mFrom1to2, 1/dist);
        vec2.scaleAndAdd(pos, cPos, mFrom1to2, circShape.getRadius());
        collided = true;
    }
    return collided;
};
```

5. Similar to the case of a circle, add the `resolveRectPos()` function to resolve the collision between a rectangle and a particle. Once again, particles inside a rectangle are simply projected out along the nearest x- or y-boundary.

```
var resolveRectPos = function (rectShape, particle) {
    var collided = false;
    var alongX = rectShape.getWidth() / 2;
    var alongY = rectShape.getHeight() / 2;

    var pos = particle.getPosition();
    var rPos = rectShape.getPosition();

    var rMinX = rPos[0] - alongX;
    var rMaxX = rPos[0] + alongX;
    var rMinY = rPos[1] - alongY;
    var rMaxY = rPos[1] + alongY;

    collided = ((rMinX<pos[0]) && (rMinY<pos[1]) &&
                (rMaxX>pos[0]) && (rMaxY>pos[1]));

    if (collided) {
        vec2.subtract(mFrom1to2, pos, rPos);
        mVec[0] = mFrom1to2[0];
        mVec[1] = mFrom1to2[1];

        // Find closest axis
        if (Math.abs(mFrom1to2[0] - alongX) < Math.abs(mFrom1to2[1] - alongY)) {
            // Clamp to closest side
            mNormal[0] = 0;
            mNormal[1] = 1;
            if (mVec[0] > 0) {
                mVec[0] = alongX;
            }
        } else {
            mNormal[0] = 1;
            mNormal[1] = 0;
            if (mVec[1] > 0) {
                mVec[1] = alongY;
            }
        }
    }
};
```

```

        } else {
            mVec[0] = -alongX;
        }
    } else { // y axis is shorter
        mNormal[0] = 1;
        mNormal[1] = 0;
        // Clamp to closest side
        if (mVec[1] > 0) {
            mVec[1] = alongY;
        } else {
            mVec[1] = -alongY;
        }
    }
}

vec2.subtract(mVec, mVec, mFrom1to2);
vec2.add(pos, pos, mVec); // remember pos is particle position
}
return collided;
};

```

6. Define a function for processing the collision between an object and a set of particles.

```

// Rigid Shape interactions: a game object and a set of particle game objects
var processObjSet = function(obj, pSet) {
    var s1 = obj.getPhysicsComponent(); // a RigidBody
    var i, p;
    for (i=0; i<pSet.size(); i++) {
        p = pSet.getObjectAt(i).getPhysicsComponent(); // a Particle
        s1.resolveParticleCollision(p);
    }
};

```

7. Define a function to process the collision between a GameObjectSet and a set of particles.

```

// Rigid Shape interactions: game object set and a set of particle game objects
var processSetSet = function(objSet, pSet) {
    var i;
    for (i=0; i<objSet.size(); i++) {
        processObjSet(objSet.getObjectAt(i), pSet);
    }
};

```

8. Finally, implement the necessary get and set accessors for the system's acceleration and remember to add the necessary functions to `mPublic`.

```

var getSystemAcceleration = function() { return mSystemAcceleration; };
var setSystemAcceleration = function(g) { mSystemAcceleration = g; };

```

```

var mPublic = {
    getSystemAcceleration: getSystemAcceleration,
    setSystemAcceleration: setSystemAcceleration,
    resolveCirclePos: resolveCirclePos,
    resolveRectPos: resolveRectPos,
    processObjSet: processObjSet,
    processSetSet: processSetSet
};
return mPublic;

```

Interacting Particles with RigidBody

To properly support the interaction between Particle objects and RigidBody objects, edit the `RigidBody_Collision.js` file and define the following function to define the corresponding collision function:

```

RigidBody.prototype.resolveParticleCollision = function(aParticle) {
    var status = false;
    switch (this.rigidType()) {
        case RigidBody.eRigidType.eRigidCircle:
            status = gEngine.Particle.resolveCirclePos(this, aParticle);
            break;
        case RigidBody.eRigidType.eRigidRectangle:
            status = gEngine.Particle.resolveRectPos(this, aParticle);
            break;
    }
    return status;
};

```

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 capable of colliding with RigidBody objects and resulting in expected behaviors. The test case is based mainly on the previous project with one simple modification—invoking the `_createParticle()` function when the Z key is pressed. Please consult the source code for the details of the implementation.

The `_createParticle()` function listed in the following configures and creates particles. There are two important observations. First, the `random()` function is used many times to configure each created Particle. Particle systems utilize large numbers of similar yet slightly different particles to build and convey a sense of the desired visual effect. It is important to avoid patterns of any sort, and this use of randomness is an important rule to follow. Second, there are many seemingly arbitrary numbers 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, unfortunately is the nature of working with particle systems: there is quite a bit of ad hoc trying. Commercial game engines typically alleviate this difficulty by releasing a collection of preset values for their particle systems. In this way, game designers can fine-tune specific desired effects by tweaking the provided presets.

```

MyGame.prototype._createParticle = function(atX, atY) {
    var life = 30 + Math.random() * 200;
    var p = new ParticleGameObject(this.kParticleTexture, atX, atY, life);
    p.getRenderable().setColor([1, 0, 0, 1]);
}

```

```

// size of the particle
var r = 5.5 + Math.random() * 0.5;
p.getXform().setSize(r, r);

// final color
var fr = 3.5 + Math.random();
var fg = 0.4 + 0.1 * Math.random();
var fb = 0.3 + 0.1 * Math.random();
p.setFinalColor([fr, fg, fb, 0.6]);

// velocity on the particle
var fx = 10 - 20 * Math.random();
var fy = 10 * Math.random();
p.getPhysicsComponent().setVelocity([fx, fy]);

// size delta
p.setSizeDelta(0.98);

return p;
};

```

Observations

Run the project and press the Z key to observe the generated particles. It appears as though there is combustion occurring underneath the mouse pointer. Hold the Z key and move the mouse pointer around slowly to observe the combustion following the mouse as though there is an engine generating flames under the mouse pointer. If you move the mouse pointer quickly, you will observe individual pink circles changing color while dropping toward the floor. Particle systems must be fine-tuned for individual situations. Now, hold the Z key and move the mouse pointer over any of the other `RigidShape` objects, such as one of the Minions. Observe that particles are projected to the boundary and continue to behave as expected. In addition, notice that the particles skirt the `RigidShape` object bounds when collided, just as you would expect.

Particle Emitters

With your current particle system implementation, you can create particles at a specific point and time. These particles when created can move and change based on their properties. However, particles can be created only when there is an explicit state change such as a key click. This becomes restricting when it is desirable to persist the generation of particles after the state change, such as an explosion that persists for a short while after the dye pack collides with the hero. A particle emitter addresses this issue by defining the functionality of generating particles over a time period.

The Particle Emitters Project

This project demonstrates how to implement a particle emitter for your particle system to support emitting particles over time. You can see an example of this project running in Figure 9-11. The source code of this project is located in the Chapter9/9.4.ParticleEmitters folder.



Figure 9-11. Running the Particle Emitters project

The controls of the project are as follows:

- *WASD keys*: Moves the Hero object
- *Left mouse button click in the game window*: Sends a DyePack chasing after the Hero
- *Z key*: Spawns particles at the current mouse position

The goals of the project are as follows:

- To understand the need for particle emitters
- To experience implementing particle emitters

You can find the following external resources in the assets folder: the fonts folder that contains the default system fonts and four texture images (`minion_sprite.png`, which defines the sprite elements for the hero and the minions; `platform.png`, which defines the platforms and floor and ceiling tiles; `wall.png`, which defines the walls; `dye_pack.png`, which defines the dye packs; and `particle.png`, which is used to draw one particle).

Creating the ParticleEmitter Object

Recall that when working with particles, it is important to avoid patterns. In this case, as the `ParticleEmitter` object generates new particles over time, it is important to build in randomness to avoid any patterns.

1. Under the `src/Engine/Particles` folder, add a new file and name it `ParticleEmitter.js`. Remember to load this new source file in `index.html`.

2. Create a constructor to define and set the default number of particles to emit at each cycle, the emitter's position, the number of particles left to be emitted, and `mParticleCreator`, the callback function for actual particles creation.

```
function ParticleEmitter(pos, num, createrFunc) {
    // Smallest number of particle emitted per cycle
    this.kMinToEmit = 5;

    // Emitter position
    this.mEmitPosition = pos; // this can be a reference to a xform.mPosition

    // Number of particles left to be emitted
    this.mNumRemains = num;

    this.mParticleCreator = createrFunc;
}
```

3. Define a function to return the functioning status for the emitter. When there are no more particles to emit, the emitters should be removed.

```
ParticleEmitter.prototype.expired = function () { return (this.mNumRemains <= 0); };
```

4. Create a function to actually emit particles. Take note of the randomness in the number of particles that are actually emitted and the invocation of the `mParticleCreator()` callback function. With this design, there would be no patterns in the number of particles that are created over time. In addition, the emitter defines only the mechanisms of how, when, and where particles will be emitted and does not define the characteristics of the created particles. The function pointed to by `mParticleCreator` is responsible for defining that.

```
ParticleEmitter.prototype.emitParticles = function (pSet) {
    var numToEmit = 0;
    if (this.mNumRemains < this.kMinToEmit) {
        // If only a few are left, emits all of them
        numToEmit = this.mNumRemains;
    } else {
        // Otherwise, emits about 20% of what's left
        numToEmit = Math.random() * 0.2 * this.mNumRemains;
    }
    // Left for future emitting.
    this.mNumRemains -= numToEmit;
    var i, p;
    for (i = 0; i < numToEmit; i++) {
        p = this.mParticleCreator(this.mEmitPosition[0], this.mEmitPosition[1]);
        pSet.addToSet(p);
    }
};
```

Modifying the Particle Game Object Set

The defined `ParticleEmitter` object needs to be integrated into `ParticleGameObjectSet` to manage the emitted particles.

1. Edit the `ParticleGameObjectSet.js` file; in the constructor, define and initialize a particle emitter set.

```
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.

```
ParticleGameObjectSet.prototype.addEmitterAt = function (p, n, func) {
    var e = new ParticleEmitter(p, n, func);
    this.mEmitterSet.push(e);
};
```

3. Modify the update function to loop through the emitter set to generate new particles. Expired emitters are removed.

```
ParticleGameObjectSet.prototype.update = function () {
    // ... identical to previous project

    // Emit new particles
    for (i=0; i<this.mEmitterSet.length; i++) {
        e = this.mEmitterSet[i];
        e.emitParticles(this);
        if (e.expired()) {
            this.mEmitterSet.splice(i, 1);
        }
    }
};
```

Testing the Particle Emitter

This is a straightforward testing of the correct functioning of the `ParticleEmitter` object. The two interesting implementation details are as follows:

1. In the `Hero.js` file, at the end of the `update()` function, when a collision between a dye pack and the hero is detected, a new `ParticleEmitter` object is instantiated with the `func` callback function.

```
for (i=0; i<dyePacks.size(); i++) {
    obj = dyePacks.getObjectAt(i);
    // chase after hero
    obj.rotateObjPointTo(p, 0.8);
    if (obj.pixelTouches(this, collisionPt)) {
        dyePacks.removeFromSet(obj);
        allParticles.addEmitterAt(collisionPt, 200, func);
    }
}
```

The `func` parameter is defined in the `update()` function of the `MyGame.js` file as follows:

```
var func = function(x, y) { this.createParticle.call(this, x, y); };
```

where the `createParticle()` function is identical to the one from the previous project.

2. The `RigidBody` and `Particle` system interaction is triggered from the end of the `_physicsSimulation()` function defined in the `MyGame_Physics.js` file as follows:

```
MyGame.prototype._physicsSimulation = function() {
    // ... identical to previous project

    // Particle system collisions
    gEngine.Particle.processSetSet(this.mAllMinions, this.mAllParticles);
    gEngine.Particle.processSetSet(this.mAllPlatforms, this.mAllParticles);
};
```

As previously, the rest of the implementation is straightforward and will not be listed to avoid unnecessary distraction. Please refer to the source code in the `src/MyGame` folder for details.

Observations

Run the project and see the same behavior as in the previous project. The only exception is the new explosion effects after the dye packs collide with the `Hero` object. Notice how each explosion is triggered by the collision and persists for a short while before disappearing gradually. Comparing this effect with the one resulting from a short tapping of the `Z` key, observe that without a dedicated particle emitter, the explosion seems to have fizzled before it begins.

Summary

This chapter led you in building a simple yet flexible physics infrastructure for your game engine. Focusing on the implementation of core concepts, you learned to approximate Newtonian motion integrals with the Symplectic Euler Integration and how to build a numerically stable physics simulation that includes relaxation, collisions detection computation between axis-aligned rectangles and circles, extraction of collision normal and depth, and collision resolution using the Projection and Impulse methods. In addition, you have integrated a basic particle system with your physics engine with particles that interact with the `RigidBody` objects in the system. Through working with your particle system, you have learned that appropriate use of randomness is important and that creating interesting visual effects requires hands-on experience and fine-tuning iterations.

Chapter 8 complemented Chapter 5 by covering the simulation of illumination to improve the fidelity of the drawn game objects. In turn, this chapter augmented Chapter 6 by presenting the approximation of Newtonian motions to support the increase in behavioral complexity of the interacting game objects. The next chapter will extend Chapter 7 and describe more advanced topics in working with the camera.

Game Design Considerations

The puzzle level I've been building in the examples to this point has focused entirely on creating an understandable and consistent logical challenge. I've avoided burdening the exercise with any kind of visual design, narrative, or setting (design elements traditionally associated with enhancing player presence) to ensure you're thinking only about the rules of play without introducing superfluous distractions. However, as you create core game mechanics, it's important to understand how certain elements of gameplay can contribute directly to presence (recall that *presence* is the player's sense of feeling as if they're connected to the game world). The logical rules and requirements of core mechanics often have no effect on presence until they're paired with an interaction model, audiovisual design, or game setting, although some elements of game mechanics can directly influence presence. As discussed in Chapter 8, lighting is one example of a typically presence-enhancing visual design element that can also be used directly as a core game mechanic; object physics is similarly a presence-enhancing design element that's perhaps even more frequently connected directly to core game mechanics.

Our entire experience in the physical world is governed by physical laws, and when you model physics in virtual worlds, it makes an immediate and natural connection to the physical world and will therefore frequently strengthen presence. An example of object physics enhancing presence but not contributing to design would be destructible environments that have no direct impact on gameplay. If you're playing a first-person shooter, for example, and shoot at crates and other game objects that respond by realistically exploding on impact or if you 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 the game mechanic. If you're playing Angry Birds, however, and launching one of the birds from your slingshot into the game space and must aim your shot based on the physics-modeled parabolic arc the bird follows upon launch, as shown in Figure 9-12, that's an example of physics being used as both a core element of gameplay while also enhancing presence. In fact, any game that involves jumping a hero character in an environment with simulated gravity is an example of physics contributing to both presence and the core mechanic, so most platformer games utilize physics as both a core mechanic and a presence-enhancing design element.

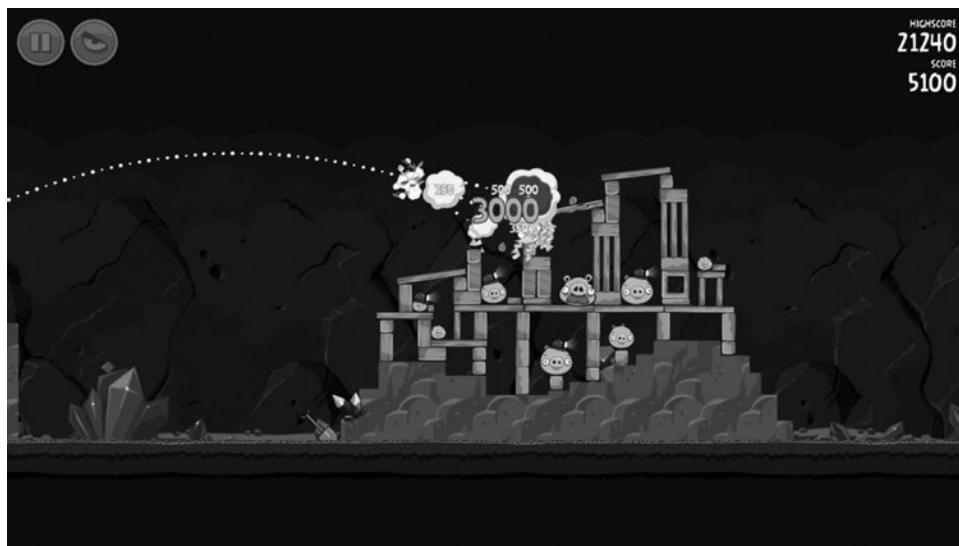


Figure 9-12. Rovio's Angry Birds requires players to launch projectiles from a slingshot in a virtual world that models gravity, mass, momentum, and object collision detection. The game physics are a fundamental component of the game mechanic and enhance the sense of presence by assigning physical world traits to virtual objects

The Rigid Shape Bounds and Rigid Shape Impulse projects introduce you to the powerful ability of physics to bring players into the game world. Instead of simply moving the hero character like a screen cursor, the player is now experiencing simulated inertia, momentum, and gravity requiring the same kind of predictive assessments around aiming, timing, and trajectory that would exist when manipulating objects in the physical world, and objects in the game are now colliding in a manner similarly anchored in physical experience. Even though specific values might take a detour from the physical world in a simulated game space (e.g., lower or higher gravity, more or less inertia, and the like), as long as the relationships are consistent and reasonably analogous to our physical experience, presence will typically increase when these effects are added to game objects.

As with earlier projects, it's possible to use basic behaviors to explore new game mechanics. Imagine, for example, a game level where the hero character in the Rigid Shape Impulse project was required to push all the robots into a specific area within a specified time limit while avoiding being hit with the tracking projectiles. Imagine the same level without physics, and it's a different experience (although not necessarily inferior). The Verlet Particles and Particle Emitters projects provide an example where object physics contribute to presence but don't necessarily have a direct impact on the game mechanic. If the tracking projectiles explode on impact without any kind of simulated particle effect (perhaps using a sprite-based animation to show an explosion), it will potentially be less effective as a presence-enhancing effect than a particle simulation based on a physically accurate model of particle dispersion (again, because we experience combustion in the physical world in a way that follows the laws of physics).

At this point in the discussion it's worth stepping back and interrogating the notion that presence is *necessarily* increased by modeling physical world experience in game environments. While modeling physical behavior in virtual environments is a common and high-confidence method of connecting players to game worlds, there are frequently other style choices that can be quite effective at bringing players into the game. In the case of a particle emitter versus an animated sprite to show an explosion, consider a game featuring a comic book style and displaying a "BLAM!" graphic to signal an explosion. Objects don't show the word "BLAM!" when they explode in the physical world, of course, but the highly stylized use of "BLAM!" in the context of a comic book aesthetic can be quite effective at bringing players into the game world. Many AAA studios have embraced the notion that high-fidelity environments featuring physically accurate materials and robust physics are critical for enhancing presence, although the recent surge of independent studios working in a myriad of different styles is challenging this notion.

I left the level design in Chapter 8 with an interesting two-stage mechanic that focused almost exclusively on abstract logical rules and therefore hadn't yet considered elements that would add presence to the experience and bring players into the game world. Recall the current state of the level in Figure 9-13.

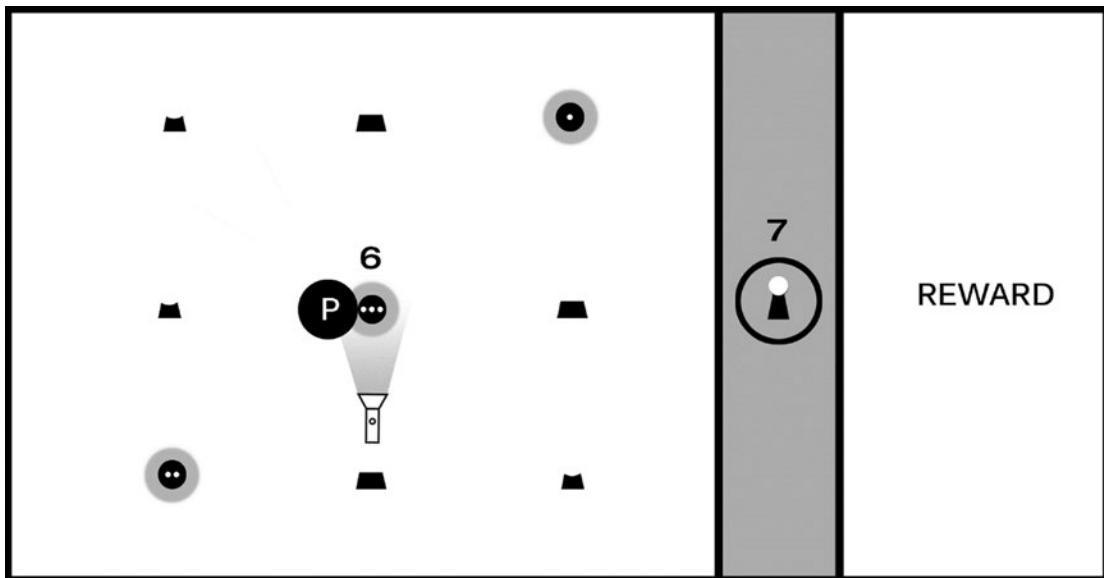


Figure 9-13. The level as it currently stands includes a two-step puzzle first requiring players to move a flashlight and reveal hidden symbols; the player must then activate the shapes in the correct sequence to unlock the barrier and claim the reward

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 conveyed 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 fairly weak at this stage, however, as you have yet to place the game experience in a setting or context and the intentionally generic shapes don’t provide the game with any kind of visual identity the player can relate to. If you decided to take this mechanic further, now would be a reasonable time to begin exploring the game’s setting. This example used a flashlight metaphor to reveal hidden symbols, but it’s now possible to abstract the mechanic’s logical rules from the current implementation. This mechanic can be described as follows: “The player must explore the environment to find tools required to assemble a sequence in the correct order”

For the next iteration of the mechanic, let’s revisit the interaction model and evolve it from purely a logic puzzle to something a bit more active that makes use of object physics. Figure 9-14 changes the game screen to include a jumping component.

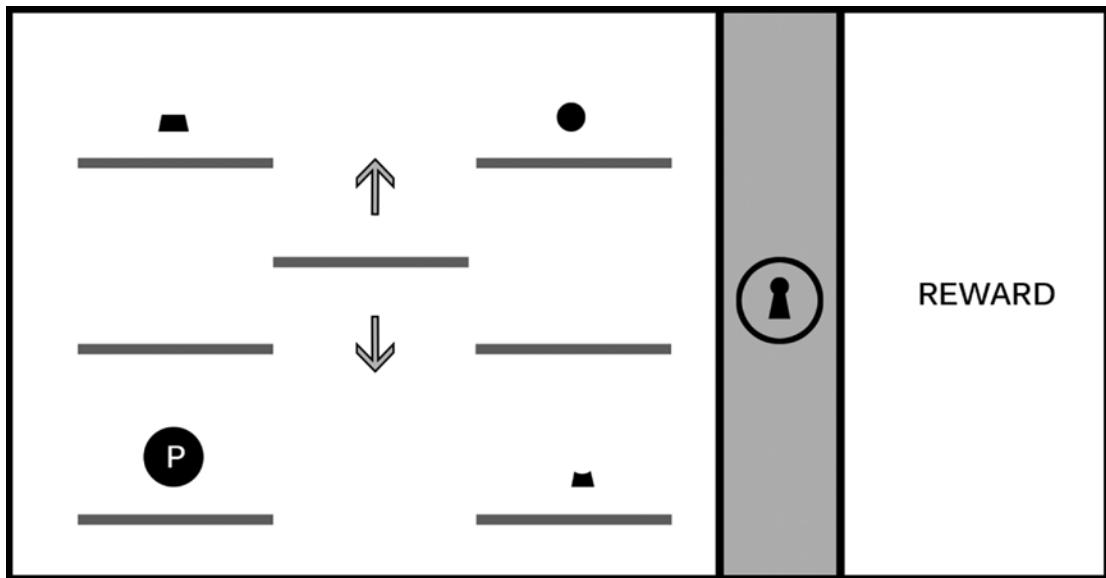


Figure 9-14. The game screen now shows just one instance of each part of the lock (top, middle, bottom), and the hero character moves in the manner of a traditional jumping 2D platformer. The six platforms on the left and right are stationary, and the middle platform moves up and down, allowing the player 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.)

The mechanic is evolving to include a dexterity challenge (in this case, timing the jumps), yet it retains the logical rules from the earlier iteration: the shapes must be activated in the correct order to unlock the barrier blocking the reward. Imagine the player experiences this screen for the first time; they’ll begin exploring the screen to learn the rules of engagement for the level, including the interaction model (the keys and/or mouse buttons used to move and jump the hero character), whether missing a jump results in a penalty (for example, 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.

Figure 9-14 is the beginning of an interesting (although still basic) platformer puzzle, but you’ve now simplified the solution compared to the earlier iteration, and the platformer jumping component isn’t especially challenging. 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; you can add a similar second challenge to the platformer component, as shown in Figure 9-15.

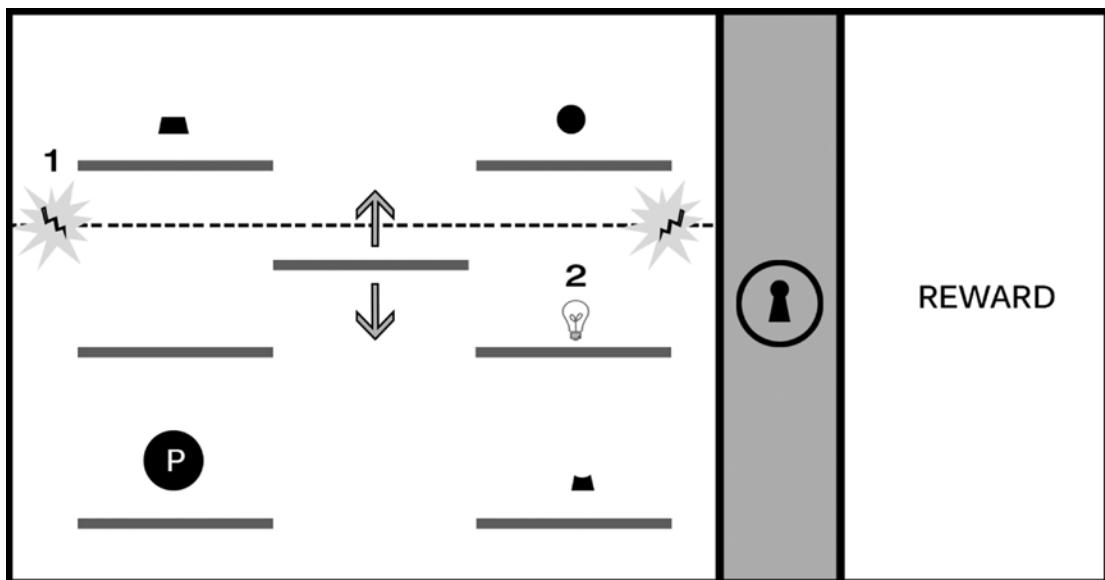


Figure 9-15. The introduction of a force field blocking access to the upper platforms (#1) can significantly increase the challenge of the platformer component. In this design, the player must activate the switch (represented with a lightbulb in #2) to disable the force field and reach the first and third shapes

The introduction of a force field opens a number 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 engaged and that the puzzle will reset if all shapes aren't activated before the force field is reengaged.

You've now taken an elemental mechanic based on a logical sequence and adapted it to support an action platformer experience. At this stage, the mechanic is becoming more interesting and beginning to feel more like a playable level, but it's still lacking setting and context, both of which will influence the further evolution and implementation of this particular mechanic. This is a good opportunity to explore the kind of story you might want to tell with this game. Are you interested in a sci-fi adventure, perhaps a survival horror experience, or maybe a series of puzzle levels with no connected narrative? The setting you choose will not only help inform the visual identity of the game but can also guide your decisions on the kinds of challenges you create for players (that is, are "enemies" in the game working against you, or will you keep the emphasis on solving logic puzzles?). A good exercise to practice connecting a mechanic to a game setting is to pick a place (for example, the interior of a space ship) and begin contextualizing the mechanic in that scenario and defining the elements of the challenge in a way that make sense for the setting. For a spaceship setting, perhaps something has gone wrong and you must make your way from one end of the ship to the other while neutralizing security lasers through the clever use of environment objects. Experiment with applying the spaceship setting to the current mechanic and adjusting the elements in the level to fit that theme: lasers are just one option, but can you think of other uses for the mechanic that don't involve the unlocking sequence? Practice working with the structure of the mechanic from the examples and try applying it to a range of different settings to begin building your comfort for taking abstract mechanics and applying them to specific settings.

Remember, including object physics in your level designs isn't a universal requirement. Game design is infinitely flexible, and sometimes you may want to subvert or completely ignore the laws of physics in the game worlds you create. The final quality of your game experience is the result of how effectively you harmonize all nine elements of game design; it's not about the mandatory implementation of any one design option. Your game might be completely abstract and involve shapes and forms shifting in space in a way that has no bearing on the physical world, or it might take place in an analogous real-world environment; yet, your mechanic may not rely on object physics at all, and your use of color, audio, and narrative might still combine to create an experience with a strong presence for players. However, if you find yourself with a game environment that seeks to convey a sense of physicality by making use of objects that people can relate to objects found in the physical world, it's worth exploring how object physics might enhance the experience.

CHAPTER 10



Supporting Camera Background

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

Introduction

By this point your game engine is capable of illuminating 2D images to generate highlights and shadows and of simulating basic physical behaviors. To complete the engine development, this chapter focuses on the general support for creating the game world environment with background tiling and parallax and relieving the game programmers from having to manage draw ordering. Background images or objects are included to decorate the game world to further engage the players; this often requires 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.

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 the duplicating boundaries. Figure 10-1 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 duplication boundaries. Proper tiling conveys a sense of complexity in a boundless game world by creating only a single image.

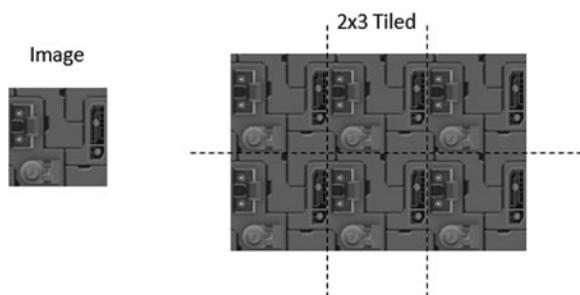


Figure 10-1. Tiling of a strategically drawn background image

Parallax is the apparent displacements of objects when the objects are viewed from different positions. Figure 10-2 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 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.

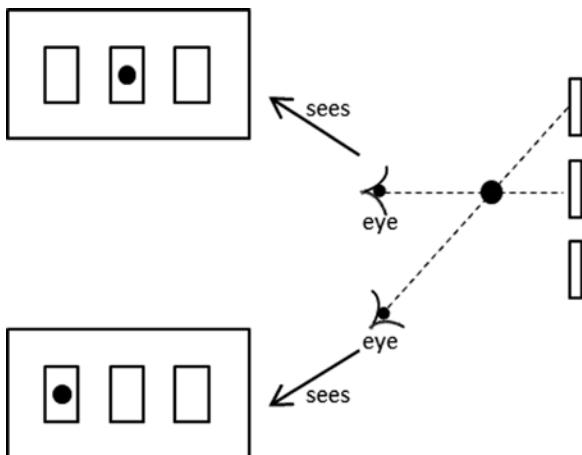


Figure 10-2. Parallax: seeing different positions of objects from different viewpoints

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 result in covering the camera WC bounds need to be drawn. This is illustrated in Figure 10-3. 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 10-3 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 that located at the origin does not need to be drawn.

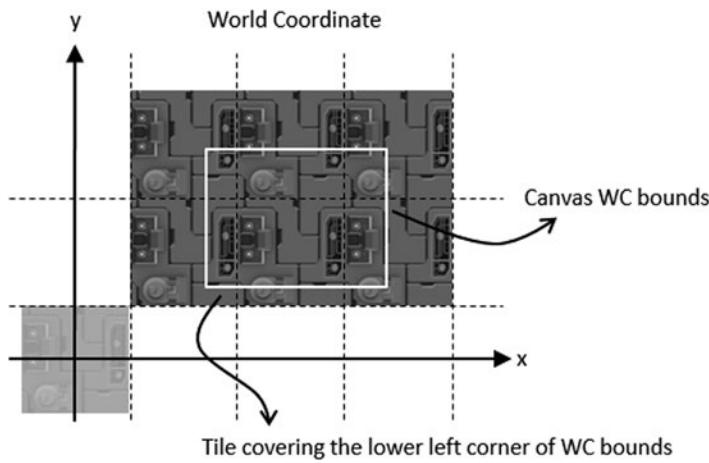


Figure 10-3. Generating tiled background for camera WC bounds

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.

The Tiled Objects Project

This project demonstrates how to implement simple background tiling. You can see an example of this project running in Figure 10-4. The source code to this project is defined in the Chapter10/10.1.TiledObjects folder.

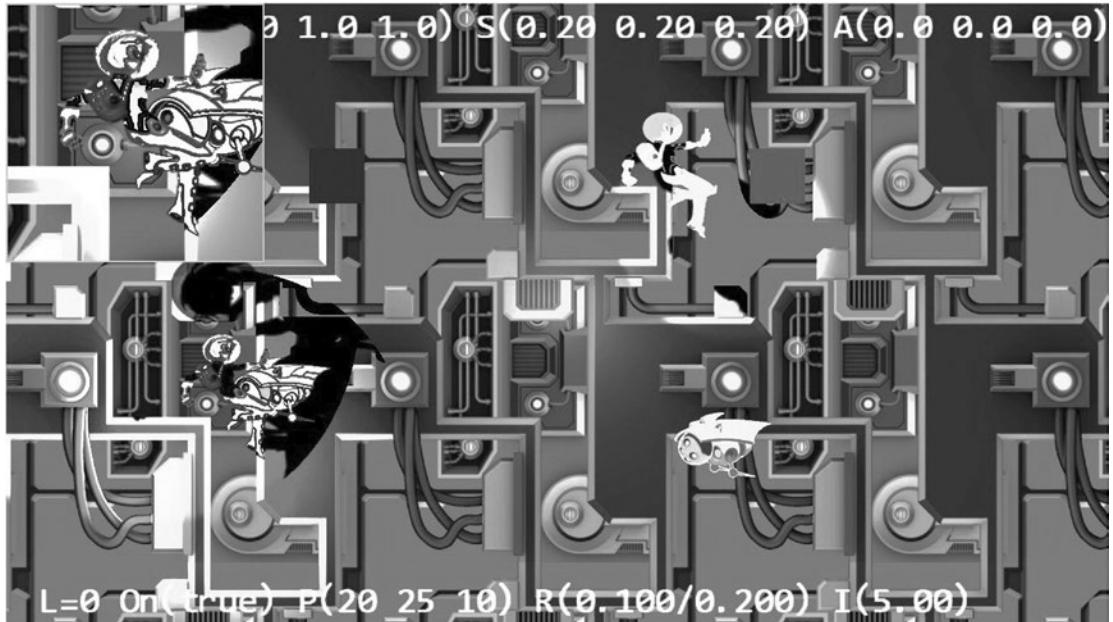


Figure 10-4. Running the Tiled Objects project

The controls of the project are 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 that contains the default system fonts and six texture images (`minion_sprite.png`, `minion_sprite_normal.png`, `bg.png`, `bg_normal.png`, `byLayer.png`, and `bgLayer_normal.png`). The Hero and Minion objects are represented by sprite elements in the `minion_sprite.png` image, and `bg.png` and `bgLayer.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/GameObjects/` folder and name it `TiledGameObject.js`. Add the following code to construct the object:

```
function TiledGameObject(renderableObj) {
    this.mShouldTile = true; // can switch this off if desired
    GameObject.call(this, renderableObj);
}
gEngine.Core.inheritPrototype(TiledGameObject, GameObject);
```

The `TiledGameObject` object has only one variable, `mShouldTile`, which is a Boolean that determines whether the object should tile.

2. Define the getter and setter functions for `mShouldTile`.

```
TiledGameObject.prototype.setIsTiled = function (t) {
    this.mShouldTile = t;
};

TiledGameObject.prototype.shouldTile = function () {
    return this.mShouldTile;
};
```

3. Define the function to tile and draw the `Renderable` object to cover the WC bounds of the `aCamera` object.

```
TiledGameObject.prototype._drawTile = function(aCamera) {
    // Step A: Compute the positions and dimensions of tiling object.
    var xf = this.getXform();
    var w = xf.getWidth();
    var h = xf.getHeight();
    var pos = xf.getPosition();
```

```

var left = pos[0] - (w/2);
var right = left + w;
var top = pos[1] + (h/2);
var bottom = top - h;

// Step B: Get the world positions and dimensions of the drawing camera.
var wcPos = aCamera.getWCCenter();
var wcLeft = wcPos[0] - (aCamera.getWCWidth() / 2);
var wcRight = wcLeft + aCamera.getWCWidth();
var wcBottom = wcPos[1] - (aCamera.getWCHeight() / 2);
var wcTop = wcBottom + aCamera.getWCHeight();

// Step C: Determine the offset to the camera window's lower left corner.
var dx = 0, dy = 0; // offset to the lower left corner
// left/right boundary?
if (right < wcLeft) { // left of WC left
    dx = Math.ceil((wcLeft - right)/w) * w;
} else {
    if (left > wcLeft) { // not touching the left side
        dx = -Math.ceil((left-wcLeft)/w) * w;
    }
}
// top/bottom boundary
if (top < wcBottom) { // Lower than the WC bottom
    dy = Math.ceil((wcBottom - top)/h) * h;
} else {
    if (bottom > wcBottom) { // not touching the bottom
        dy = -Math.ceil((bottom - wcBottom)/h) * h;
    }
}

// Step D: Save the original position of the tiling object.
var sX = pos[0];
var sY = pos[1];

// Step E: Offset tiling object and modify the related position variables.
xf.incXPosBy(dx);
xf.incYPosBy(dy);
right = pos[0] + (w/2);
top = pos[1] + (h/2);

// Step F: Determine the number of times to tile in the x and y directions.
var nx = 1, ny = 1; // number of times to draw in the x and y directions
nx = Math.ceil((wcRight - right) / w);
ny = Math.ceil((wcTop - top) / h);

// Step G: Loop through each location to draw a tile.
var cx = nx;
var xPos = pos[0];
while (ny >= 0) {
    cx = nx;

```

```

        pos[0] = xPos;
        while (cx >= 0) {
            this.mRenderComponent.draw(aCamera);
            xf.incXPosBy(w);
            --cx;
        }
        xf.incYPosBy(h);
        --ny;
    }

    // Step H: Reset the tiling object to its original position.
    pos[0] = sX;
    pos[1] = sY;
};


```

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 `Math.ceil()` function ensure that the computed nx and ny are integers.
 - c. Step D saves the original position of the Renderable object before offsetting and drawing it. Step E offsets the Renderable object to cover the lower-left corner of the camera WC bounds.
 - d. Step F computes the number of repeats required, and step G tiles the Renderable object in the positive x and y directions until the results cover the entire camera WC bounds.
 - e. Step H resets the position of the tiled object to the original location.
4. Override the `draw()` function to call the `_drawTile()` function when tiling is true.

```

TiledGameObject.prototype.draw = function (aCamera) {
    if (this.isVisible()) {
        if (this.shouldTile()) {
            // find out where we should be drawing
            this._drawTile(aCamera);
        } else {
            this.mRenderComponent.draw(aCamera);
        }
    };
};


```

Modify MyGame to Test Tiled Objects

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 newly added camera is focused on one of the Hero objects.

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 `MyGame initialize()` function is examined in detail here. The rest of the `MyGame` functions are largely similar to previous projects and are not listed here to avoid unnecessary distraction.

```
MyGame.prototype.initialize = function () {
    // Step A: set up the cameras
    this.mHeroCam = new Camera(
        vec2.fromValues(20, 30.5), // position of the camera
        14,                      // width of camera
        [0, 420, 300, 300],      // viewport (orgX, orgY, width, height)
        2
    );
    this.mHeroCam.setBackgroundColor([0.5, 0.5, 0.9, 1]);

    this.mCamera = new 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

    // Step B: the lights
    this._initializeLights(); // defined in MyGame_Lights.js

    // Step C: the far Background
    var bgR = new IllumRenderable(this.kBg, this.kBgNormal);
    bgR.setElementPixelPositions(0, 1024, 0, 1024);
    bgR.getXform().setSize(30, 30);
    bgR.getXform().setPosition(0, 0);
    bgR.getMaterial().setSpecular([0.2, 0.1, 0.1, 1]);
    bgR.getMaterial().setShininess(50);
    bgR.getXform().setZPos(-20);
    bgR.addLight(this.mGlobalLightSet.getLightAt(1)); // only directional light
    this.mBg = new TiledGameObject(bgR);

    // Step D: the closer Background
    var i;
    var bgR1 = new IllumRenderable(this.kBgLayer, this.kBgLayerNormal);
    bgR1.getXform().setSize(30, 30);
    bgR1.getXform().setPosition(0, 0);
    bgR1.getXform().setZPos(-10);
    for (i = 0; i < 4; i++)
        bgR1.addLight(this.mGlobalLightSet.getLightAt(i)); // all the lights
    bgR1.getMaterial().setSpecular([0.2, 0.2, 0.5, 1]);
```

```

bgR1.getMaterial().setShininess(10);
this.mBgl1 = new TiledGameObject(bgR1);
this.mBgl1.setSpeed(0.1);
this.mBgl1.setCurrentFrontDir([-1, 0]);

// Initialize the other objects in the scene
//
// ... code not shown because of similarity to previous projects ...
//
};

}

```

In the previous code, the two cameras are first created in step A, followed by the creation and initialization of all the light sources (in `MyGame_Lights.js`, not shown because of similarity to previous projects). Step C defines `bgR` as an `IllumRenderable` object that is being illuminated by one light source and creates a `TiledGameObject` instance based on `bgR`. Step D defines the second `IllumRenderable` object that is being illuminated by four light sources and again creates a `TiledGameObject` instance based on the `Renderable` object. Notice that the second tile object initializes its speed to 0.1 and front direction to point toward the negative x-axis. This object will move toward the left continuously. Since the `TiledGameObject` `mShouldTile` variable defaults to true, both of the tile objects will tile the camera that they are drawing to.

You can now run the project and move the `Hero` object with the WASD keys. As expected, the two layers of tiled backgrounds are clearly visible with the front layer moving continuously toward the left. 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.

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 10-5 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 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 (for example, 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.

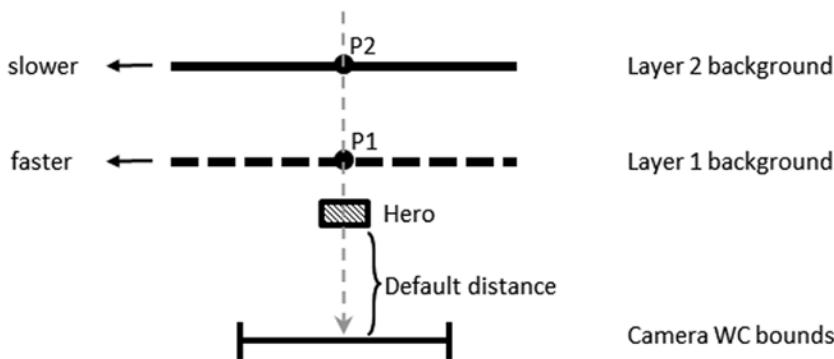


Figure 10-5. Top view of a scene with two background objects at different distances

Figure 10-6 shows the results of leftward parallax scrolling with a stationary camera. With Layer1 scrolling at a faster speed than Layer2, position P_1 has a greater displacement than P_2 from their original positions. A continuous scrolling will move Layer1 faster than Layer2 and properly convey the sense that it is closer than Layer2. In parallax scrolling, objects that are closer to the camera always have a greater scroll speed than objects that are farther.

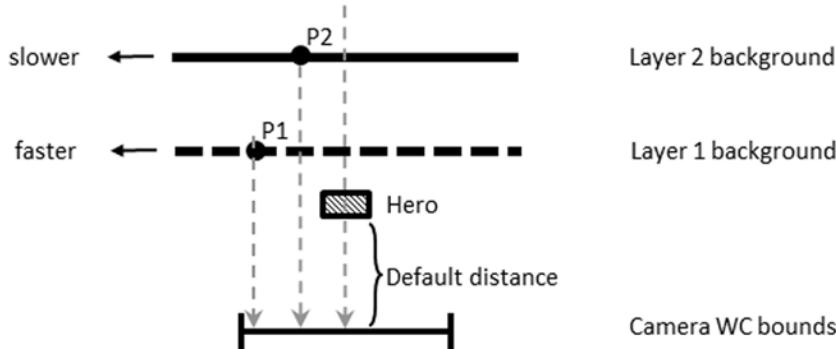


Figure 10-6. Top view of parallax scrolling with stationary camera

In the case when the camera is in motion, relative speeds of objects must be considered when implementing parallax scrolling. Figure 10-7 illustrates, with a top view, the situation of a moving camera with stationary objects. In this example, the camera WC bounds have moved rightward by d units. Since the movement is in the camera, all stationary objects in the camera view will appear to have been displaced by the inverse of the camera movement. For example, the stationary Hero object is displaced from the center leftward to the left edge of the new WC bounds. To properly simulate motion parallax, the two backgrounds, Layer1 and Layer2, must be displaced by different relative distances. In this case, relative distances must be computed such that farther objects will appear to move slower. At the end of the camera movement, in the new WC bounds, the Hero object that is closest to the camera will appear to have been displaced leftward by d units, the Layer1 object by $0.25d$, and the Layer2 object by $0.75d$. In this way, the displacements of the objects reflect their relative distances from the camera. To achieve this, the translation of the Hero object is zero, and the Layer1 and Layer2 objects must be translated rightward by $0.25d$ and $0.75d$, respectively. Notice that the backgrounds are translated rightward by amounts that are less than that of the camera movement, and as a result the backgrounds are actually moving leftward. For example, although the Layer1 object is translated rightward by $0.25d$, when viewed from the camera that has been moved rightwards by d , the resulting relative movement is such that the Layer1 object has been displaced leftward by $0.75d$.

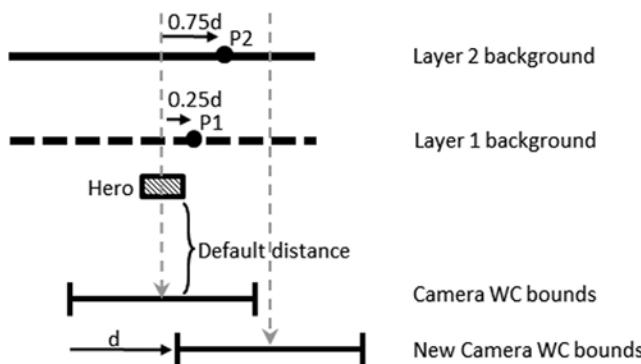


Figure 10-7. Top view of parallax scrolling with the camera in motion

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 implementation. 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 ParallaxObjects Project

This project demonstrates parallax scrolling. You can see an example of this project running in Figure 10-8. The source code to this project is defined in the Chapter10/10.2.ParallaxObjects folder.



Figure 10-8. Running the Parallax Objects project

The controls of the project are as follows:

- *P key:* Toggles the drawing of parallax camera to provide a zoomed view of object parallax
- *WASD keys:* Move the Dye character (the hero) to pan the WC window bounds

The goals of the project are as follows:

- To understand and appreciate motion parallax
- To simulate motion parallax with parallax scrolling

Define ParallaxGameObject to Implement Parallax Scrolling

Parallax scrolling involves the continuous scrolling of objects, and `TiledGameObject` provides a convenient platform for never-ending scrolling. For this reason, `ParallaxGameObject` is defined as a subclass to the `TiledGameObject` class.

1. Create a new file in the `src/Engine/GameObjects/` folder and name it `ParallaxGameObject.js`. Add the following code to construct the object:

```
function ParallaxGameObject(renderableObj, scale, aCamera) {
    this.mRefCamera = aCamera;
    this.mCameraWCCenterRef = vec2.clone(this.mRefCamera.getWCCenter());
    this.mParallaxScale = 1;
    this.setParallaxScale(scale);
    TiledGameObject.call(this, renderableObj);
}
gEngine.Core.inheritPrototype(ParallaxGameObject, TiledGameObject);
```

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.

2. Define the getter and setter functions for `mParallaxScale`, and notice that `mParallaxScale` is the inverse of the object distance.

```
ParallaxGameObject.prototype.getParallaxScale = function () {
    return this.mParallaxScale;
};

ParallaxGameObject.prototype.setParallaxScale = function(s) {
    if (s <= 0) {
        this.mParallaxScale = 1;
    } else {
        this.mParallaxScale = 1/s;
    }
};
```

3. Override the `update()` function to implement parallax scrolling.

```
ParallaxGameObject.prototype.update = function () {
    // simple default behavior
    this._refPosUpdate(); // check to see if the camera has moved
    var pos = this.getXform().getPosition(); // our own xform
    vec2.scaleAndAdd(pos, pos, this.getCurrentFrontDir(),
                     this.getSpeed() * this.mParallaxScale);
};
```

The `_refPosUpdate()` function is the one that computes a relative displacement based on the reference camera's WC center position. The `vec2.scaleAndAdd()` function moves the current object at a speed that is scaled by the `mParallaxScale`.

4. Define the `_refPosUpdate()` function.

```
ParallaxGameObject.prototype._refPosUpdate = function () {
    // now check for reference movement
    var deltaT = vec2.fromValues(0, 0);
    vec2.sub(deltaT, this.mCameraWCCenterRef, this.mRefCamera.getWCCenter());
    this.setWCTranslationBy(deltaT);
    // now update WC center ref position
    vec2.sub(this.mCameraWCCenterRef, this.mCameraWCCenterRef, deltaT);
};
```

The `deltaT` variable records the relative displacements, and `setWCTranslationBy()` moves the object to simulate parallax scrolling.

5. Define the function to translate the object to implement parallax scrolling.

```
ParallaxGameObject.prototype.setWCTranslationBy = function (delta) {
    var f = (1-this.mParallaxScale);
    this.getXform().incXPosBy(-delta[0] * f);
    this.getXform().incYPosBy(-delta[1] * f);
};
```

Testing ParallaxGameObject in MyGame

The testing of `ParallaxGameObject` involves testing for the correctness of parallax scrolling with the stationary camera, testing for the motion camera with an object in front of and behind the default distance, and observing the `ParallaxGameObject` from an alternative camera. The `MyGame`-level source code is largely similar to that from the previous project, and the details are not listed. The relevant part of the `initialize()` function is listed for the purpose of demonstrating how to create the `ParallaxGameObject` instances.

```
MyGame.prototype.initialize = function () {
    // Step A: set up the cameras
    // creating the two cameras and initializing the lights
    //
    // ... code not shown because of similarity to previous projects ...
    //

    // Step C: the far Background
    var bgR = new IllumRenderable(this.kBg, this.kBgNormal);
    bgR.setElementPixelPositions(0, 1024, 0, 1024);
    bgR.getXform().setSize(30, 30);
    bgR.getXform().setPosition(0, 0);
    bgR.getMaterial().setSpecular([0.2, 0.1, 0.1, 1]);
    bgR.getMaterial().setShininess(50);
    bgR.getXform().setZPos(-10);
```

```

bgR.addLight(this.mGlobalLightSet.getLightAt(1)); // only the directional light
this.mBg = new ParallaxGameObject(bgR, 5, this.mCamera);
this.mBg.setCurrentFrontDir([0, -1, 0]);
this.mBg.setSpeed(0.1);

// Step D: the closer Background
var i;
var bgR1 = new IllumRenderable(this.kBgLayer, this.kBgLayerNormal);
bgR1.getXform().setSize(25, 25);
bgR1.getXform().setPosition(0, 0);
bgR1.getXform().setZPos(0);
bgR1.addLight(this.mGlobalLightSet.getLightAt(1)); // the directional light
bgR1.addLight(this.mGlobalLightSet.getLightAt(2)); // the hero spotlight light
bgR1.addLight(this.mGlobalLightSet.getLightAt(3)); // the hero spotlight light
bgR1.getMaterial().setSpecular([0.2, 0.2, 0.5, 1]);
bgR1.getMaterial().setShininess(10);
this.mBgl1 = new ParallaxGameObject(bgR1, 3, this.mCamera);
this.mBgl1.setCurrentFrontDir([0, -1, 0]);
this.mBgl1.setSpeed(0.1);

// Step E: the front layer
var f = new TextureRenderable(this.kBgLayer);
f.getXform().setSize(30, 30);
f.getXform().setPosition(0, 0);
this.mFront = new ParallaxGameObject(f, 0.9, this.mCamera);

// ... Identical to previous project ...
};

}

```

The `mBg` object is created as a `ParallaxGameObject` with a scale value 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, and it signifies the object distance from the camera, with values greater than 1 being farther from the default distance and less than 1 being closer than the default distance. In this case, `mBg` is the furthest and `mBgl1` is closer, and both are behind the default distance. The `mFront` object is actually closer to the camera than the default distance. It is in front of the `Hero` object. Notice that `mBg` and `mBgl1` are both defined to be continuously scrolling downward (`setCurrentFrontDir([0, -1, 0])`) with a speed of 0.1.

You can now run the project and observe the darker foreground layer partially blocking the `Hero` and `Minions` objects. You can see the two background layers continuously scrolling downward but with different speeds. The `mBg` object is farther away and thus scrolls slower than the `mBgl1` object. Move the `Hero` object with WASD and pan the camera. You will 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 animated `Minion` objects.

Press the P key to enable the drawing of the second camera. Notice that the view in this camera is as expected. The `Hero` object is stationary, and the two background layers scroll continuously downward. 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 designer, it is important to ensure this side effect does not cause player confusion.

Layer Management

Although the engine you are developing is for supporting 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 defined and drawn in the order of their depth ordering. It is convenient for the game engine to provide a utility manager to help game programmers manage and work with the depth layering. A typical 2D game can have the following layers, in the order of the distance from the camera, from nearest to furthest:

- *Heads-up display (HUD) layer*: Typically closest to the camera displaying essential user interface information
- *Foreground or front layer*: The layer in front of the game objects for decorative or partial occlusion of the game objects
- *Actor layer*: The default distance layer in Figure 10-5, where all game objects reside
- *Shadow Receiver layer*: The layer behind the actor layer to receive potential shadows
- *Background layer*: The decorative background

Each layer will reference all objects defined for that layer, and these objects will be drawn in the order they were inserted into the layer, with the last inserted drawn last and covering objects before it. This section presents the `LayerManager` engine component to support the described five layers to relieve game programmers from the details of managing updates and drawings the objects. Note that the number of layers a game engine should support is determined by the kinds of games that the engine is designed to build. The five layers presented are logical and convenient for simple games. You may choose to expand the number of layers in your own game engine.

The Layer Manager Project

This project demonstrates how to develop a utility component to assist in managing layers for the game programmers. You can see an example of this project running in Figure 10-9. The source code to this project is defined in the [Chapter10/10.3.LayerManager](#) folder.

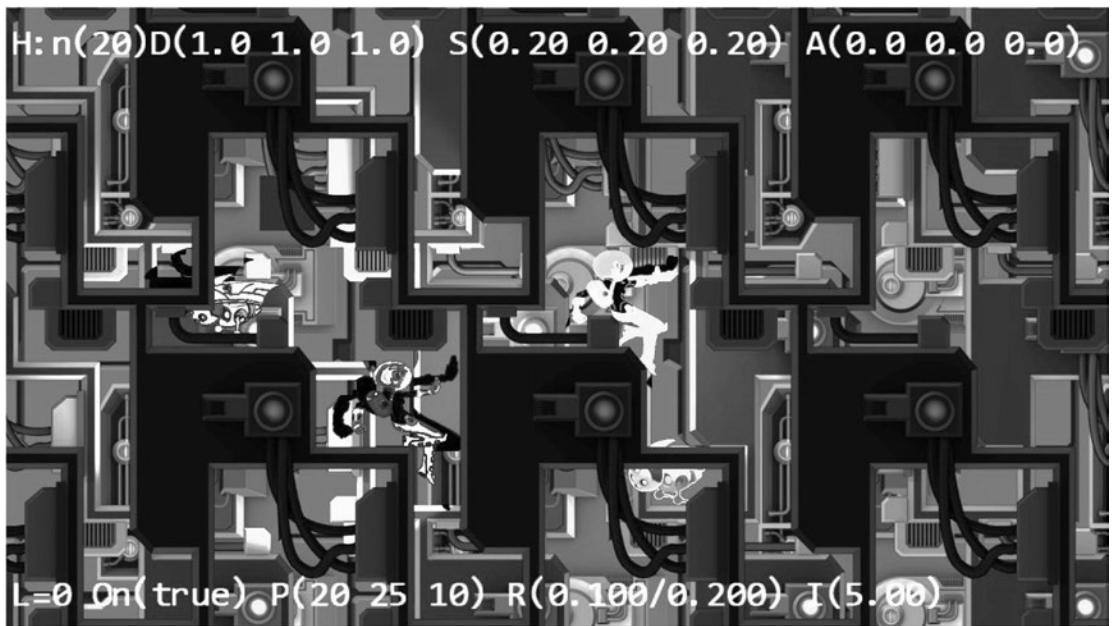


Figure 10-9. Running the Layer Manager project

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

- *P key:* Toggles the drawing of parallax camera, to provide a zoomed view of object parallax
- *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 engine components.

1. Create a new file in the `src/Engine/Core/` folder and name it `Engine_LayerManager.js`. This file will implement the `LayerManager` engine component.
2. Define enumerators for the layers.

```
gEngine.eLayer = Object.freeze({
    eBackground: 0,
    eShadowReceiver: 1,
    eActors: 2,
    eFront: 3,
    eHUD: 4
});
```

3. Define the `LayerManager` component following the JavaScript Module Pattern as follows:

```
var gEngine = gEngine || { };
// initialize the variable while ensuring it is not redefined

gEngine.LayerManager = (function () {
    var mPublic = { };
    return mPublic;
}() );
```

4. Within the `LayerManager` component, define appropriate constants and instance variables to keep track of the layers.

```
var kNumLayers = 5;
var mAllLayers = [ ];
```

`mAllLayers` is an array of `GameObjectSet` instances representing each of the five layers.

5. Define an `initialize()` function to create the array of `GameObjectSet` instances.

```
var initialize = function() {
    mAllLayers[gEngine.eLayer.eBackground] = new GameObjectSet();
    mAllLayers[gEngine.eLayer.eShadowReceiver] = new GameObjectSet();
    mAllLayers[gEngine.eLayer.eActors] = new GameObjectSet();
    mAllLayers[gEngine.eLayer.eFront] = new GameObjectSet();
    mAllLayers[gEngine.eLayer.eHUD] = new GameObjectSet();
};
```

6. Define a `cleanUp()` function to reset the `mAllLayer` array.

```
var cleanUp = function() {
    initialize();
};
```

7. Define functions to add to, remove from, and query the layers.

```
var addToLayer = function(layerEnum, obj) {
    mAllLayers[layerEnum].addToSet(obj);
};

var removeFromLayer = function(layerEnum, obj) {
    mAllLayers[layerEnum].removeFromSet(obj);
};

var layerSize = function(layerEnum) {
    return mAllLayers[layerEnum].size();
};
```

```
var addAsShadowCaster = function(obj) {
    var i;
    for (i = 0; i < mAllLayers[gEngine.eLayer.eShadowReceiver].size(); i++)
        mAllLayers[gEngine.eLayer.eShadowReceiver].getObjAt(i).
            addShadowCaster(obj);
};
```

Note the `addAsShadowCaster()` function assumes that the shadow receiver objects are already inserted into the `eShadowReceiver` layer. Additionally, this function adds the casting object to all receivers in the layer.

8. Define functions to draw a specific layer or all the layers, from the furthest to the nearest to the camera.

```
var drawLayer = function(layerEnum, aCamera) {
    mAllLayers[layerEnum].draw(aCamera);
};

var drawAllLayers = function(aCamera) {
    var i;
    for (i=0; i < kNumLayers; i++)
        mAllLayers[i].draw(aCamera);
};
```

9. Define a function to move a specific object such that it will be drawn last (on top).

```
var moveToLayerFront = function(layerEnum, obj) {
    mAllLayers[layerEnum].moveToLast(obj);
};
```

10. Define functions to update a specific layer or all the layers.

```
var updateLayer = function(layerEnum) {
    mAllLayers[layerEnum].update();
};

var updateAllLayers = function() {
    var i;
    for (i=0; i < kNumLayers; i++)
        mAllLayers[i].update();
};
```

11. Remember to add all the functions to the public interface.

```
var mPublic = {
    initialize: initialize,
    drawAllLayers: drawAllLayers,
    updateAllLayers: updateAllLayers,
    cleanUp: cleanUp,
```

```

        drawLayer: drawLayer,
        updateLayer: updateLayer,
        addToLayer: addToLayer,
        addAsShadowCaster: addAsShadowCaster,
        removeFromLayer: removeFromLayer,
        moveToLayerFront: moveToLayerFront,
        layerSize: layerSize
    };
    return mPublic;
}

```

Modify Engine Components and Objects

You must modify the rest of the game engine slightly to integrate the new LayerManager component.

Enhance the GameObjectSet Functionality

Add the following two functions to support removing membership and moving objects to the end of a set array:

```

GameObjectSet.prototype.removeFromSet = function (obj) {
    var index = this.mSet.indexOf(obj);
    if (index > -1)
        this.mSet.splice(index, 1);
};

// Ensures the given obj is the last element (when drawn, appears on top)
GameObjectSet.prototype.moveToLast = function (obj) {
    this.removeFromSet(obj);
    this.addSet(obj);
};

```

Initialize LayerManager in gEngine_Core

Modify the engine's Core component to initialize the LayerManager component in the `initializeEngineCore()` function.

```

// initialize all of the EngineCore components
var initializeEngineCore = function (htmlCanvasID, myGame) {
    _initializeWebGL(htmlCanvasID);
    gEngine.VertexBuffer.initialize();
    gEngine.Input.initialize(htmlCanvasID);
    gEngine.AudioClips.initAudioContext();
    gEngine.Physics.initialize();
    gEngine.LayerManager.initialize();

    // Inits DefaultResources, when done,
    // invoke the anonymous function to call startScene(myGame).
    gEngine.DefaultResources.initialize(function () { startScene(myGame); });
};

```

Define the Update Function for Layer Membership Objects

Define update functions for objects that may appear as members in the LayerManager layers: Renderable, FontRenderable, and ShadowReceiver.

Modify MyGame to Work with LayerManager

The MyGame level implements the same functionality as in the previous project. The only difference is the delegation of layer management to the LayerManager component. The following description focuses only on how the LayerManager functions are called.

1. Modify the `unloadScene()` function to clean up the LayerManager.

```
MyGame.prototype.unloadScene = function () {
    gEngine.LayerManager.cleanUp();

    gEngine.Textures.unloadTexture(this.kMinionSprite);
    gEngine.Textures.unloadTexture(this.kBg);
    gEngine.Textures.unloadTexture(this.kBgNormal);
    gEngine.Textures.unloadTexture(this.kBgLayer);
    gEngine.Textures.unloadTexture(this.kBgLayerNormal);
    gEngine.Textures.unloadTexture(this.kMinionSpriteNormal);
};
```

2. Modify the `initialize()` function to add the game objects to the corresponding layers in the LayerManager component.

```
MyGame.prototype.initialize = function () {
    // ... Identical to previous project ...

    // add to layer managers ...
    gEngine.LayerManager.addToLayer(gEngine.eLayer.eBackground, this.mBg);
    gEngine.LayerManager.addToLayer(gEngine.eLayer.eShadowReceiver, this.mBgShadow1);

    gEngine.LayerManager.addToLayer(gEngine.eLayer.eActors, this.mIllumMinion);
    gEngine.LayerManager.addToLayer(gEngine.eLayer.eActors, this.mLgtMinion);
    gEngine.LayerManager.addToLayer(gEngine.eLayer.eActors, this.mIllumHero);
    gEngine.LayerManager.addToLayer(gEngine.eLayer.eActors, this.mLgtHero);

    gEngine.LayerManager.addToLayer(gEngine.eLayer.eFront, this.mBlock1);
    gEngine.LayerManager.addToLayer(gEngine.eLayer.eFront, this.mBlock2);
    gEngine.LayerManager.addToLayer(gEngine.eLayer.eFront, this.mFront);
    gEngine.LayerManager.addToLayer(gEngine.eLayer.eHUD, this.mMsg);
    gEngine.LayerManager.addToLayer(gEngine.eLayer.eHUD, this.mMatMsg);
};
```

3. Modify the `draw()` function to rely on the `LayerManager` component for the actual drawings.

```
MyGame.prototype.draw = function () {
    // Step A: clear the canvas
    gEngine.Core.clearCanvas([0.9, 0.9, 0.9, 1.0]); // clear to light gray

    this.mCamera.setupViewProjection();
    gEngine.LayerManager.drawAllLayers(this.mCamera);

    if (this.mShowHeroCam) {
        this.mParallaxCam.setupViewProjection();
        gEngine.LayerManager.drawAllLayers(this.mParallaxCam);
    }
};
```

4. Modify the `update()` function to rely on the `LayerManager` component for the actual update of all game objects.

```
MyGame.prototype.update = function () {
    this.mCamera.update(); // to ensure proper interpolated movement effects
    this.mParallaxCam.update();

gEngine.LayerManager.updateAllLayers();

    // ... Identical to previous project ...
};
```

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 `LayerManager` layers during `initialize()`, the `draw()` and `update()` functions of a game level can be much cleaner. The simpler and cleaner `MyGame update()` function is of special importance. Instead of being crowded with mundane game object `update()` function calls, the `MyGame update()` function can now focus on implementing the game logic and controlling the interactions between game objects.

Summary

This chapter explained the need for tiling and introduced the `TileGameObject` to implement a simple algorithm that tiles and covers a given camera WC bounds. The basics of parallax and approaches to simulate motion parallax with parallax scrolling were introduced. Motion parallax with stationary and motioned camera were examined, and solutions were derived and implemented. You learned that computing movements relative to the camera motions to displace background objects results in visually pleasing motion parallax but may cause player confusion when viewed from different cameras. With shadow computations introduced earlier and now parallax scrolling, game programmers must dedicate code and attention to coordinate the drawing order of different types of objects. To facilitate the programmability of the game engine, the `LayerManager` engine component is presented as a utility tool to alleviate game programmers from managing the drawing of the layers.

Your game engine is now completed, and it can draw objects with texture maps, sprite animations, and even illumination by various light sources. The engine defines proper abstractions for simple behaviors, mechanisms to approximate and accurately compute the collisions, and simulates the physical behaviors of these objects. Views from multiple cameras can conveniently be displayed over the same game screens with manipulation functionality that is smoothly interpolated. Keyboard/mouse input is supported, and now background objects can scroll without bounds and simulate motion parallax.

The important next step is to go through a simple game design process and implement a game based on your new completed game engine.

Game Design Considerations

In previous “Game Design Considerations” sections, you’ve explored how developing one simple mechanic from the ground up can lead in many directions and be applied to a variety of game types. Creative teams in game design studios frequently debate which elements of game design take the lead in the creative process. Writers often believe story comes first, while many designers believe that story and everything else must be secondary to gameplay. There’s no right or wrong answer, of course. The creative process is a chaotic system, and every team and studio is unique; some creative directors want to tell a particular story and search for mechanics and genres that are best suited to supporting specific narratives, 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. If you’re creating an AAA first-person-shooter experience, for example, consumers will have specific expectations for many of the core elements of play, and it’s usually a smart move to ensure that gameplay drives the design. If you’re creating an indie adventure game designed to provide players with new experiences and unexpected twists, however, story and setting might lead the way.

Many game designers (and this includes seasoned veterans as well as those new to the discipline) begin new projects by designing complex experiences that are relatively minor variations on existing well-understood mechanics; and while there are sound reasons for this approach (as in the case of AAA studios developing content for particularly demanding audiences or a desire to work with mechanics that have proven to be successful across many titles), it tends to significantly limit exploration into new territory and is one reason why many gamers complain about creative stagnation and a lack of gameplay diversity between games within the same genre. Many of us who are professional game designers grew up enjoying certain kinds of games and dreamed about creating new experiences based on the mechanics we know and love, and several decades of that culture has solidified much of the industry around a comparatively few number of similar mechanics and conventions. The good news is that a rapidly growing independent and small studio community has boldly begun throwing long-standing genre convention to the wind in recent years, and new distribution platforms like mobile app stores and Valve’s Steam have opened opportunities for a wide range of never-before-seen mechanics and experiences to flourish.

If you continue exploring game design, you’ll realize there are relatively few completely unique core mechanics, but there are endless opportunities for innovating as you intentionally and logically build those elemental interactions into more complex causal chains, adding unique flavor and texture through elegant integration with the other elements of game design. Some of the most groundbreaking games in recent memory were created through exercises very much like the mechanic exploration you’ve done in the “Game Design Considerations” sections. Valve’s Portal, for example, is based on the same kind of “escape the room” sandbox you explored and is designed around a similarly simple base mechanic. So, what made Portal such a breakthrough hit? While many things need to come together to create a hit game, Portal undoubtedly benefitted from a design team that started building the experience from the most basic mechanic and smartly increased complexity as they became increasingly fluent in its unique structure and characteristics, instead of starting at the 10,000-foot level with a codified genre and a host of expectations and overly familiar mechanics.

Of course, nobody talks about Portal without also mentioning the rogue artificial intelligence character GLaDOS and her Aperture Laboratories playground: setting, narrative, and audiovisual design are as important to the Portal experience as the portal-launching mechanic, and it's hard to separate gameplay from the narrative given how skillfully intertwined they are. The projects in this chapter provide a good opportunity to begin similarly situating the mechanic from the "Game Design Considerations" sections in a unique setting and context. You've probably noticed many of the projects throughout this book are building toward a sci-fi visual theme, with a spacesuit-wearing hero character, a variety of flying robots, and now in Chapter 10 the introduction of large-scale parallaxing environments. And while you're not building a game with the same degree of environment and interaction complexity as Portal, that doesn't mean you don't have the same opportunity to develop a highly engaging game setting, context, and cast of characters.

The first thing you should notice about the Tiled Objects project is the dramatic impact on environment experience and scale compared to earlier projects. The factors enhancing presence in this project are the three independently moving layers (hero, moving wall, and stationary wall) and the seamless tiling of the two background layers. Compare the Tiled Objects project to the Shadow Shaders project from Chapter 8; notice the difference in presence when the environment is broken into multiple layers that appear to move in an analogous (if not physically accurate) way to how you experience movement in the physical world. The sense of presence is further strengthened when you add multiple background layers of parallaxing movement in the Parallax Objects project; as you move through the physical world, the environment appears to move at different speeds, with closer objects seeming to pass-by quickly while objects toward the horizon appear to move slowly. Parallaxing environment objects simulate this effect, adding considerable depth and interest to game environments. The Layer Manager project pulls things together and begins to show the potential for a game setting to immediately engage the imaginations of players. With just a few techniques, you're able to create the impression of a massive environment that might be the interior of an ancient alien machine, the outside of a large space craft, or anything else you might care to create. Try using different kinds of image assets with this technique: exterior landscapes, underwater locations, abstract shapes, and the like would all be interesting to explore. You'll often find inspiration for game settings by experimenting with just a few basic elements, as you did in Chapter 10.

Pairing game environment design (both audio and visual) with interaction design (and occasionally the inclusion of haptic feedback-like controller vibrations) is the mechanism you 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 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. In Chapter 11, you'll pair and 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.

CHAPTER 11



Building a Sample Game: From Design to Completion

The projects included in the main sections of Chapters 1 to 10 began with simple shapes, slowly introducing characters and environments to illustrate the concepts in 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 for game mechanics. 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 mechanics. This chapter changes the focus to emphasize the design process from earliest conception through functional prototype, bringing together and extending the work done in earlier projects by using the characters and environments from prior chapters along with the basic idea for the “unlocking platformer” from the “Design Considerations” section of Chapter 10. 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 considering most of the nine elements of game design described in the *How Do You Make a Great Video Game?* Section of Chapter 1 while crafting the mechanic to stay focused on refining the core characteristics of play. The design approach used in this book is a ground-up framework that emphasizes first working with a pure mechanic prior to considering genre or setting; when you begin incorporating a setting and building out levels that include additional design elements, the mechanic will grow and evolve in unique directions as you grow the game world. There are endless potential variations for the mechanics you design, and you’ll be surprised by how differently the same elemental mechanic develops and evolves based on the kind of game you make.

Part 1: Refining the Concept

At this point, you should have the beginning of a concept using a 2D jumping and puzzle-solving mechanic to unlock a barrier and reach a reward. Recall Figure 11-1 as the final screen layout and design from Chapter 10.

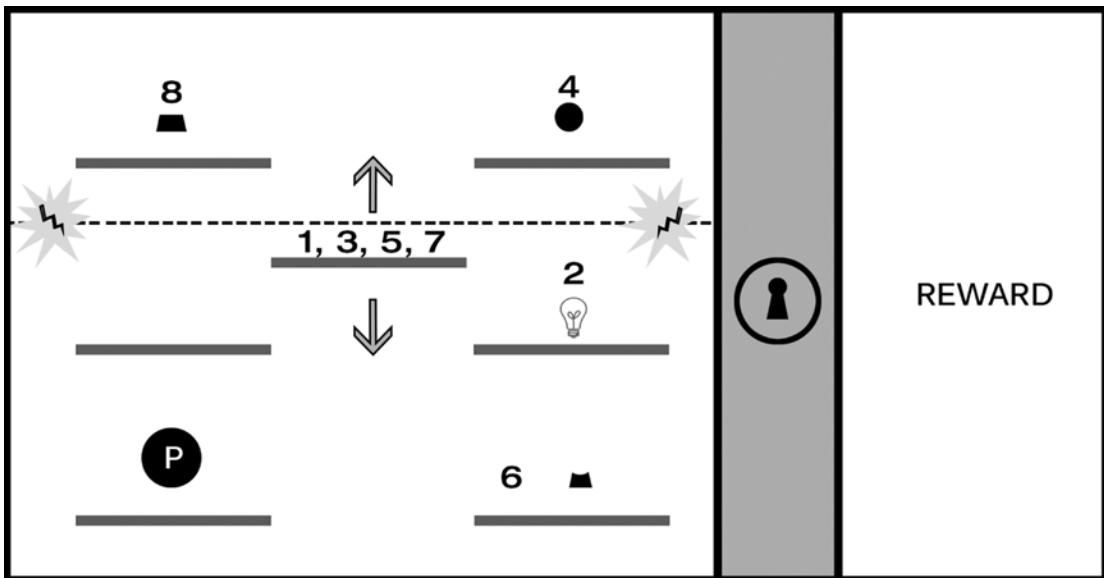


Figure 11-1. The 2D implementation from Chapter 10

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 11-1) on the moving elevator platform (#1 in Figure 11-1) and jump off to the middle platform of the right column before touching the energy field.
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 11-1).
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.
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.
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.
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 glow, signaling progress.
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.
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.

Writing out this sequence (or *game flow*) may seem unnecessary given the mock-up screens you've created for the mechanic. However, it's important 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 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 smartly refine the gameplay logic in a way that allows you to visualize the effect of each change on the flow of the level.

You could continue building out the mechanic to make the level more interesting and challenging (for example, you might include a timer on the off switch for the energy field requiring players to collide with all lock parts in rapid succession). 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 mechanic evolves from here.

Recall from Chapter 10 that the projects ended with a set of concept explorations supporting a sci-fi setting. Figure 11-2 shows a futuristic industrial environment design, a hero character wearing a space suit, and what appear to be flying robots.

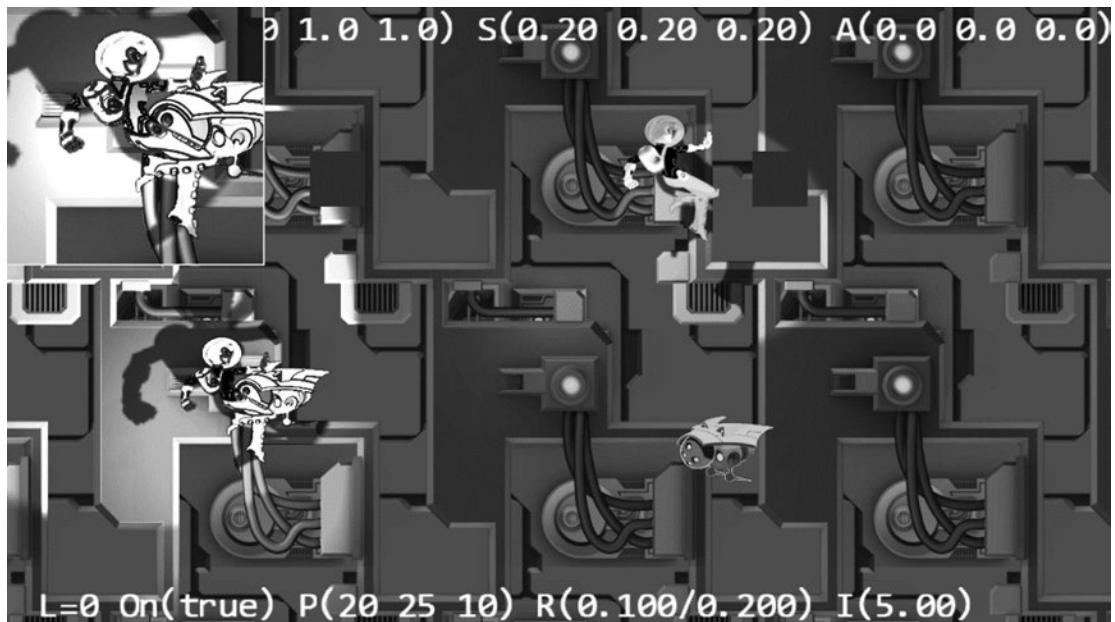


Figure 11-2. Concepts from Chapter 10

Note that there isn't anything specific about the game mechanic you've been creating that would necessarily lead in a sci-fi direction; game mechanics are abstract structures and can typically integrate with any kind of setting or visual style. In this case, the authors of this book enjoy settings that take place in space (particularly 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 mechanic from Chapter 10 be adapted to a jungle setting, a contemporary urban location, a medieval fantasy world, or an underwater metropolis?

Part 2: Integrating a Setting

It can be helpful at this stage of design to begin assigning some basic fictional background to the game. This will help to evolve and extend the mechanic in unique ways that integrate with and 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 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 mechanic driving this narrative. The design task at this stage includes brainstorming some fictional context that propels the player through the game. (Using the same few concept art assets already created, the hero could be participating in a race, looking for something that was lost, exploring an abandoned alien vessel, or any of a million other possibilities.)

Contextual Images Bring the Setting to Life

Now that you've described a basic fictional wrapper and mechanic ("Players must complete sequence-based levels to save their spaceship before it explodes"), swap just a few of the shapes from the mechanic design with some of the concept elements. Figure 11-3 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.

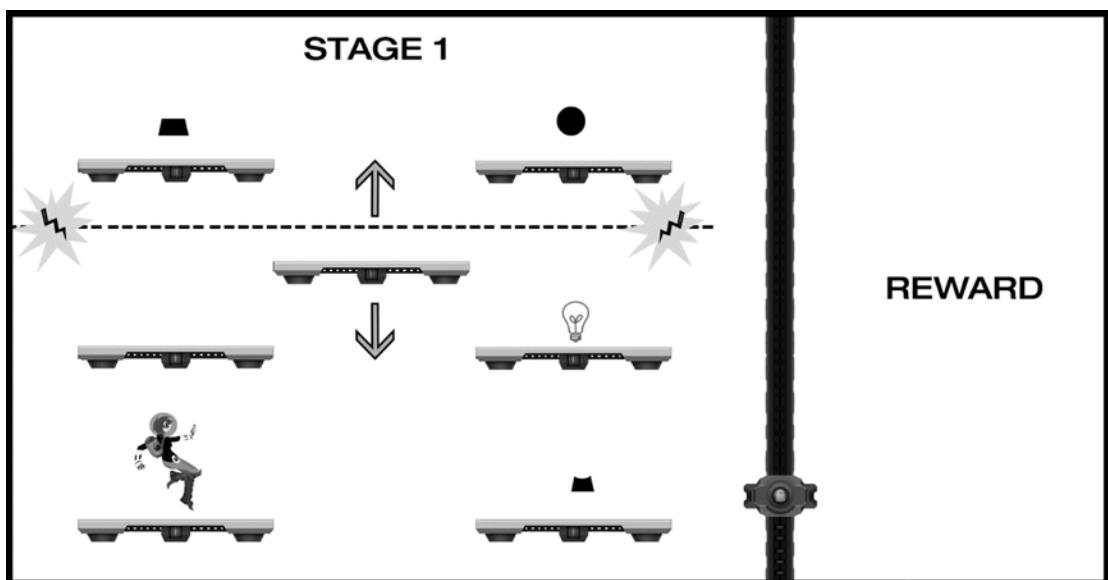


Figure 11-3. The introduction of several visual design elements supporting the game setting and nascent fiction

Although you've made only a few minor substitutions and don't yet have the visual elements anchored in an environment, Figure 11-3 conveys quite a bit more context and contributes significantly more to presence than the abstract shapes of Figure 11-1. The hero character now suggests a scale that will naturally be contextualized as players benchmark relative sizes against the human figure, which brings the relative size of the entire game environment into focus for players. The implementation of object physics for the hero character as described in Chapter 10 also becomes an important component of play: simulated gravity,

momentum, and the like connect players viscerally to the hero character. This becomes an extension of their own physical bodies. (If you've ever seen gamers lean in when virtually jumping or running during gameplay, it's this extension of physical presence through a virtually controlled object that's responsible.) By implementing the design as described in Figure 11-3, you've already accomplished some impressive cognitive feats that support presence simply by adding a few visual elements and some object physics.

Defining the Playable Space

At this point in the design process you've sufficiently described the mechanic and setting to begin expanding the single screen into a full-level concept. It's not critical at this stage to have a final visual style defined, but including some concept art will help guide how the level grows. (Figure 11-3 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 11-3 in a working prototype to begin getting a sense for how movement feels (for example, the speed the hero character runs, the height the hero can jump, and so on), the scale of objects in the environment and the zoom level of the camera, and the like. There's no need to include interaction behaviors such as the lock components or the energy field at this stage because you haven't yet designed the level; you're still working with the abstract schematic layout for the sequencing mechanic, and the next set of tasks includes laying out the full level and tuning all the interactions. At this stage you're experimenting with basic hero character movement, object placement, and collision.

The current state of the design in Figure 11-3 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, extend the horizontal game space to include a more playable area and provide additional room for the character to maneuver, as shown in Figure 11-4.

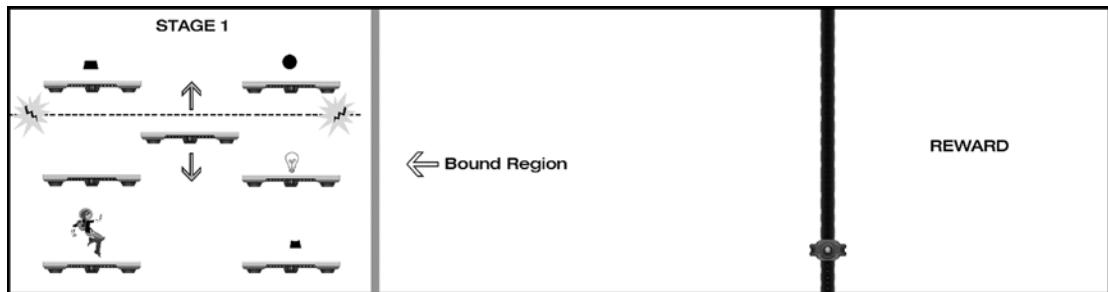


Figure 11-4. The level design grows to include an additional playable area

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. 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 will drive game progression.

Adding Layout to the Playable Space

It's now time to begin laying out the level to make good use of the additional space. There's no need to change the gameplay at this point; you just need to work with the new dimensions of the game screen. Figure 11-5 includes some additional platforms placed with no particular methodology other than ensuring players can successfully reach each platform.

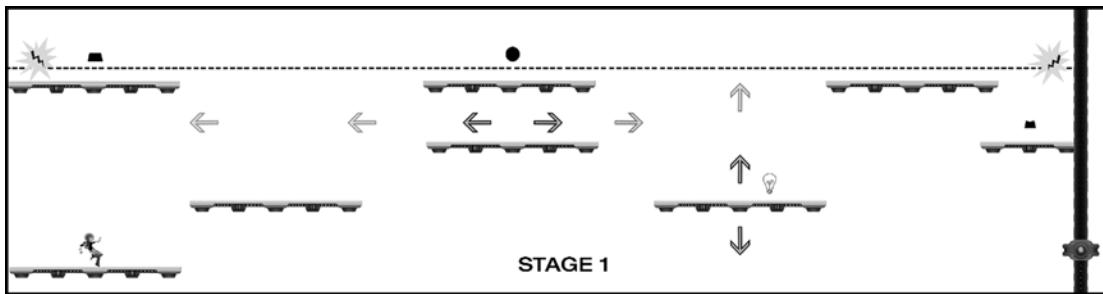


Figure 11-5. Expanding the layout to use the additional screen real estate, the diagram represents the entire length of stage 1 with the player able to see approximately 50 percent of the full level at any time. The camera moves forward or backward as the player moves the hero character toward the screen bound regions (for the moving platforms, darker arrows represent direction, an lighter arrows represent the range of movement)

Now that the level has some additional space to work with, there are several factors to evaluate and tune. The scale of the hero character in Figure 11-5 has been slightly reduced, for example, to increase the number of vertical jumps that can be executed on a single screen. Note that at this point you also have the opportunity to include additional vertical gameplay in the design if desired, implementing the same mechanism used to move the camera to the left and right; many 2D platformer games allow players to move through the game world both horizontally and vertically. This level prototype will limit movement to the x plane (left and right), although you can easily extend the level design to include vertical play in future iterations and/or subsequent levels.

As you're placing platforms in the level, you will again want to minimize design complexity while blocking out the game flow. Figure 11-5 adds one additional design element, a platform that moves left to right. Try to list the detailed sequence required to activate the three lock sections in Figure 11-5 using the same numbering methodology shown in Figure 11-1. When you're finished mapping out the sequence, compare it with Figure 11-6.

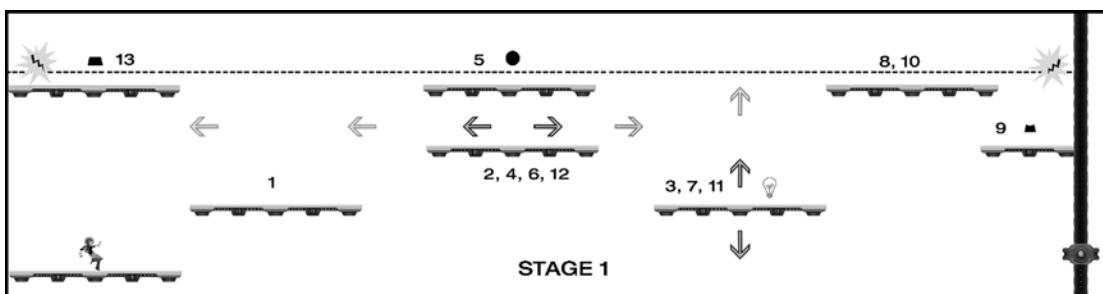


Figure 11-6. The most efficient sequence to unlock the barrier

Did your sequence match Figure 11-6, 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).

Tuning the Challenge and Adding Fun

Design aside: this stage of design is when the puzzle-making process really begins to open up; Figure 11-6 shows the potential to create highly engaging gameplay with only the few basic elements you've been working with. The authors' studio uses 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; 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 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.

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). The second factor is less immediately evident but equally challenging: only a portion of the level is visible at any time, so the player is not able to easily create a mental model of the entire level sequence like they can when the entire layout is visible on a single screen. It's important for designers to understand both explicit challenges (such as requiring players to time jumps between two moving platforms) but also less obvious (and often unintentional) challenges such as being able to see only part of the level at any given time. Think back to a game you've played where it felt like the designers expected you remember too many elements; that kind of frustration is often the result of unintentional challenges overburdening what the player can reasonably hold in short-term memory.

As a designer, you need to be aware of unintentional challenges and areas of unintentional frustration or difficulty. This is a key reason it's vital to observe people playing your game as early and often as possible (as a general rule, any time you're 100 percent certain you've designed something that makes perfect sense, at least half the people who play your game will tell you exactly the opposite). Although a detailed discussion of the benefits of user testing is outside the scope of this book, you should plan to observe people playing your game from the earliest proof-of-concept all the way to final release. There is no substitute for the insights you'll gain from watching different people play what you've designed.

The level currently assumes the hero character can rest only on platforms; although there's no design plan for what happens if the character misses a jump and falls to the bottom of the screen, players might reasonably imagine that it would result in a loss condition and trigger a game reset. If you added a “floor” to the level, gameplay would noticeably change. In addition to removing a significant risk, players would be able to access the elevator platform directly, as shown in Figure 11-7.

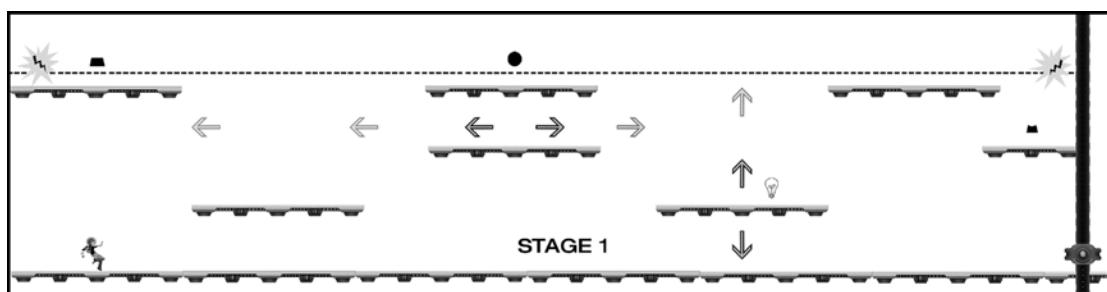


Figure 11-7. The addition of a “floor” to the game world significantly changes the level challenge

Further Tuning: Introducing Enemies

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 colliding with the energy field. The addition of the floor removes the falling risk and decreases the challenge of the level, but you might decide that the floor encourages players to more freely explore and pay closer attention to the environment. You're becoming increasingly conversant with the gameplay and flow for this mechanic and layout, so it's time to introduce a new element: attacking enemies (you can't let those robot designs from previous chapters go to waste)! Figure 11-8 introduces two basic enemy robot types: one that fires a projectile and one that simply patrols.

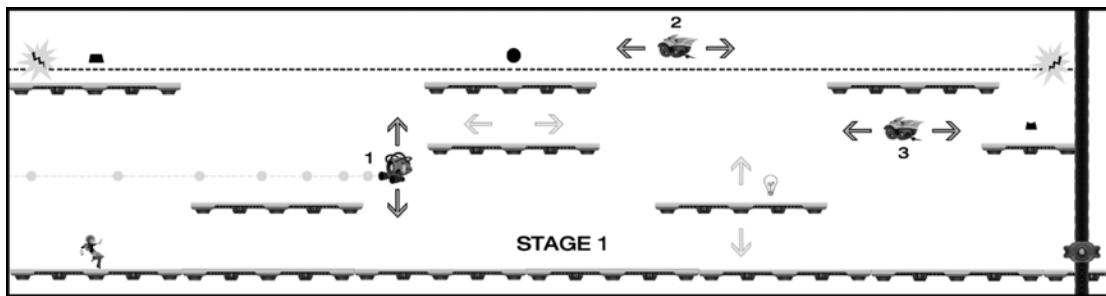


Figure 11-8. Two new object types are introduced to the level: a shooting robot (#1) that moves vertically and fires at a constant rate and a patrolling robot (#2) that moves back and forth in a specific range

You've reached a turning point in the design of this level, where the setting is now exerting a significant influence on the evolution of the mechanic. The core of the mechanic hasn't changed from Chapter 10. This level is still fundamentally about activating sections of a lock in the proper sequence to remove a barrier; moving platforms and attacking enemies are additional obstacles that add incremental challenge to play, strongly influenced by the particular setting you've chosen.

You certainly could have added the attacking enemy behavior while still working with abstract shapes and pure mechanics. However, it's worth noting 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 design to laying out a level (or part of a level) in the context of a particular setting when the mechanic is still fairly elemental is helpful. Designers typically want the game mechanic to feel deeply integrated with the game setting, so it's beneficial to allow both to develop in tandem. Finding that sweet spot can be challenging. Bring in the setting too soon and you lose focus on refining pure gameplay; bring in the setting too late, and gameplay may feel like an afterthought or something that's bolted on.

General Considerations

Returning to the current design as represented in Figure 11-8, you now have all the elements required to create a truly engaging sequence situated in an emerging setting. Players will need to observe patterns of movement for both platforms and enemies to time their jumps so they can navigate the level without getting zapped or rammed, all while discovering and solving the unlocking puzzle; you can also fairly easily tune the movement and placement of individual units to make the challenge harder or easier. Note how quickly the level went from trivially easy to complete to potentially quite challenging. Working with multiple moving platforms adds an element of complexity, and the need to use timing for jumps and adding attacking enemies—even the simple enemies from Figure 11-8 that are locked into basic movement patterns—opens nearly unlimited possibilities to create devious puzzles in a controlled and intentional way.

If you haven't already, now is a good time to prototype your level design (including interactions) in code to validate gameplay. For this early-level prototype, it's only important that major behaviors (running, jumping, projectile firing, moving platforms, object activations, and the like) and steps required to complete the level (puzzle sequences) are properly implemented. Some designers insist at this stage that players who have never encountered the level before should be able to play through the entire experience and fully understand what they need to do with little or no assistance, while others are willing to provide direction and fill in gaps around missing onscreen UI and incomplete puzzle sequences. In the authors' studio, we typically playtest and validate major sections of gameplay at this stage and provide playtesters with additional guidance to compensate for incomplete UI or unimplemented parts of a sequence. As a general rule, the less you need to rely on over-the-shoulder guidance for players at this stage, 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, for example). 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 and other 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.

Visual Design

Although you've begun integrating some visual design assets that align with the setting and narrative, the game typically will have few (if any) final production assets at this time, and any animations will be either rough or not yet implemented (the same is true for game audio). While it's good practice to have gameplay evolve in parallel with the game setting, studios don't want to burn time and resources creating production assets until the team is confident that the level design is locked and they know what objects are needed and where they'll be placed.

You should now have a fairly well-described layout and sequence for your level design. (If you've been experimenting with a different layout compared to what's shown in the examples, make sure you have a complete game flow described, as in Figure 11-1 and Figure 11-6.) At this point in the project, you can confidently begin "rezzing in" production assets (*rezzing in* is a term used by game studios to mean "increasing the visual polish and overall production quality of the level over time"). Rezzing in is typically a multistage process that begins when the major elements of the level design are locked, and it can continue for most of the active production schedule. There are often hundreds (or thousands) of individual assets, animations, icons, and the like that will typically need to be adjusted a number of times based on the difference between how they appear outside the game build compared to inside the game build. Elements that appear to harmonize well in isolation and in mockups often appear quite differently after being integrated into the game.

The process of rezding in assets can be tedious (there always seems to be an order of magnitude more assets than you think there will be) and frustrating (it can be challenging to make things look as awesome in the game as they do in artist mockups). However, it's also typically a satisfying time: something magical happens to a level design as it transitions from blockout to polished production level, and there will usually be one build where a few key visual assets have come in that make the team say "Wow, now this feels like our game!" For AAA 3D games, these "wow" moments frequently happen as high-resolution textures are added to 3D models and as complex animations, lighting, and shadows bring the world to life; for the current prototype level, adding a parallaxing background and some localized lighting effects should really make the spaceship setting pop.

The working prototype included with this book represents a build of the final game that would typically be midway between blockout and production polish. The hero character includes several animations 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.

Game Audio

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. Audio is an interesting topic in 2015. As with visual design, audio often contributes directly to the game mechanic (countdown timers, warning sirens, positional audio that signals enemy location, for example), and background scores enhance drama and emotion in the same way directors use musical scores to support the action on film. However, all 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.

Audio designers work with level designers to create a comprehensive review of game objects and events that require sounds, and as the visuals rez 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.

Interaction Model

The current prototype uses a common interaction model. A and D keys on the keyboard move the character right and left, and the spacebar is used to jump. Object activations in the world happen simply by colliding the hero character with the object, and the complexity is fairly low. Imagine, however, that as you continue building out the mechanic (perhaps in later levels), you include the ability for the character to launch projectiles and collect game objects to store in inventory. As the range of possible interactions in the game expands, complexity can increase dramatically, and unintentional challenges (as mentioned previously) can begin to accumulate, which can lead to bad player frustration (as opposed to “good” player frustration, which results from intentionally designed challenges).

It's also important to be aware of the challenges encountered when adapting interaction models between different platforms. Interactions designed initially for mouse and keyboard often face considerable difficulty when moving to a game console or touch-based mobile device. Mice and keyboard interaction schemes allow for extreme precision and speed of movement compared to the imprecise thumb sticks of game controllers, and although touch interactions can be precise, screens tend to be significantly smaller and obscured by fingers covering the play area. The industry took many years and iterations to adapt the first-person-shooter (FPS) genre from using mice and keyboards to game consoles (for example), and there are still no definitive FPS conventions for touch devices (which is largely responsible for why so few FPS games ship on tablets and phones). Many designers prototype games on PCs that they hope to eventually deliver on mobile devices and are often surprised at the difficulty in translating interaction metaphors. If you plan to deliver a game across platforms, make sure you consider the unique requirements of each as you're developing the game.

Game Systems and Meta Game

The current prototype has few systems to balance and does not yet incorporate a meta game, but imagine adding elements that require balancing such as variable-length timers for object activations and the energy field. If you're unsure what this means, consider the following scenario. The hero character has two potential ways to deactivate the energy field, and each option is a trade-off. The first option perhaps deactivates the energy field permanently but spawns more enemy robots and considerably increases the difficulty in reaching the target object, while the second option does not spawn 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 versus 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 no longer 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 (for example, score the most points or complete each level in the shortest time) with 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 scope of this book. However, it's most effective to teach players the logical rules and interactions of the game as they play through an introductory level or levels. It's also more effective to show players what you want them to do rather than making them read long blocks of text. (Research shows that many players never access optional tutorials and will dismiss tutorials with excessive text without reading them; one or two very short sentences per tutorial event is a reasonable target.) If you were creating an in-level tutorial system for your prototype, how would you implement it? What do you think players would reasonably discover on their own versus what you might need to surface for them in a tutorial experience? If you haven't played mobile games recently, it's worth exploring a few titles and paying attention to how they integrate tutorial content with gameplay.

User Interface (UI) Design

Game UI design is important not just from a functionality perspective (in-game menus, tutorials, and contextually important information such as health, score, and the like) but also as a contributor to the overall setting and visual design of the experience. Game UI is a core component of visual game design that's frequently overlooked by new designers and can mean the difference between a game people love and a game nobody plays. Think back to games you've played that make use of complex inventory systems or that have many levels of menus you must navigate through before you can access common functions or items; can you recall games where you were frequently required to navigate through multiple sublevels to complete often-used tasks? Or perhaps games that required you to remember elaborate button combinations to access common game objects?

Elegant and logical UI is critical to player comprehension, but aesthetically integrated UI also supports the game setting and narrative. Using the current prototype and proposed systems design as reference, how would you visually represent game UI in a way that supported the setting and aesthetic? If you haven't spent time evaluating UI before (and even if you have), revisit several games with sci-fi settings and pay particular attention to how they integrate UI elements visually in the game screen. Figure 11-9 shows the weapon customization UI from Visceral Games' *Dead Space 3*. Note how the interface design is completely embedded within the game setting, represented as an information screen on the fictional ship.



Figure 11-9. 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 still important and contributes directly to presence. 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 high-level discipline that can be challenging to master; you'll be well-served, however, by spending focused time to ensure elegant and aesthetically appropriate UI integration into the game worlds you create.

Game Narrative

You've added just a basic narrative wrapper to the prototype example: a hero character must complete a number of objectives to prevent her spaceship from exploding. At the moment, you haven't exposed this narrative to players at all, and they have no way of knowing the environment is on a spaceship or what the objective might be other than perhaps eventually unlocking the door at the far right of the screen. Designers have a number of options for exposing the game narrative to players. You might create an introductory cinematic or animated sequence that introduces players to the hero character, her ship, and the crisis. Perhaps choose something simple like a pop-up window at the start of the level with brief introduction text that provides players with the required information. Alternatively, you might not provide any information about what's happening when the game starts but instead choose to slowly reveal the dire situation of the ship and the objectives over time as the player proceeds through the game world. You could even choose to keep any narrative elements implied, allowing players to overlay their own interpretation. As with many other aspects of game design, there's no single way to introduce players to a narrative and no universal guidance for how much (or how little) narrative might be required for a satisfying experience.

Narrative can also be used by designers to influence the way levels are evolved and built out even if those elements are never exposed to players. In the case of this prototype, it's helpful as the designer to visualize the threat of an exploding ship to propel the hero character through a series of challenges with a sense of urgency; players, however, might experience a well-constructed side-scrolling action platformer only with a series of devilishly clever levels. You might create additional fiction around robots that have been infected with a virus, causing them to turn against the hero as a reason for their attack behavior (as just one example). By creating a narrative framework for the action to unfold within, you're able to make informed decisions about ways to extend the mechanic that feel nicely integrated into the setting even if you don't share all the background with players.

Of course, some game experiences have virtually no explicit narrative elements either exposed to players or not and are simply implementations of novel mechanics. Games like Zynga's *Words with Friends* and Gabriele Cirulli's hit *2048* are examples of game experiences purely based on a mechanic with no narrative wrapper.

If you continue developing this prototype, how much narrative would you choose to include, and how much would you want to expose to players to make the game come alive?

Bonus Content: Adding a Second Stage to the Level

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.

Figure 11-10 shows a layout in the style you've been using to prototype basic mechanics.

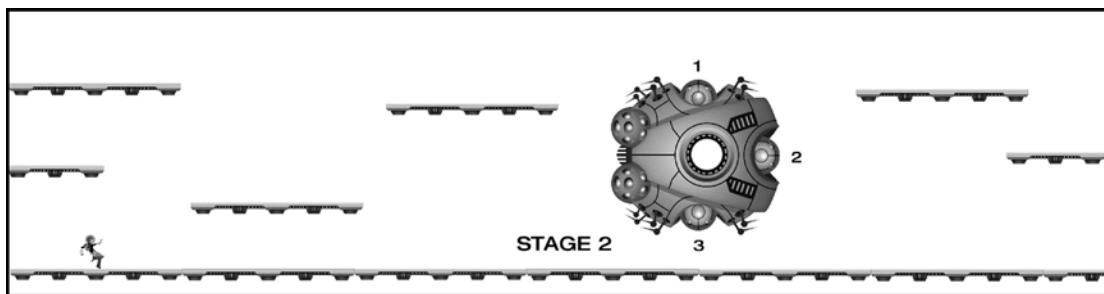


Figure 11-10. This is a possible second stage the hero character can enter after unlocking the door in stage 1. This concept includes a large “boss” unit with three nodes; one objective for this stage might be to disable each of the nodes to shut the boss down

It's a bit of a shortcut to begin the mechanic exploration with the diagram in Figure 11-10, 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.

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 units into the mechanic for disabling the nodes on the boss). If you find your designs becoming difficult to describe as part of an explicit and repeatable game flow, it may signal that you're working with more complex systems and may need to evaluate them in a playable prototype before you can effectively balance their integration with other components of the level. Of course, you can also reuse conventions and units from stage 1. You might choose to combine patrolling robots with hero-seeking robots and an energy field, creating a challenging web of potential risks for the player to navigate as they work to disable the boss nodes.

You might also decide that the main objective for the level is to *enable* the boss nodes in order to unlock the next stage or level of the game. You can extend the narrative in any direction you like, so units can be helpful or harmful, objectives can involve disabling or enabling, the hero character can be running toward something or away from something, or any other possible scenario you can imagine. Remember, narrative development and the level design will play off each other to drive the experience forward, so stay alert for inspiration as you become increasingly fluent with the level designs for this prototype.

Summary

Game design is unique among the creative arts in the ways it requires players to become active partners in the experience, which can change dramatically depending on who's in the driver's seat. Although some games share quite a bit in common with cinema (especially as story-driven indie games become more popular), there's always an unpredictable element when the player controls the action to a greater or lesser extent. Unlike movies and books, video games are interactive experiences that demand constant two-way engagement with players, and poorly designed mechanics or levels with unclear rules can block players from making progress through the experience you've created.

The design methodology presented in this book focuses first on teaching you the letters of the design alphabet (basic interactions), leading into the creation of words (game mechanics), followed by sentences (levels); we hope you'll take the next step and begin writing the next great novel (full-game experiences in existing or entirely new genres). The "escape the room" design template can be used to quickly prototype a wide range of mechanics for many kinds of game experiences, from the included 2D side-scroller to isometric games to first-person experiences and more. Remember, game mechanics are fundamentally well-formed abstract puzzles that can be adapted as needed. If you find yourself having difficulty brainstorming new mechanics in the beginning, borrow some simple 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. 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, and some of those variations might just contribute to the next breakthrough title.

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.

Index

A

addAsShadowCaster() function, 435
Ambient light
 background light, 274
 camera and scene objects, 279
 color and intensity, 275
 DefaultResources.js file, 278
 functionality, 279
 GLSL shaders, 276–277
 minion_sprite.png, 276
 mPublic() function list, 278
 MyGame.js, 278
 real time, 274
 RGB values, 281
 shader/renderable structure, 275
 SimpleShader, 277–278
 update function, 280
Angry Birds, 12
Audio, video games
 asyncLoadCompleted(), 119
 AudioClips engine, 118
 BlueLevel constructor, 124
 external resources, 117
 file formats, 118
 game engine development, 118
 initialize() function, 123
 isAssetLoaded() function, 121
 isKeyPressed() function, 125
 loadAudio() function, 119, 122
 load commands, 121
 loadScene() function, 122
 MapEntry object, 121
 mAUDIOContext, 118
 mAUDIOContext.decodeAudioData(), 119
 mBgAudioNode, 120
 MyGame scene, 122
 playACue() function, 120
 reference counts, 117
 ResourceMap.asyncLoadCompleted(), 119

ResourceMap.incAssetRefCount(), 119
ResourceMap.isAssetLoaded(), 119
sound effects, 116
unloadAsset() function, 122
update() function, 123

B

Background
 AAA first-person-shooter experience, 439
 aperture laboratories playground, 440
 creative process, 439
 engine development, 419
 game designers, 439
 game engine, 439
 groundbreaking games, 439
 large-scale parallaxing environments, 440
 layer management, 432
 Layer Manager Project, 432–438, 440
 motion parallax, 426–428, 438
 parallax, 420
 ParallaxGameObject in MyGame, 430–431
 parallaxing environment objects, 440
 ParallaxObjects project, 428–430
 players experience, 440
 professional game designers, 439
 Shadow Shaders project, 440
 side-scrolling game, 419
 test tiled objects, 425–426
 TiledGameObject, 422–424
 tiled objects project, 421–422
 Tiled Objects project, 440
 tiling, 419–420
 visual complexity, 420
Birds-style block destruction games, 365
Bounding box
 and collisions project (*see* Collisions project)
 definition, 209–210
 description, 207
 in engine, 211
 lower-left corner and size of object, 207

■ INDEX

- Bounding box (*cont.*)
WC rectangular area/WC window, 207
with MyGame, 212
- BoundingBox class, 371
- Brain GameObject, 205–206
- C**
- Camera
Camera Objects Project, 71–74
testing, 74–76
- Camera manipulation
abrupt transitions, 234
abstraction mechanism, 233
aXform object, 236
behaviors, 233, 267
canvas coordinate space, 266
clampAtBoundary() function, 236, 239
convenient functions, 234
Dye character, 235
external resources, 236
functionality, 240
game programmer, 234
Hero object, 239
HTML canvas, 233
lighting, 267
minion_sprite.png, 236
MyGame.js file, 238
panTo() function, 239
panWidth() function, 237
Portal object, 235
setUpViewProjection()
 function, 233
source code, 234, 236
target position, 238
Transform object, 237
update() function, 238
utility functions, 266
video camera, 234
WASD keys, 235
WC, 233
zone variable, 236
zoomBy() function, 238
zoom functions, 234
zoomTowards() function, 238
- Camera modification
per-render invocation, 290
setUpViewProjection() function, 291
transformation, 290–292
utility functions, 290
WC space, 290
- Camera shake project
abstraction layer, 251
CameraState parameter, 251
constructor, 251
- disorientation, 247
displacements, 251
Dye character, 248
frac variable, 249
harmonic motion, 247, 249
mathematical formulations, 246
Math.random() function, 250
mCamera.shake() function, 253
modeling displacements, 249
nextDampedHarmonic() function, 250
portal object, 248
pseudorandomness, 247, 250
resource files, 249
setting functions, 251
setUpViewProjection() function, 252
shakeDuration parameter, 249, 253
shakeFrequency parameter, 249, 253
ShakePosition.js, 249
source code, 247
trigonometric functions, 247
update() function, 253
xDelta and yDelta variables, 249
- Candy Crush, 11
- clamp() function, 376
- Client source code
index.html body element, 39
src/MyGame folder, 38
collide() function, 378
collided() function, 377, 388–389
collidedCircCirc() function, 378, 388
collidedRectCirc() function, 375
collidedRectRect() function, 377, 387
- Collision resolution
axis-aligned rigid rectangles and
circles, 381
CollisionInfo object, 385
observations, 396
physics engine component, 389–390, 392–396
Rigid Circle collision, 388–389
RigidRectangle collision, 387–388
RigidBody collision, 385–387
Rigid Shape Impulse project
 (*see* Rigid Shape Impulse project)
symplectic Euler integration, 383–384
testing impulse resolution, 396
- Collisions project
controls of, 208
goals of, 209
running of, 208
containsPos() function, 377–378
- Coordinate systems and transformations
major axes, 61
NDC (*see* Normalized device coordinates
 (NDC) system)
origin, 61

2D Cartesian, 61–62
 WC system (*see* World coordinate (WC) system)
 Cross product, 199–200

D

Diffuse reflection
 3D shapes, 315
 activateShader() function, 309
 activateTexture() function, 313
 boundary surfaces, 302
 cleanUp() function, 312
 color texture, 304
 createShaders() function, 312
 default system, 306
Engine_DefaultResources.js, 311
 GLSL shader, 306
 human vision system, 303–304
IllumFS.gsls file, 306
 IllumRenderable object, 310–311
 IllumShader object, 305
 initialize() function, 312
 JavaScript IlluminationShader, 309
 light attenuation, 307
 light computation, 305
 LightEffect() function, 307
 map integration, 306
minion_sprite_normal.png, 306
 normalized vector, 302
 normal mapping, 303
 peculiar effect, 302
 public function, 312
 resource files, 306
 RGB channels, 303
 static background, 315
 surface normal vectors, 302
 texture mapping, 303
 TextureShader object, 310
 texture uv coordinates, 308
 uNormalSampler, 308
 WebGL texture, 309, 313
 Dot product, 198–199
 draw() function, 373–374, 401
 Drawing operations
 encapsulation, 47, 48
 subregions, 47
 Drawing operations. *See* Renderable Objects project
 Draw One Square project
 global variable *gSquareVertexBuffer*, 25
 GLSL shaders (*see* GLSL shaders)
 goals of, 24
 initSquareBuffer() function, 25
 running of, 23–24
 STATIC_DRAW informs, 25
 VertexBuffer.js, 24

The DyePack GameObject, 192–193
 DyePack objects, 397

E

Engine core initialization
 Engine.Core.initializeEngineCore(), 103
 initializeEngineCore() function, 102
 onload event, 102
 SimpleShaders, 103
Engine_Core.js file, 396
 Engine core modification
 caster support, 345
 clearCanvas() function, 346
Engine_DefaultResources.js, 346
 GLSL shaders, 345
 initializeWebGL() function, 345
 shadow caster object, 346–348, 350
 SpriteShader, 346
 WebGL stencil buffer, 345
Engine_GameLoop.js file, 396
Engine_Physics.js file, 393
 Extensible Markup Language (XML), 105

F

First-person-shooter (FPS) genre, 451
 FontRenderable objects
 aString variable, 175
 bitmap fonts, 170–171
 CharacterInfo, 174
 cleanUp() function, 178
 Consolas-72.png, 172
 draw() function, 175
Engine_VertexBuffer.js, 178
 GameOver scene, 180–181
gEngine_DefaultResources, 178
gEngine_Fonts, 175
 getCharInfo() function, 176
 initialize() function, 177
 initText() function, 182
 loadScene() function, 181
minion_sprite.png, 172
 mOneChar variable, 175
 mOneChar.setElementUVCoordinate()
 function, 176
MyGame scene, 181, 183–184
 ResourceMap.retrieveAsset(), 174
 setFont() function, 183
 setTextHeight() function, 177
SimpleShader.js, 179
 storeLoadedFont() function, 173
 system-default-font, 172
 unloadScene() function, 181
 uv coordinates, 173

- Front and chase project
 controls of, 201
 functionality
 Brain GameObject, 205–206
 MyGame Scene, 206–207
 GameObject modification, 202–205
 gl-matrix library, 202
 goals of, 201
 running of, 200–201
- G**
- Game design
 2D jumping and puzzle-solving
 mechanic, 441–442
 activation, 269–270, 361
 assortment, 269
 attacking enemy, 448
 basic narrative wrapper, 453–454
 “blockout-plus” stage, 449
 bonus content, 454–455
 challenge, 447
 chiaroscuro techniques, 357
 contextual images, 444
 dark environment, 358
 development teams, 362
 early-level prototype, 449
 “elevator” platform, 442
 experience, 455
 explicit steps, 442
 flashlight, 359–360
 frustration, 272
 game audio, 450
 game environment, 357
 game systems and meta game, 451–452
 genres, 267
 hero character, 268, 270, 357
 horizontal “energy field”, 442
 interaction model, 267, 362, 451
 interactive experience, 267
 keyboard support project, 268
 lighting, 357
 localized environment, 357
 logical consistency, 269, 272, 362
 mastering, 267
 mechanics, 268, 356
 metaphors, 361
 mini-map, 272
 multistage task, 271
 playable space, 445, 447
 player controls, 358
 puzzle-solving game, 272
 reinforcement, 267
 sci-fi setting, 443
 sequence, 271
 sequencing principles, 361
 technical and experiential aspects, 455
 user interface (UI) design, 452–453
 visual design, 449
 well-formed mechanics, 441
- Game engine
 gEngine.Core, 32–34
 resource management subsystem, 32
 self-contained subsystems, 32
 single-instance/Singleton-like objects, 32
- Game loop
 draw() functions, 83
 draw() operation, 79
 Engine_GameLoop.js, 81
 Engine_VertexBuffer, 81
 frame rate, 78
 incRotationByDegree(), 85
 incSizeBy() functions, 85
 initialize() function, 83
 kMPF interval, 86
 pseudocode, 78
 real time, 78
 requestAnimationFrame() function, 82
 runLoop() function, 82, 84
 runLoop.call(mMyGame), 82
 source code, 79
 start() function, 83
 update() function, 79, 85
- Game objects. *See also* Game Objects project
 chasing behavior, 196
 initialize() and update() functions, 188
- Game Objects project
 definition, 191
 The DyePack GameObject, 192–193
 goals of, 189
 The Hero GameObject, 193
 The Minion GameObject, 194
 MyGame Scene, 194–195
 new sprite elements, minion_sprite.png
 image, 189–190
 renderable and shader objects
 modification, 190–191
 running of, 188–189
 set management, 191–192
 WASD keys, 189
- Generalized per-pixel collisions project
 axes-aligned texture, 223
 controls of, 224
 GameObject_PixelCollision.js
 modification, 226–227
 goals of, 224
 rotated texture and component
 vectors, 223
 running of, 224
 testing, 227

TextureRenderable_PixelCollision
modification, 225–226
two normalized component vectors, vector decomposition, 222
gEngine.Core property, 32–34
Geometric data and OpenGL Shading Language (GLSL), 23
gl.blendFunc() setting, 403
GLCanvas, canvas element, 17
glMatrix Library, 53–54
GLSL. *See* Geometric data and OpenGL Shading Language (GLSL)
GLSL shaders
activateShader() function, 137
aTextureCoordinate attribute, 134
description, 26
mShaderVertexPositionAttribute, 137
sampler2D data, 134
ShaderSupport.js source file, 26
SimpleShader object, 136
SimpleShader.call() syntax, 137
syntax, C programming, 26
texture2D() function, 135
TextureShader.js, 137
texture vertex, 133
TextureVS.glsl, 133
uPixelColor, 135
varying vTexCoord, 134
vertex and fragment shaders (*see* tex and fragment shaders)
vertexShaderPath, 137

H

Heads-up display (HUD) layer, 432
The Hero GameObject, 193
Hero.js file, 380
HTML5 Canvas project
editing index.html file, 16
GLCanvas, 17
gl.clearColor(), 18
goals of, 16
running of, 15–16

I

IllumRenderable objects, 365
initialize() function, 437
initializeEngineCore() function, 436
Integrated development environment (IDE). *See* NetBeans IDE
Interpolation
annoyance/confusion, 240
assets folder, 242
BoundingBox objects, 242

camera parameters, 240
CameraState object, 244–245
constant speed, 240
Dye character, 241
gl-matrix.js file, 244
Hero object, 241
interpolateValue() function, 243
InterpolateVec2 utility, 241
linear and exponential functions, 240
mCurrentValue, 242
mFinalValue, 242
MyGame update() function, 246
operations, 241
panBy() function, 246
Portal object, 242
this.mCameraState, 245
transform objects, 242
trivial replacements, 246
vec2.lerp() function, 244
WC center, 245

J

JavaScript objects project
goals of, 31
project creation, 31
running of, 30–31
source code organization, 31–32
JavaScript source file project
clearCanvas() function, 22
doGLDraw() function, 22
goals of, 19
initializeGL() function, 21
loading and running, 22–23
new JavaScript source code file, 20–21
new source code folder creation, 19–20
running of, 18–19

K

Keyboard input
engine component, 87
functionality, 86
gEngine_VertexBuffer, 88
initializeWebGL() function, 91
mIsKeyClicked, 89
mKeyPreviousState, 89
MyGame constructor, 92
MyGame.prototype.update() function, 92
onKeyUp/Down() event, 90
public interface, 89
runLoop() function, 91
update() function, 90
WebGL, 91
window.addEventListener(), 90

L**Light source**

accessor function, 289
 activateShader() function, 287
 camera, 290–292
 cleanUp() function, 290
 computations, 285
 convenient interface, 287
 createShaders() function, 289
 DC space, 285
 distance attenuation, 327
 energy traveling, 281
 Engine_DefaultResources.js file, 289
 environment, 281
 geometries, 282
`gl_FragCoord.xyz`, 285
 global lights, 327
 GLSL fragment shaders, 283
 initialize() function, 289
 GLSL LightFS shader, 286
 light renderable object, 288
 main() function, 285
`mPublic()`, 290
 MyGame level, 292
 Phong illumination model, 282
 pixel colors, 282
 point light, 281, 292, 327
 SimpleLightShader, 283–284
 space functionality, 288
 spotlight models, 327
 SpriteShader object, 287
 TextureRenderable objects, 282
 WebGL, 282

M**Mapping**

normalized system, 128
 resolution, 128
 shaders (*see Shaders project*)
 texel color, 128
 texture space, 129
 uv values, 129

`Math.ceil()` function, 424

Matrix operators

concatenation, 53
 4×4 identity matrix, 52
 4×1 vector, 52
 rotation operator $R(\theta)$, 52
`gIMatrix Library`, 53–54
 m -rows by n -columns array of numbers, 51
 scaling operator $S(sx, sy)$, 51–52
 translation operator $T(tx, ty)$, 51

Matrix Transform project

goals of, 55

renderable objects, 54

Renderable object modification, 56–57

running of, 54–55

SimpleShader modification, 56

testing, 57

vertex shader modification, 55–56

`mDrawBounds` variable, 371

The Minion GameObject, 194

`Minion.js` file, 380

Mouse input, camera

`camera.isMouseInViewport()`, 266

canvas coordinate space, 259

`canvaid` parameter, 263

device coordinate (DC) space, 260

event handler, 263

`gEngine_Input` component, 262

`initialize()` function, 263

`initializeEngineCore()` function, 262

keyboard input, 262

main functionality, 265

`mHeroCam` view, 266

portal object, 261, 266

resource files, 262

transform positions, 260

`update()` function, 264–265

WC space, 259, 264–265

`mParallaxScale`, 429

`mPhysicsComponent`, 379

`mRelaxationOffset`, 390

Multiple cameras

assets folder, 255

bound-number, 255

Camera objects, 253

`draw()` function, 258–259

gameplay information, 253

`getViewport()` function, 256

`gl.scissor()` function, 256

`gl.viewport()` function, 256

HTML canvas, 259

`initialize()` function, 257

interpolation, 255

`mBrainCam` object, 257

`mBrainCam.configInterpolation()` function, 259

`mHeroCam`, 257

`mScissorBound`, 256

`mViewport`, 255

`setUpViewProjection()` function, 256

`setViewport()` function, 255

`update()` functions, 257

video games, 253

WASD keys, 254

WC window, 253

Multiple lights project

`activateShader()` function, 300

additive property, 302

array of lights, 293

- attenuation and intensity, 297
- distance attenuation, 293
- hero character, 294
- initialize function, 301
- intensity, 294
- `LightEffect()`, 296
- `LightFS.gsl` file, 295
- `LightRenderable` object, 301
- `LightSet` Object, 298
- `loadToShader()` function, 299
- `minion_sprite.png`, 295
- point light, 293
- quadratic attenuation, 296
- quadratic functions, 293
- radius variable, 297
- repetitive code listings, 301
- `setLight()` function, 300
- `ShaderLightAtIndex` object, 293, 298–299
- single point light source, 293
- single scene, 294
- `uLights` array, 299
- `MyGame` initialize() function, 425
- `MyGame_Physics.js` file, 412
- `MyGame._physicsSimulation()` function, 396
- `MyGame` Scene, 194–195

N, O

- `NDc`. *See* Normalized device coordinates (NDC) system
- NetBeans IDE
 - action items window, 4
 - editor window, 4
 - HTML5 project
 - application, 6
 - creation, 5
 - folders and files, 8
 - naming, 6
 - running, 7
 - selection, 5
 - projects window, 4
- Newtonian movement formulation, 366
- Normalized device coordinates (NDC) system, 62
- Normal map functionality
 - color texture, 314
 - Hero and Minion objects, 314
 - `IllumRenderable` object, 314
 - load and unload, 315

P, Q

- `ParallaxGameObject`, 429
- Parameterized fragment shader project
 - goals of, 44
 - new shader drawing, 45
- running of, 43
- `SimpleFS.gsl` Fragment Shader, 44
- SimpleShader modification, 44–45
- Particle Emitters Project
 - controls of, 409
 - goals of, 409
 - `minion_sprite.png`, 409
 - observations, 412
 - `ParticleEmitter` Object, 409–410
 - Particle Game Object Set, 411
 - running, 409
 - testing, 411
- `ParticleGameObjectSet.js` file, 411
- Per-pixel collisions project *See also* Pixel Collisions project
 - between large and small texture, 215
 - controls of, 214
 - `Engine_Texture` component, 216–217
 - `GameObject_PixelCollision.js`, 220
 - goals of, 214
 - limitation with bounding box-based collision, 213
 - in `MyGame`, 220–221
 - nontransparent pixels overlapping, 213
 - overlapping bounding boxes without actual collision, 215
 - `pixelCameraSpace`, 216
 - running of, 213–214
 - `TextureRenderable` modification, 217–218
 - `TextureRenderable_PixelCollision.js` File, 218–220
- Phong illumination model, 274
- Physics simulations in games
 - analogous real-world environment, 418
 - collision detection, 368
 - collision resolution (*see* Collision resolution)
 - engine logic processes, 397
 - force field blocking access to upper platforms, 417
 - mechanic, 416
 - movement, 366–367
- Particle Emitters Project (*see* Particle Emitters project)
- particles project
 - controls of, 398
 - creating, 401
 - default particleShader instance, 399
 - `Engine Particle Component`, 404–406
 - GLSL particle fragment shader, 399
 - goals of, 398
 - `minion_sprite.png`, 398
 - observations, 408
- `ParticleGameObject`, 402–403
- `ParticleGameObjectSet`, 403–404
- `ParticleRenderable` object, 400
- with `RigidShape`, 407

Physics simulations in games (*cont.*)
 RigidBody object, 397
 running, 398
 testing, 407–408
 platformer puzzle, 416
 puzzle levels, 417
 RigidBody objects, 412
 Rigid Shape Bounds Project (*see* Rigid Shape Bounds project)
 Rovio's Angry Birds requires players, 413
 simple and flexible particle system, 397
 spaceship setting, 417
 UV light, 415
 Verlet particles and particle emitters projects, 414
 P key, 433
 Pong-like reaction-based games, 365
 processObjSet() function, 394
 processSetSet() function, 394

■ R

Renderable Objects project
 goals of, 49
 matrix transformations, 51
 programmability and extensibility, 49
 running of, 48
 testing, 50
 resolveCollision() function, 391
 resolveRectPos() function, 405, 406
 Resource management
 asyncLoadCompleted() function, 96
 callbackFunction, 99
 checkForAllLoadCompleted(), 96
 compileShader(), 101
 createShaders() function, 100
 DefaultResources._createShaders()
 function, 102
 engine core, 102–103
 external resources, 93
 fileName parameter, 98
 GLSL shaders, 94, 97
 initialize() function, 100
 loadAndCompileShader(), 101
 MapEntry, 96
 mAsset, 95
 mConstColorShader, 101
 mLoadCompleteCallback, 95
 mNumOutstandingLoads, 95
 real-time interactivity, 93
 renderable objects, 99
 ResourceMap.isAssetLoaded(), 99
 ResourceMap.retrieveAsset(), 101
 SimpleShader object, 100
 SimpleVS.glsl, 93

source code, 94
 startScene() function, 102
 TextFileLoader, 97, 99
 XMLHttpRequest, 99
 XMLHttpRequest.open(), 93
 Rezzing, 450
 RigidCircle_Collision.js file, 388
 RigidCircle.js., 373
 RigidRectangle_Collision.js, 377, 387
 RigidRectangle.js, 372
 Rigid Shape Bounds project
 controls of, 370–371
 game objects, 380
 MyGame object, 380–381
 rectangle and circle, 375–376
 RenderableLine class, 370
 Rigid Circle Object, 373–374
 Rigid Rectangle Object, 372–373
 Rigid Shape Base Class, 371–372
 two circles, 378–379
 two rectangles, 377–378
 RigidBody_Collision.js, 375, 385
 Rigid Shape Impulse project
 controls of, 382
 goals of, 382
 running, 382
 symplectic Euler integration, 381
 RigidBody.setMess() function, 396
 rigidType() function, 373–374
 Rovio's Angry Birds requires players, 413
 rVelocityInNormal, 392

■ S

Scene object
 BlueLevel, 111, 113
 EngineCore.startScene() function, 114
 Engine_GameLoop.js, 112
 game behaviors, 112
 GameLoop.stop() function, 110, 114, 116
 inheritPrototype() function, 112–113
 initialize() function, 115
 loadScene() function, 114
 MyGame.unloadScene(), 116
 runLoop() function, 112
 startScene(), 110
 superclass, 110
 unloadScene() function, 112
 update() function, 114
 update()/draw(), 116
 setMass() function, 384
 Shader object, 35, 37–38
 Shader Source Files project
 goals of, 40
 HTML Code cleaning up, 42

loading shaders, SimpleShader, 40–41
 running of, 39–40
 shaders extraction into files, 41–42
Shaders project
 activateTexture() function, 140
 blendFunc() function, 141
 canvas.getContext(), 141
 createTexture() function, 143
 draw() function, 140
 Engine_DefaultResources, 131, 138
 Engine_Textures.js, 131
 Engine_VertexBuffer.js, 135
 gEngine_Texture.activateTexture(), 150
 gEngine_Texture._processLoadedImage(), 150
 GLSL, 130, 132
 initialize() function, 136
 initializeWebGL(), 140
 loadScene() function, 150
 mGLTexID, 141
 mTextureCoordBuffer, 135
 MyGame.js, 131
 pixelStorei() function, 141
 processLoadedImage() function, 142
 retrieveAsset() function, 144
 SimpleVS/FS, 131
 texImage2D() function, 143
 texParameteri() function, 144
 TextureRenderable class, 133, 139–140
 TextureShader.js, 131
 TextureVS.gsls, 131
 transparency, 130
 unloadTexture() function, 142
 WebGL, 130, 141
Shadow caster object
 aLight parameter, 350
 computation, 350
 computeShadowGeometry()
 function, 348
 cxf transform, 350
 distToReceiver, 350
 draw() function, 347
 geometries, 346
 LightRenderable object, 347
 mShadowReceiver, 347
 renderable object, 346
 SpriteRenderable object, 347
 variables, 348
Shadow implementation
 activateShader() function, 344
 aesthetic effect, 273
 ambient light, 274–281
 AngularDropOff(), 342
 Camera WC center, 353
 caster geometry, 337
 casting shadow, 337
 cloneTo() function, 353
 computation, 337
 diffuse reflection, 273
 DistanceDropOff() functions, 342
 drawCamera() function, 355
 external resource files, 341
 functionality, 338
 functional modules, 273
 game fidelity, 337
 gl_FragColor, 343
 GLSL, 274
 Hero object, 338
 human vision system, 337
 infrastructure, 273
 initialize() function, 354
 intensity, 339
 kMaxShadowOpacity, 341
 lighting model, 273
 LightStrength() function, 342
 main() function, 342
 material properties, 341
 mIllumHero object, 355
 Minion object, 338
 minion shadow receiver, 339
 minion_sprite.png, 341
 mLgtHero shadow, 355
 MyGame_Shadow.js, 354
 Phong illumination model, 356
 physical models, 356
 pixels, 338
 receiver, 338
 renderable objects, 344
 setupShadow() function, 354
 setUpViewProjection() function, 353
 ShadedResult() function, 342
 shaders project, 340
 ShadowCasterFS fragment, 341
 ShadowCasterShader, 344
 simulation algorithm, 338–340
 specular reflection, 273
 SpriteAnimateRenderable, 355
 SpriteShader object, 343, 352
 swapShader() function, 352
 TextureFS.gsls, 343
 Transform utility, 353
 transparencies, 339
 visual fidelity, 273
 WebGL stencil buffer, 338
Shadow receiver object
 addShadowCaster(), 351
 draw() function, 351
 mShadowCaster, 351
 ShadowReceiver_SignedDistanceField.js, 352
 shadowRecieverStencilOn(), 352
 WebGL, 350

Shared vertex buffer
 gEngine.VertexBuffer object, 34–35
 initSquareBuffer() function, 35

Source code organization, 43

Specular reflection
 aCamera.getPosInPixelSpace(), 325
 activateShader() function, 324
 ambient lighting, 317
 computation cost, 317
 constructor, 322
 diffuse lighting, 315
 DiffuseResults() functions, 321
 draw() function, 325
 halfway vector, 317
 hero character, 319
 IllumFS GLSL, 318
 IllumRenderable object, 325
 IllumShader object, 324
 initialize() function, 326
 integration, material, 318
 LightAttenuation() function, 320
 light source, 315, 317
 main() function, 322
 materials property, 317
 minion_sprite.png, 320
 MyGame_MaterialControl.js, 327
 PerRenderCache function, 325
 Phong illumination model, 315
 Phong lighting computation, 318
 real-life experience, 316
 reflection direction, 316
 resource files, 320
 selectCharacter(), 327
 setMaterialAndCameraPos() function, 324
 setUpViewProjection() function, 326
 ShadedResult() function, 321
 ShaderMaterial object, 323
 shininess, 316
 shiny surface, 315
 SpecularResult(), 321
 surface material property, 322
 uCameraPosition, 320
 wcPosToPixel() function, 326

Spot lights project
 angular attenuation, 332
 AngularDropOff() function, 331
 boundary edges, 336
 camera transform object, 336
 default system fonts, 330
 directional lights, 328–329, 336
 DropOff variable, 331
 half angles, 336
 IllumFS.gsl, 330
 intensity lights, 336
 LightFS.gsl source code, 333

LightShader fragment, 330
 loadToShader() function, 335
 material property controls, 330
 minion_sprite.png, 330
 mInnerRef, 336
 mOuterRef, 336
 MyGame_LightControl.js, 336
 num variable, 332
 Phong illumination model, 330
 setShaderReferences() function, 334
 ShadedResults() function, 333
 ShaderLightAtIndex object, 334
 smoothstep() function, 332
 wcDirToPixel() function, 336

Sprite animations
 constructor, 165
 eAnimateSwing, 164
 eAnimationType, 164
 game developer, 162
 GameLoop_runLoop() function, 168
 initAnimation() function, 165
 initialize() function, 168
 mCurrentAnimAdvance, 165–166
 mFontImage, 162
 retraction, 162
 setSpriteElement(), 166
 setSpriteSequence() function, 167, 169
 SpriteAnimateRenderable, 164
 texture-mapping, 162
 transformation operators, 161
 updateAnimation() function, 168

Sprite Pixel Collisions project
 goal of, 228
 in MyGame, 230
 modification, 229–230
 running of, 228
 SpriteRenderable_PixelCollision.js.
 creation, 229
 TextureRenderable, SpriteRenderable, and
 SpriteAnimateRenderable objects, 228
 TextureRenderable modification, 230

SpriteRenderable objects, 365

Sprite sheets
 activateShader() function, 155
 draw() function, 157
 dye character, 151
 Engine_DefaultResources.js, 158
 Engine_VertexBuffer.js, 153
 getGLTexCoordRef() function, 153
 GLSL texture shaders, 154–155
 initialize() function, 159
 kMinionSprite, 159
 minion_sprite.png, 151, 153
 model space, 151
 MyGame.js file, 158

- pixel locations, 150
 - `setElementPixelPositions()` functions, 159
 - `setElementUVCoordinate()`, 161
 - source code, 152
 - SpriteRenderable objects, 154, 156
 - `texCoord`, 155
 - TextureRenderable objects, 156
 - uv values, 151
 - Symplectic Euler Integration method, 367

T

 - Texture coordinates
 - bitmap fonts, 184
 - custom-composed images, 127
 - game design, 184–185
 - loading and unloading, 127
 - rendering, 127
 - WebGL, 128
 - Texture mapping functionality
 - `BlueLevel.xml` scene, 145
 - `BlueScene` scene, 145
 - `draw()` function, 149
 - `initialize()` functions, 146, 149
 - `loadScene()`, 146
 - `minion_collector.jpg`, 146
 - `MyGame` constructor, 148
 - `ResourceMap`, 149
 - `SceneFileParser.js`, 146
 - TextureRenderable object, 148
 - TextureSquare element, 146
 - `unloadScene()` function, 147
 - `update()` function, 147, 149
 - TiledGameObject, 422–424
 - Tiled Objects Project, 421–422
 - Tiling, 420–421
 - Transform Objects project
 - concatenated transform operator, TRS, 60
 - drawing modification, 60
 - getters and setters, 59
 - goals of, 59
 - running of, 58
 - `src/Engine` folder, 59
 - transformable renderable objects, 60
 - 2D game engine development, JavaScript
 - audio design, 12
 - big-budget, 10
 - game mechanics, 10
 - games settings, 11
 - graphical user interface (GUI), 1
 - guidance, 9
 - HTML5, 2
 - inheritance, 2
 - interaction model, 11
 - JSLint, download and installation, 3
- level design, 11
 - meta-game, 12
 - NetBeans (*see* NetBeans IDE)
 - programmability and maintainability, 1
 - set up
 - Connector Google Chrome plug-in, 3
 - `glMatrix` math library, 3
 - IDE, 3
 - runtime environment, 3
 - Unix, 3
 - software library, 1
 - systems design, 10
 - technical design, 10
 - visual design, 12
 - web browsers, 2
 - `_positionalCorrection()` function, 390
 - `_refPosUpdate()` function, 430
-
- ## U
- `unloadScene()` function, 437
 - `update()` function, 402
-
- ## V
- `vec2.scaleAndAdd()` function, 430
 - Vectors
 - 2D space with two vectors equal to each other, 198
 - being normalized, 197
 - cross product, 199–200
 - definition, 196–197
 - dot product, 198–199
 - object movements and behaviors, 196
 - object's velocity/acceleration, 196
 - origin to position being rotated by angle theta, 197–198
 - Vertex and fragment shaders
 - `clearCanvas()` function, 29
 - compile, link and load, 27–28
 - definition, 26–27
 - `doGLDraw()`, 30
 - `initializeGL()` function, 29
 - Video games
 - causal chain, 126
 - constructor, 107
 - decision factors, 105
 - `DefaultResources`, 109
 - drawing operations, 77
 - functional interface, 103
 - game environments, 126
 - `initialize()` function, 105
 - interpret player, 77
 - `loadScene()`, 108
 - loop (*see* Game loop)

■ INDEX

- Video games (*cont.*)
 - MyGame.unloadScene() function, 110
 - Number() function, 107
 - parsing utility, 105
 - public interface, 104
 - renderable objects, 107
 - resource management, 77
 - responsiveness, 125
 - SceneFileParser.js, 106
 - sceneFilePath, 106
 - shader Loads project, 126
 - startLoop() function, 110
 - startScene() function, 109
 - translation errors, 126
 - unloadScene() function, 108
 - update()/draw(), 107
 - Viewport
 - description, 64
 - working with, 64
 - View-Projection transform
 - goals of, 65
 - RenderObject modification, 66
 - running of, 64–65
 - SimpleVertex modification, 66
 - testing
 - design implementation, 68–70
 - scene designing, 67
 - vertex shader modification, 65–66
 - View-Projection transform *See* Camera
 - View-Projection transform operator, vpMatrix, 63
 - Visceral Games' Dead Space 3, 453
- ## ■ W, X, Y, Z
- WASD keys, 433
 - WebGL*See also* Draw One Square project; HTML5 Canvas project; JavaScript Source File Project
 - description, 15
 - drawing, 15
 - World Coordinate (WC) System, 233
 - description, 62
 - glMatrix library, 63
 - View-Projection transform operator, vpMatrix, 63
 - working with, 63