

INTRODUCTION TO

GAME DEVELOPMENT

USING PROCESSING



INTRODUCTION TO GAME DEVELOPMENT USING *PROCESSING*

LICENSE, DISCLAIMER OF LIABILITY, AND LIMITED WARRANTY

By purchasing or using this book (the “Work”), you agree that this license grants permission to use the contents contained herein, but does not give you the right of ownership to any of the textual content in the book or ownership to any of the information or products contained in it. *This license does not permit uploading of the Work onto the Internet or on a network (of any kind) without the written consent of the Publisher.* Duplication or dissemination of any text, code, simulations, images, etc. contained herein is limited to and subject to licensing terms for the respective products, and permission must be obtained from the Publisher or the owner of the content, etc., in order to reproduce or network any portion of the textual material (in any media) that is contained in the Work.

MERCURY LEARNING AND INFORMATION (“MLI” or “the Publisher”) and anyone involved in the creation, writing, or production of the companion disc, accompanying algorithms, code, or computer programs (“the software”), and any accompanying Web site or software of the Work, cannot and do not warrant the performance or results that might be obtained by using the contents of the Work. The author, developers, and the Publisher have used their best efforts to insure the accuracy and functionality of the textual material and/or programs contained in this package; we, however, make no warranty of any kind, express or implied, regarding the performance of these contents or programs. The Work is sold “as is” without warranty (except for defective materials used in manufacturing the book or due to faulty workmanship).

The author, developers, and the publisher of any accompanying content, and anyone involved in the composition, production, and manufacturing of this work will not be liable for damages of any kind arising out of the use of (or the inability to use) the algorithms, source code, computer programs, or textual material contained in this publication. This includes, but is not limited to, loss of revenue or profit, or other incidental, physical, or consequential damages arising out of the use of this Work.

The sole remedy in the event of a claim of any kind is expressly limited to replacement of the book, and only at the discretion of the Publisher. The use of “implied warranty” and certain “exclusions” vary from state to state, and might not apply to the purchaser of this product.

INTRODUCTION TO GAME DEVELOPMENT USING *PROCESSING*

JAMES R. PARKER



MERCURY LEARNING AND INFORMATION

Dulles, Virginia
Boston, Massachusetts
New Delhi

Copyright ©2015 by MERCURY LEARNING AND INFORMATION LLC. All rights reserved.

This publication, portions of it, or any accompanying software may not be reproduced in any way, stored in a retrieval system of any type, or transmitted by any means, media, electronic display or mechanical display, including, but not limited to, photocopy, recording, Internet postings, or scanning, without prior permission in writing from the publisher.

Publisher: David Pallai
MERCURY LEARNING AND INFORMATION
22841 Quicksilver Drive
Dulles, VA 20166
info@merclearning.com
www.merclearning.com
(800) 232-0223

James R. Parker. *Introduction To Game Development Using Processing*.
ISBN: 978-1-937585-40-2

The publisher recognizes and respects all marks used by companies, manufacturers, and developers as a means to distinguish their products. All brand names and product names mentioned in this book are trademarks or service marks of their respective companies. Any omission or misuse (of any kind) of service marks or trademarks, etc. is not an attempt to infringe on the property of others.

Library of Congress Control Number: 2015934495

151617321 This book is printed on acid-free paper.

Our titles are available for adoption, license, or bulk purchase by institutions, corporations, etc.
For additional information, please contact the Customer Service Dept. at 800-232-0223(toll free).

All of our titles are available in digital format at authorcloudware.com and other digital vendors. Companion files (figures and code listings) for this title are available by contacting info@merclearning.com. The sole obligation of MERCURY LEARNING AND INFORMATION to the purchaser is to replace the disc, based on defective materials or faulty workmanship, but not based on the operation or functionality of the product.

*To Reni.
Thanks for everything.*

CONTENTS

<i>Acknowledgments</i>	xvii
<i>Introduction</i>	xix
Chapter 1 Introduction to How Games Work	1
In This Chapter	1
Aspects of Interesting Games	2
Venue	2
Conflict	3
Mechanics	4
Graphics and Sound	5
Props	5
Interface	6
Pace/Scale	8
Fidelity/Accuracy	9
Game Architecture	9
Presenting the Virtual Universe	10
The Audio System	15
Playing the Game by the Rules	16
Game State	18
Making a Game Function on a Computer	22
An Example - <i>Hockey Pong</i>	22
The Game Loop	24
Front-End/Back-End	31
A Sample Game Design Document Hockey Pong: High Concept	32
High Concept	32
Features	32
Player Motivation	33
Genre	33
Target Customer	33
Competition	33
Unique Selling Points	33
Target Hardware	33
Design Goals	33
Story	34
Objects/Characters	34
Puck	34
Paddles	34
Timer	34
Face-Off	34

Summary	34
Exercises	35
Resources	35
Bibliography	36
Chapter 2 Graphics and Images in Processing	37
In This Chapter	37
Review of Basic Processing	38
The <i>PImage</i> Type	39
Image Files	43
Graphics Interchange Format (GIF)	43
Joint Photographic Experts Group (JPEG)	43
Tagged Image File Format (TIFF)	43
Portable Network Graphics (PNG)	44
Comparisons	44
Pixels and Color	45
Getting and Setting Color Values	45
Drawing Shapes	47
Lines and Color	48
Hockey Pong	49
Text and Fonts	50
Internals of the Type <i>PImage</i>	51
Image Size	52
Accessing Pixels	53
The PGraphics Type	55
Rendering to a <i>PImage</i>	57
Summary	57
Exercises	58
Resources	59
Bibliography	59
Chapter 3 Sound	61
In This Chapter	61
Basic Audio Concepts	62
Introduction to <i>Minim</i>	66
Declaring and Initializing <i>Minim</i>	66
Using <i>Minim</i>	67
Playing Sounds	67
Snippet	67
Sample	68
File	69
Display of Synthesized Sounds	70

Monitoring Input	72
Recording Sounds to a File	73
Minim and Java Sound Support	74
A Processing Game Audio System	75
Positional Audio	76
Example: Distance Attenuation	77
Example: 2D Positional Sound	78
The <i>simpleAudio</i> Class	80
Using <i>simpleAudio</i>	81
Exercises	83
Resources	84
Bibliography	84
Chapter 4 Hockey Pong: A 2D Game	87
In This Chapter	87
Implementing the Game Using Prototypes	88
Prototype 0	88
Prototype 1	89
Screens	90
Buttons	91
Start Screen	91
Options Screen	92
Play Screen	93
End Screen	94
Prototype 2	94
User Control	94
Sound	95
Ambience	95
Artificial Intelligence	97
Prototype 3	100
A Timer Class	101
Game State 0: Start the Game	102
Game State 1: Start a Period	103
Game State 2: Face-Off	103
Game State 3: Play	103
Game State 4: End Period	103
Game State 5: End Game	103
The Scoreboard	104
Testing	105
<i>Hockey Pong</i> Game Design Document	106
Version history	106
Table of Contents	107

Game Overview	107
Game Concept	107
Genre	107
Target Audience	108
Game Flow	108
Look and Feel	108
Project Scope	108
Locations	108
Levels	108
Non-player Characters	108
Gameplay and Mechanics	109
Gameplay	109
Objectives	109
Mechanics	109
General Movement	110
Other Movement	110
Picking up Objects	110
Moving Objects	110
Screen Flow	110
Main Menu Screen	111
Options Screen	111
Play Screen	112
Exit Screen	112
Option 1: Two Player/One Player	112
Option 2: Music On/Off	112
Option 3: Home Team Selection	112
Replaying and Saving	113
Cheats and Easter Eggs	113
Story, Setting and Character	113
Game World	113
General look and feel of world	113
Characters	114
Levels	114
Interface	115
Control System	115
Audio	115
Music	115
Sound Effects	115
Help System	116
Artificial Intelligence	116

Opponent AI and Collision Detection	116
Technical	116
Target Hardware	116
Development Hardware and Software	116
Development Procedures and Standards	116
Game Engine	116
Network	116
Scripting Language	116
Game Art	116
Concept Art	116
Style Guides	116
Characters	116
Environments	116
Props (Tools, Equipment)	117
Cut scenes	117
Miscellaneous	117
Development Software	117
Editor, Installer, etc.	117
Management	117
Schedule (From start of project)	117
Budget	117
Localization, Risk, Special Considerations	118
Test Plan	118
Appendices	118
Asset List	118
Model and Texture List	118
Interface Art List	118
Sound	119
Interface Sounds	120
Music	120
Voice	120
Summary	120
Exercises	120
Resources	121
Sound Effects	121
Sound Editing	122
Graphics Editing	122
Chapter 5 Graphics in Three Dimensions	123
In This Chapter	123
Using Polygons to Build Objects	124
Storing and Drawing Polygons	125

Introduction to <i>OpenGL</i> Graphics in <i>Processing</i>	127
Triangles	127
Viewing	130
Projecting the Image of a Scene onto a Plane	131
Perspective Projection in Processing	133
Viewpoint	136
A Prism	139
Geometry: Translation	141
Geometry: Scaling	143
Geometry: Rotation	144
Colors, Shading, and Textures	145
Texturing	146
Theory	150
Object Models	153
PShape	155
Summary	156
Exercises	156
Resources	157
Bibliography	158
Chapter 6 Game AI: Collisions	159
In This Chapter	159
Collision Detection	161
One-Dimensional Collisions	161
Two-Dimensional Collisions	164
Broad Phase Collision Detection	166
Operational Methods	166
Geometric Tests	167
Using Enclosing Spheres	169
Using Bounding Boxes	175
Space Subdivision	178
Narrow Phase Collision Detection	180
Ray/Triangle Intersection	181
Collision Detection Packages	183
<i>Vclip</i>	183
<i>ColDet</i>	184
<i>I-Collide</i>	184
<i>Swift</i>	184
<i>Rapid</i>	184
Summary	184
Exercises	185
Resources	186
Bibliography	186

Chapter 7 Navigation and Control	189
In This Chapter	189
Basic Autonomous Control	190
How to Control a Car	191
Cruising Behavior	192
Avoidance Behavior	193
Waypoint Representation and Implementation	195
Finite State Machines	196
FSA in Practice	198
State and the ‘What Do We Do Now’ Problem	200
Other Useful States	201
Start	201
Air	202
Damaged	202
Attacking	203
Defending	203
Searching	203
Patrolling	204
Skidding	204
Stopping	205
Pathfinding	205
A* Search	207
Stochastic Navigation	211
Summary	213
Exercises	213
Resources	216
Bibliography	216
Chapter 8 A 3D Game Example	217
In This Chapter	217
SMV Rainbow	218
Screens	219
Game Environment	220
Three Dimensional Assets	221
Cargo Containers	222
A Cargo Vessel	224
The Bridge	225
Heads-up Map Display	226
Radiation Sensor	228
The Indicators	229
Controlling the Submarine	230
Illumination	232
Sound	232

3D versus 2D Rendering	233
3D versus 2D Screens: Popups	234
Summary	236
Exercises	236
Resources	237
Bibliography	237
Chapter 9 The Web and HTML5 Games	239
In This Chapter	239
How the Web Works	240
HTML and HTML5	241
JavaScript	243
HTML5	245
<i>Processing.js</i>	246
Making a <i>Processing.js</i> Program Work	247
Hockey Pong	249
Integers in JavaScript	251
Preloading	252
Sound in <i>Processing.js</i>	253
Fonts	256
More <i>JavaScript</i>	257
<i>Processing.js</i> Development Tips	258
Summary	259
Exercises	260
Resources	261
Tutorials	261
HTML	261
HTML5	261
JavaScript	261
Minim JavaScript Emulation	261
Downloads	262
Bibliography	262
Chapter 10 Animation	263
In This Chapter	263
Creating Elementary Animations	264
Animation Math	269
Motion Equations	271
Reactive Animations	274
Using Real Images	277
Ambient Animations	279
Character Animation	283

Cut Scenes	284
Summary	286
Exercises	286
Resources	288
Bibliography	288
Chapter 11 Android Handheld Devices	289
In This Chapter	289
How Cell Phones Work	290
<i>Android</i>	291
Targeting <i>Android</i> from <i>Processing</i>	292
Getting Set Up	293
Getting Through the Installation Issues	295
Common Problem #1	295
Common Problem #2	296
Common Problem #3	298
Common Problem #4	298
Your First <i>Android</i> Program	299
An <i>Android</i> Puzzle: Touch Events	300
Playing a Game on a Real Device	302
Exporting an <i>Android</i> Game	304
Sound in <i>Android</i>	308
Installing APWidgets	308
Using the APMediaPlayer	308
Summary	311
Exercises	312
Resources	312
Bibliography	313
About the CD-ROM	314
Appendix A: Mathematics Tutorial for Game Development	A.1–A.23
Appendix B: A Processing Primer	B.1–B.10
Index	319

ACKNOWLEDGMENTS

This book has taken longer to write than anticipated, but would have taken even longer without the help of a few key folks. Thanks go out: to Matthias Kabel for his panoramic image; to Herminio Nieves for the use of his Cycle model; to NASA for having the generosity to allow their images to be used by others; to my old friends at Radical Entertainment for g-etting me started, to Damien Di Fede for *Minim*; and, of course, to the developers of *Processing*: Ben Fry, Casey Reas, and a host of others.

INTRODUCTION

Games have been a subject of fascination for students for well over a decade. There are now thousands of people studying video game development at universities in North America alone, either as part of a formal game development degree or as a part of another program. Games can be used to teach, to convince, to sell, and simply to enjoy, and all have the same basic underlying structure as software and as media objects. Any computer game has the same essential structure: they use images and sounds, have objects that interact visually on the screen, and repeatedly update the display many times per second according to the simulated situation. There are some very advanced systems for game development available on the Internet, some being free and some for sale, but underneath they all help accomplish very similar tasks.

This book is intended to provide game development students with the insights into how games function that are an essential starting point for further study. Each of the chapters focuses on a particular aspect of game development, from graphics to audio to artificial intelligence to animation, and does so in a very hands-on way. For people with little programming background Appendix B gives a quick introduction to *Processing* with some basic examples. Appendix A illustrates some of the common mathematics techniques used in building games. There is a lot of code on the companion disk, including three complete games: a 2D arcade style game named *Hockey Pong*, a 3D submarine game named *SMV Rainbow*, and a puzzle game for Android. These are examples, but are fully functional and show how to deal with many of the typical problems one would normally encounter in a game development project.

The language *Processing* is used because it provides a visual output very easily – it was designed for art students – while having access to the full power of Java. Almost anyone in a development team could learn *Processing* and appreciate the game principles that are described using that language. It's also possible to build quite interesting games in *Processing*, games that can run on personal computers, on the Internet through browsers, and as Android apps.

The subject of “gamification” is not pursued in this book specifically, but the material here can be used in implementing gamified interfaces and systems. After all, running *Processing* on a Web page is spectacularly easy.

A computer game is a microcosm of the discipline of computer science. A game contains within it many of the problems one would study in a computer science degree program, and a small set of games could encompass nearly ev-

ery concept a CS degree would present. It's an entertaining way to learn the subject, not just programming but artificial intelligence, interfaces, graphics, and design. So read, modify the code, add new art, and have fun – if you are not having fun, you're doing it wrong.

Jim Parker
April 2015

CHAPTER 1

INTRODUCTION TO HOW GAMES WORK

In This Chapter

- What makes games interesting
- An overview of game architecture
- A sample design document

It has been estimated that 69% of the human beings on this planet have played a video game. This amounts to about 5 billion people. This seems a reasonable number given the popularity and ubiquity of games as observed in shopping malls, in movies and in media. What these players know about games varies quite a lot, but common knowledge includes rules of specific games, interface issues like what keys to press or buttons to push, and some tactical information about game play, like where to hide, when to jump, and so on. Very few people know how the games work at an implementation level—the internal actions of the game, from key press to avatar motion.

You can infer some internal structure by examining what people *do* know about games. That players know rules of games implies that games have a consistent set of rules that will be implemented by a computer program. The players' knowledge of the user interface implies a level of consistency there too. We observe, for example, that the arrow keys, or sometimes

the keys W, A, S, and D, are used to move the player's avatar, but not the keys R, G, N, and M; there are conventions for user interfaces. The video game screens display images that move, and game actions result in sounds that the player can hear and use as cues. Thus, we can infer that a video game must be able to display images and sounds, and do so in response to user commands or internal events in the game.

On the other hand, we need to know significantly more to design and build an original game. There's only so much that you can learn from examining a game from the outside. Games are complex systems involving computer devices and software, art (textures, 3D models, sprites), music, sound effects, video and animation, story, and a designed structure for play. There are a great many existing games, and techniques and tools exist that will help a game developer in their task have evolved over the past few decades. If we're going to build a game, we should become familiar with at least some of these things.

A good way to begin is to look at what aspects of design make for a good game, and then take a close look at how a computer game is structured at the design and implementation level.

Aspects of Interesting Games

Games that have had an enduring quality tend to have common features that should be taken into consideration when designing a game. Genre is key in determining which features are appropriate for the game, and there can be a significant amount of entertainment value in manipulating things like venue and conflict.

Venue

Some games take place in interesting locations, often real places that you may have visited, such as San Francisco, New York, or Paris. People like to see new and interesting places, and people who have been to those places like to see spots they can recognize within the game. Of course, many people have seen these places in movies and on television, so that gives the sites an extra degree of familiarity. This is important, and not just because the locations are often distant and exotic to many of us. For instance, if we could create a game that would drive through your own home town, that might interest you. The makers of the *Monopoly* game have done this by customizing their game for various cities, and it seems to work.

Naturally, simply placing the action in an exciting place is not good enough—you must portray that place using graphics and sound well enough so that the player can identify it. This is an implementation issue more than a design issue, but it is important, and will be discussed in more detail.

If the game is a driving game, then placing it on a racetrack is an obvious thing to do. But this can actually take away from the fun *unless* that track is familiar to the player. For example, the track at Indianapolis is well known, and a racing fan would identify it in a moment. In a basketball or hockey game, the designers and implementers go to significant effort to try to make the players look and act like the real players, and have the venues be accurate representations of the real thing. Fans know their sport, and can be very critical when the game does not look right.

It is a little unusual to permit the player to get too far away from the path that the game designers have set out. In some games you can go anywhere in the simulated world and manipulate objects —this is an essential aspect of play in *The Sims*, for instance. Such games are sometimes called *sandboxes*, because the player, rather than the designer, defines the way the game is played. It is a feature of most games that the player appears to have much more freedom than they really do; they appear to have choices, but for the sake of simplicity the choices all seem to lead to the same few consequences. This appearance of freedom is necessary because we just can't simulate the entire universe.

Not *yet*, anyway.

Conflict

Games most often have winners and losers; games keep score. Clearly the goal is to win, and any game that does not allow the player to win will not be popular. This is the minimum conflict requirement of any game, even ones that involve collaboration. Perhaps the term *challenge* should be used in place of conflict. In any case, a game presents the player with obstacles to overcome, and success is a matter of learning how to deal with these obstacles.

There are a variety of conflict sources in games, and variety can be essential to game play. Indeed, any narrative depends on a degree of conflict to be interesting. While narrative in some games is usually there only to provide background, or an excuse for the action, other times the conflict is fundamental to the story, as it defines both the goals and the means of achieving them.

So, what kind of conflict are we talking about? First person shooters have one of the obvious forms of conflict, where the goals are achieved by shooting your opponents. Sports games also have obvious conflicts, as do racing games: to defeat an opponent using the rules of the sport. In *Crazy Taxi* we have customers refusing to pay if we get there late, and they berate us for going too fast or slow. In the *Tycoon* series of games (e.g. *Zoo Tycoon*) conflict is created by forcing the player to balance costs and income through building attractions. *Little Big Planet* is a cooperative game in which the players can create new content, but there are still goals to be achieved that present challenges; it lacks the incentive to defeat an opponent, and so is in that way not competitive. Not all games use competition as conflict, but most do.

Mechanics

The subject of *game mechanics* is going to come up again and again, so a good definition would be useful. Unfortunately this term is more generally understood by designers, but is hard to define for people who don't build games, and impossible to explain to people who don't play them. Most definitions of game mechanics are somewhat circular—games use mechanics and mechanics make a game. But perhaps it is better to think of mechanics as a way of guiding a player through the game. *Game mechanics are design features of a game that allow the player to progress towards the goal.* There can be mechanics that interfere with forward progress too. Examples include a *dodge* in which the player must avoid objects moving towards their avatar—such as in *Space Invaders*—a *shoot* where you must hit your opponent or a target with a thrown missile, and a *race* where a player must get to a particular spot in the game before any opponents.

Game mechanics should be interesting to the player and simple to use. A button press or mouse gesture is usually all that is possible in a game, so these must be converted into game elements that implement the mechanics—such as pressing space to shoot, or the up arrow to move ahead. The mechanics are not specifically about the interface, though. One of the better game mechanics is the one used in the game *Portal*, where, using a mouse click, the player creates two circular doorways through any floor, ceiling, or wall—they can then walk the avatar into one doorway whereupon it exits through the other. This can be used for quite complex maze puzzles. The mouse click, however, is not the issue—it is about the possibilities created by the click.

Graphics and Sound

The appropriate graphics and sound can make a significant difference in the player's experience.

The graphics do not have to be high resolution and three dimensional, but they do have to be appropriate for the specific game. Games like *Halo* offer high quality graphics and animation, but we should not mistake *high resolution* for *interesting*. *The Simpsons Hit & Run* offers the players the ability to explore a town that they know more or less well—Springfield, home of the Simpsons—which is and has always been a cartoon. Do not underestimate the value of the *sandbox mode*; if the world is interesting then just looking around can be entertaining.

Sound may be more important than graphics, especially for imparting mood, including excitement. An intense music, such as rock, as an audio background adds energy to a game in the same way the ethereal themes from *Half Life* make their world seem dangerous and spooky.

It is not usually a trade-off, and we can have good graphics and audio if we just have the wherewithal to create it.

Of course, some games want better graphics, because they need to have a more faithful simulation, whereas others can be very cartoony, like *Double Dash*, and still offer a huge degree of entertainment value.

The best games have in common an appropriate level of detail in graphics, with good audio, appropriate and entertaining objects, and backdrops rendered predictably.

Props

Props are items that can be manipulated in the game. All games have props of some kind. Sports games in general have fewer props than most games, and simulation games often have many. A ball is a prop, so is a bullet or a missile, and they can move on their own. Power-ups and penalty objects are props too.

Props have an immense potential for making a game more interesting. Without the possibility of picking up weapons and speedups, the *Double Dash* game is just a cartoon race game. Being able to slow down an opponent from a distance, and having him be a threat from behind, adds an element of excitement.

Props also help create a more interesting narrative, since entire missions and levels can depend on moving props from one place to another. They can impart important properties that remain from level to level, like magic icons, fuel, and skill points. This leads to the conclusion that complexity makes a game more interesting. There is some truth to this, so long as the game remains playable, but it is more likely that a degree of unpredictability makes the game more interesting. The game must be consistent, but it is best if it does not repeat itself exactly each time it is played. Props can be used both to make a game more complex and to make it less predictable. The location of the props can change, and create a new set of challenges. A pattern of play can be useful sometimes, but it's rarely entertaining for long.

Interface

Games are fundamentally an interactive medium, and interfaces are at the heart of interaction. There has been a significant effort to standardize some game interfaces. For example, on consoles we find very similar controls doing similar jobs, especially on games that run on multiple consoles. Games of a specific genre tend to use consistent control sequences, such as arrow keys for motion.

Games can vary on use of keys/buttons and mouse gestures, and there is still too much variation for identical mechanics. Some racing games use the mouse to control speed and direction. The mouse has more degrees of freedom in directional control, but the faster the car goes, the harder it is to control with a mouse. One tends to over-steer, and mouse position is relative, not absolute. Still, given the popularity of games for tablets and smartphones, the use of the mouse is increasing. The touch gestures on tablets translate into mouse clicks and motions directly, and since most tablets and phones don't have keyboards, using keys is a bad idea.

There are now many special game interfaces available at low cost, most of which use the USB interface. This is much better than the old parallel port or the game port that used to be available on PCs in that the USB standard allows for faster data transmission, and has no need for special driver software in most cases. The game port was almost always used to plug in a joystick - a curious term for a game controller - whereas there is a wide variety of USB game controllers that can all be used interchangeably.

One such special purpose interface is the steering wheel and pedal set

that converts wheel and pedal motions into character sequences. The fact that they can be configured to send any sequence you like means that game interfaces no longer have to be standard, at least in the long run. More and more people will acquire the special interfaces until the keyboard becomes old fashioned. Figure 1.1 shows one particular brand, Logitech.

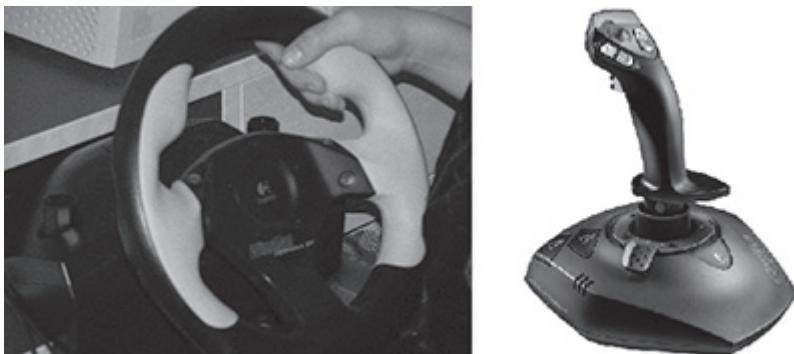


FIGURE 1.1 Special purpose game controllers for specific genres. (a) A driving controller that emulates a steering wheel. (b) A flightstick for aircraft and other vehicle simulations.

Another USB interface device is the flight stick; the idea is to make the interface look more like that of an airplane by giving it a similar control device. The Logitech *Wingman* shown in Figure 1.1 is one such device, which can be configured to provide specific control sequences for any possible action.

A problem is that there is no ‘feel,’ in that the action is from the player to the game only. In a real airplane or car, the control—wheel or stick—provides a force that counters the player’s actions, and that force is relative to the speed, direction, driving surface, or even wind. Few game controllers do this. Some interface devices can vibrate, which gives limited feedback. The *Reactive Grip* controller simulates responses of various weapons, like swords and guns, and is probably the best current example of a reactive device. (Provancher 2011)

There are many other possibilities, and progress is proceeding rapidly. At the 2013 Game Developers Conference, for example, there was an interface for sale that inputs brain waves from a couple of small electrodes attached to the skull, and uses the signals to play some simple games. The price? \$100.

NOTE

There is also university research that includes some work on hand gesture recognition as applied to controlling games on the PC—hand motions are interpreted as requests to move, pick something up, and so on. Ultimately games will become an invisible technology, like telephones and TV. They will be anywhere we like, and will require no special knowledge or hardware to play.

Pace/Scale

The games that are the most fun typically allow you to go *fast*. The pace of the game must be appropriate for the kind of situation simulated by the game. For instance, submarines move slowly but inevitably, and cannot turn or stop in any reasonable time. In addition, as the player gains experience, the game should present more and more difficulties. In some games, this means that the game speeds up; other times there are more or stronger opponents.

An impression of speed can be given by placing objects near the player that move past at a high speed. Buildings are good for this because they have a lot of detail that flashes past in a similar way to what we'd see on movies or real life. In fact, a variety of objects in a range of distances is effective in conveying the illusion.

Sound is crucial as well. Play a fast tempo game with the sound off to notice the difference. A fast moving music track helps a lot, as does a good set of speedy sound effects: positional sound, Doppler effects, and so on.

Using a backdrop with scenery painted on it is a simple and effective way to create an environment. This effect is now a standard in games and is used in movies too. Quite a lot of science fiction depends on paintings to convey distance and strange environments. For example, the shots of the inside of Borg ships on *Star Trek* are actors in front of paintings. The use of backdrops (also known as drops) in a game can add a depth that is noticed if absent, but usually draws no comment otherwise.

A 3D game consists of a terrain model on top of which we have moving and stationary objects. There is usually a distance beyond which moving objects will not be rendered (the *far clipping plane*), and as objects get closer, they seem to *pop* into existence at that point. In the far distance, we have a drop, a stationary image painted on a surface that passes for the horizon. This often has hills and sky, or an urban backdrop. In any case, the drop has

no real depth. A game will have the drop rotate as the player turns so as to give the illusion of a change in direction.

Fidelity/Accuracy

When we create a game, we create an entire universe. We get to decide where things are, how big they are, what the characters eat, and so on. In particular, the rules of physics as we understand them in the *real* universe are flexible, and we decide how they work in *our* universe. In many kinds of games, the accuracy of the physics takes a back seat to playability and entertainment value. If you have ever played *Doom*, you will probably have noticed that the player can seem to run pretty quickly through a level. If you measured it, you would see he has a top speed of 60 MPH. This does not detract from the fun; on the contrary, restricting game objects to normal speeds would slow the game down a lot.

How physics are addressed in a game includes how collisions are handled, how fuel is consumed, how fast projectiles can accelerate and what top speeds are, how fast vehicles can enter a turn before they skid, and how a vehicle can become airborne if it reaches the peak of a hill. Most games take liberties with physics to enhance game play, especially the more cartoon style games. Just remember that any rule can be violated if it makes the game more fun.

Game Architecture

Game architecture is be about the internal structure of a game, its general organization as a functional system in terms of the way that the parts are arranged to create a working game.

In order to truly understand the structure of a game, you have to know something about computer programming, because the computer is the enabling technology, and any computer game is a piece of software at its core. It is only possible to have a general appreciation of how a game functions without being a programmer. You need to know what the parts are—not the visible parts like cars and roads—but the structural and functional parts, like the audio system and the renderer. You also need to understand how the parts communicate with one another—what each part needs to know to accomplish its task.

A game player cannot be required to know this. The player needs only to know what they need to *play*—the rules of the game, the task, the in-

terface. In fact, game design students often say that after studying games and writing one, they never look at a game in the same way. They still play games, but find themselves asking, *why did they put that building there?* and *How do they implement those torches?* Knowing how games function *under the hood* can sometimes interfere with, or perhaps enhance, the experience of playing them.

In a technical sense, a computer game is an interactive real-time simulation with a graphical and audio display. If you accept this definition, then there are already a number of identifiable components that comprise the game system: the graphics system, the audio system, the user interface, and the scheduler. The final missing part is the artificial intelligence (or AI) whose job is to keep track of the simulated objects in the game.

Figure 1.2 shows a diagram of the basic components of a game and how they are connected. It is not the only organization, and certainly does not show too many details, but it should be good enough for now, and will form the basis of this discussion. So, the remainder of this chapter will describe each of these components of a game and how and what they communicate with each other. This will give you a much clearer idea of how the overall game functions, as a software system.

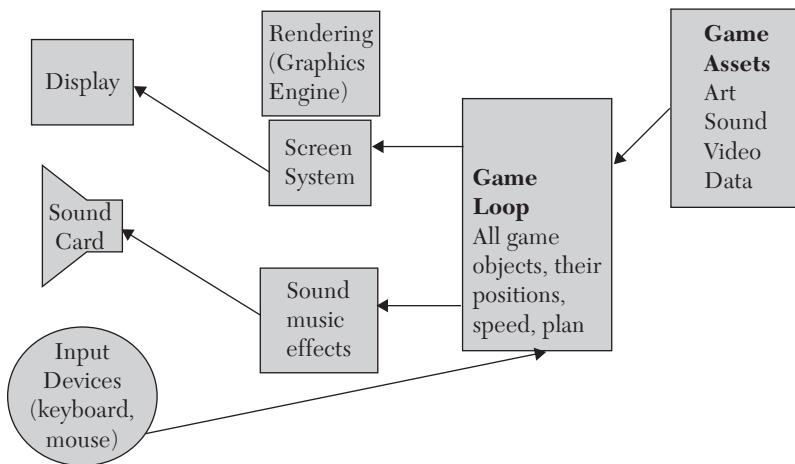


FIGURE 1.2 General Architecture of a computer game.

Presenting the Virtual Universe

A computer game offers the player a world that does not really exist. Without getting too philosophical about it, what you see through the com-

puter screen is a rendition of data that represents a simulated situation. What you see is a real screen with real images, but the situation and what is being drawn do not exist in the real world—it is an analogy, a virtual environment in which you control the laws that dictate how objects interact (for instance, gravity or the results of a collision). A significant part of a game, in terms of code and time required to create it, is the part that displays images and sounds from the imaginary world for the player to evaluate.

Before proceeding, it should be noted that there are many types of game, and that each has its own specific needs. This means that the viewpoint, or the place from which the simulated universe is seen, will vary from game to game and genre to genre, and so the discussion must be sufficiently general to allow many perspectives. Many games that are played online, through browsers, are effectively 2D (two-dimensional) and so the discussion will focus in Chapter 2 on ways to render 2D games effectively.

In a 3D environment, we perceive the game universe from our particular point in space and project the view from that point onto a 2D plane for display on the screen. What is going on a great distance away is unlikely to be relevant to us, and so the graphics system might not bother displaying it. The region ahead for the next few feet or meters or yards is crucial, and how we respond to that will influence what is displayed following that. The important thing as a game programmer is to display the things that the player needs to make game-play decisions and to feel that the simulated world is real. The display of key data involves two main aspects: visual data, requiring a computer graphics system, and sound data, which requires an audio display system. Most games have both of these things.

The Graphics System

Many people still think that games are all about graphics. They are not, really, but many games use more CPU time in drawing the scene than anything else. An efficient graphics system can leave a reasonable number of CPU cycles for use by other aspects of the game system, and that is very important. A good game uses an appropriate level of detail for the application, and that's important too. The basic problem addressed by game graphics systems is placing enough frames (images) on the screen every second to give the illusion of motion and realism. Movies use 24 frames per second to achieve their degree of realism, while television uses almost 30 per second. However, a traditional television displays an image that is about 525×525 pixels, while a motion picture has a much higher resolution.

Another aspect of picture display that must be considered is the number of distinct colors that can be shown. This is sometimes called *quantization*. Television, for example, can display far fewer colors than a motion picture, and a computer screen falls in between.

Some simple math: at 24 frames per second, with a computer screen having a resolution of 1024×768 and using 24 bit colors, we need to be able to calculate and write out 56 Megabytes of data per second. This seems like quite a lot, even on a modern PC, so we have to use a few little tricks.

First, and most importantly, the video card has been taking a larger role in the calculation of screen updates for games. It can draw millions of polygons per second, and can do more advanced and esoteric operations like texture mapping, support for stencil buffers and *mip-mapping*. This means that the CPU does not have to do these things, but simply organizes the data for the video card.

The fundamental differences between 2D and 3D games can be summarized as follows.

In a 3D game, all objects are 3D and need to be viewed from a particular point in space. The graphics system will flatten the scene by projecting it onto a flat surface. This means that some objects will be hidden by others, some will be too far away to see, some will be behind us and so not visible, and all objects will be transformed so that the usual visual cues will apply for the viewer (player). This last item usually means something called a *perspective transformation*, in which objects farther away from us appear to be smaller, and parallel lines appear to meet at a distant point. For example, in a driving game, if we are driving a car and looking through the front windshield then our field of view is restricted to the region in front of us, say 60 degrees each side of dead ahead. Objects that are not in that region can be ignored, and should not require any significant amount of computation. Objects that are too far away are also to be ignored, as they will be too small to see. Of course, figuring out what can and cannot be seen requires computational effort.

In a 2D game, we usually view all objects from the side or from above. Sometimes the game-play area is bigger than the screen, and the background scrolls as the player's character moves about. Objects in a 2D game are simpler and easier to draw, and perspective is not an issue.

Some of the work in drawing the views must be done by the game programmer, but much of it can be handled by the graphics card. There are

quite a few graphic cards available, and each has their own capabilities and interfaces. If you want your game to run on more than one computer, you cannot code your graphics system for a specific device. Fortunately, there are software packages that form a layer between the programmer and the graphics card, hiding the differences between the cards while presenting us with a consistent interface. This is essential for a commercial game, and is important for us too.

The *Processing* language has built-in support for the major such package *OpenGL*, and we will depend heavily on what *Processing* supplies for graphics support, generally 2D and 3D. Since *Processing* runs on all major operating systems, this means that a game developed using that language is playable effectively everywhere. The thing to remember about most game graphics systems is that the 3D systems are based on polygons, since polygons can be drawn quickly using a graphics card. We can represent any object as a collection of polygons, as well as shade them, place textures on them, rotate and scale them, and so on using very fast algorithms.

In game programming books, you will frequently encounter the phrase *graphics pipeline*. The idea is that if you can keep a number of software modules busy at the same time, you can achieve an increase in the number of polygons you can process per second. There are a few ways that the pipeline can be organized, but here the view will be taken that there are three basic parts: the *object* level, the *geometric* level, and the *rasterization* level.

Object Level

At this stage, the objects are still understood as such, rather than as collections of primitive graphic entities like polygon and lines. We do animation at this level, and morphing, and collision detection—basically any operation that needs to know about the objects themselves. At the end of this phase, a set of polygons or lines is sent to the geometric level.

This part of the pipeline is the most sensitive to the game itself. It is implemented in software, most often by the game designers and creators because it is they who understand the game objects best.

Geometric Level

The geometry part of the pipeline has a variety of functions that can be broken off into distinct modules, as seen in Figure 1.3. Geometry in 3D is much more complex than 2D, and the figure illustrates the more complicated situation. The first step converts model based coordinates—which are often

based on an object-centered coordinate system—into a more global system of coordinates so that objects can interact.

Next, based on the position of the viewer (or camera) we compute a coordinate transformation that aligns the polygons of the objects to a common system based on the viewer. One result of this is that some polygons become impossible to see; they may be behind us, or too far away.

Now we consider the position and color of the lights, and create appropriate shading and color transformations of the object's polygons. For example, the sun is positioned a great distance away and is colored yellow-white, while a nearby headlight might be a brighter blue. The color of a pixel is a function of its own intrinsic color and of the brightness, color, and position of the illumination sources.

Now we compute the viewing transformation, most often a perspective transform. This gives us the view we would expect of a three dimensional object, including the fact that distant objects look smaller than near ones. The view of the scene will be realistic if it represents what we expect, and we expect a perspective view. The objects that used to be 3D polygons are now 2D.

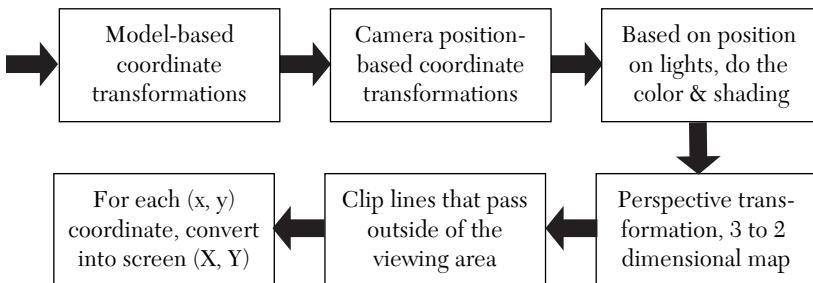


FIGURE 1.3 Geometry calculations for a 3D game.

The polygons that fall outside of the computer screen area, or viewing area, must be eliminated, or *clipped*. This is the next stage. Polygons that are too close or far away would have been clipped in the previous stage. Clipping is a non-trivial operation. For example, a triangle that is partly outside of the screen area is cut by a vertical or horizontal line, and this often means that it is not a triangle anymore. Finally, all coordinates of all lines and polygons are converted into screen (or window) X,Y coordinates so they can be drawn quickly.

Rasterization

In this stage, lines and polygons are converted into pixels. The only thing that a screen can display is pixels, so it is essential that this step be performed accurately as well as quickly. After this is done, we can perform any other operations that need to be done on a per pixel basis. Much of this is done by the graphics card.

Comments on Optimization

It should be obvious that the code must be efficiently written, because the graphics system must render a sufficient number of frames per second that the game appears to be smooth. The algorithms we choose must be able to deal with the number of polygons likely to appear in the objects, in both space and time considerations.

The game we are going to create will, first and foremost, have to display scenes on the screen with the correct positions and colors, follow the game rules as designed, and play sounds at correct moments. While we will not be intentionally wasteful, efficiency will not be the most important thing discussed in this book. There are many reference works on the subject for those interested, including *Game Coding Complete* and *Core Techniques and Algorithms in Game Programming*.

The Audio System

In general, the purpose of the game's audio system is to play music and sound effects. This is supposed to be a simplistic view, and yet even after decades of technological changes in game technology and design, the game audio system still does pretty much what it always did, and still works in a similar way. Huge steps forward have been made in the area of graphics, but audio standards are still weak and there are many ad hoc schemes out there.

Most sounds that we will need—like an engine sound, a door opening and closing, crashes and scrapes, even the music—will be read in from files, usually one file per sound. A very common format for sound files is the WAV file, basically a Microsoft standard that is supported on most platforms. These can contain compressed or uncompressed audio, can have mono or stereo, and can store audio at a variety of sample rates, including the CD standards. It is a simple file format and very convenient for our purposes.

However, there may be a need for more sophistication from the audio system. It may be asked to play positional audio, in which each sound appears to originate from a particular point in space. This can be done with stereo, but is much better suited to modern 5.1 channel audio systems, and can be truly impressive. Sound cards have recently been designed with some capacity for sound synthesis, and some degree of synthetic sound and music, especially using MIDI, can now be found on games. However, the basic function of the audio system is simple, and its job is obvious—we need to play and stop a sound on cue. How this is done will be the subject of Chapter 3.

Playing the Game by the Rules

What we have seen of game architecture up to this point is what can be called the *game board*, the part that the user sees and manipulates. There are many programs that have very sophisticated graphical and audio interfaces that are not games. What's the difference? A game, the kind of game we are discussing, works in real time, processing user choices and updating the display accordingly. A game is a simulation and, most importantly, is one that has a goal. To be a game, there must be a way to keep score at least, and usually there is a way to win. The part of the game program that does this is called the *game logic* section, or the *artificial intelligence*, and some other things too.

The graphics and audio parts of a game can largely be shared in between games that are quite different from each other. The game logic is what makes each game what it is. It is the code that reflects the game designer's intent. Even here, the structure of this program has a certain consistency from game to game; it is in the details that the code differs.

Most of a Computer Game is Hidden

The player sees a world drawn by the graphics system, but this world is particular to the game at hand and changes according to rules that are largely invisible to the player, at least at first.

Part of the game play is figuring out what the rules are. For example, how fast can someone drive into that corner before skidding? That is a kind of rule. From what height can your avatar leap without causing damage to itself? Again, this is a rule that is discovered. The number of damage points you can take before your character dies, that's an *explicit* rule that is stated up front, a slightly different thing. Most game players don't read a lot of

rules before starting to play, and discover a lot of the explicit rules as they go along.

So when we say that a lot of the game is hidden, we mean that the rules, interrelationships between game objects, goals, and even your particular progress through the game are saved in code and internal data structures, and are not necessarily displayed. Indeed, their internal representation does not lend itself to display.

The Artificial Intelligence

The AI subsystem of a game is responsible for many things that the game does that are not seen directly, but are reflected in game-play and realism aspects. The AI does object management, including physics, and the direction of independent simulated objects like opponents. Specifically, the AI keeps track of the current position and velocity of all objects. Thus, it is the logical place to do collision detection. It keeps track of attributes of objects, including earned attributes like hit points, damage, and found objects (such as ammunition and money).

Artificial Intelligence has a connotation among the general population, supported by movies and TV, of computers that can think. Computer scientists and programmers know more about the details, and they realize that AI is about making a computer *appear* to be intelligent. The techniques that computer professionals use are many and varied, but the truth is that game AI is very simplistic compared with the techniques found in research labs, and the goals are quite different too.

The basic problem is that the game AI has to function in real time, and must steal CPU cycles from what is perceived to be really important—the graphics system. Thus, the really complex and exciting functions of an advanced real AI system are simply too time consuming most games. It is a good thing they are mostly not required. For example, a voice recognition system would be an interesting feature on some games, but really isn't needed. Game AI is rarely required to prove theorems, recognize faces, or invent novel answers to complex questions. It is required to decide what to do next and to plan a route through a building or a forest to a goal. Maybe it will have to decide how best to pass you on a hairpin curve. Although it is sometimes useful to use an advanced technique like a neural network to accomplish a game goal, it is unusual.

However, one thing that all AI systems must do is keep track of everything on the screen. Not only does the game have to decide when you collide with an object, but it must also keep track of all of the other players—even the ones you cannot see—and make them behave naturally after a collision too. Most of the AI system is about simple rules implying simple choices. The most common implementation of such a choice is:

```
if (condition is true) then { do this thing }
```

This is not especially sophisticated, but it does the job quickly. The same thing can be implemented as a table or a tree, as you will see in Chapter 6.

Game State

The state of a game is collection of information that represents the game at any given time. Given the state, a game can be started from that point. The information needed in the state includes:

- Position, orientation, velocity of all dynamic entities
- Behavior and intentions of AI controlled characters
- Dynamic, and static attributes of all game-play entities
- Scores, health, power-ups, damage levels, etc.

All subsystems in the game are interested in some aspect of the game state, because the state variables are exactly those things that are essential to the look of the game and the play options possible from any point. For instance, the renderer needs to know the position of objects to draw, their damage levels, and so on.

How is the game state made available to subsystems? As always there are many options, each with their own advantages and disadvantages, but for a straightforward game there are only a few that make sense. In most cases an object is coded as an integer that indexes a table of attributes.

Global State

This is just what it sounds like. State variables are global, shared by all of the modules. A lot of programming language design and software engineering has gone into trying to show why this is poor idea. After all, imagine every module having access, complete access mind you, to every other module's variables. Chaos!

On the other hand, there is a certain convenience to this scheme. If the graphics system wants to know where a tree is, it simply gets it from where it is stored. The problem is that it can change it, of course. If you are writing a small system with pretty clear modules, and you are relatively disciplined, then this will work out. The more complex the game is, the more likely this scheme is to result in problems.

Push/Pull (Client Server)

Here, subsystems have incomplete knowledge of one another, and can request information from each other in a structured fashion (a *pull*) or send a new value to a module (a *push*). This is what we will use in our sample game, and what we often see in *Java* and *C++* as accessors and modifiers. For example, if we want to find the location of a ball, we ask for it using a function:

```
getPosition (BALL, &x, &y);
```

This is a pull. If we wish to notify the AI system that an object has been destroyed, we do a push:

```
setExist (object[i], FALSE);
```

This push sets the *exist* attribute of the object to false.

This scheme is elegant, but has another big advantage: it can be used across great distances with equal simplicity. For online multiple player games, the push-pull scheme operates on a server at a remote site, and one of various remote invocation schemes can be used, transparently.

Managers

This could be described as a push-pull model with an intermediate system for handling the requests. For example, the AI system does not own position and orientation attributes in this scheme; they are owned by a management subsystem that has the simple task of hiding the variables and structures and permitting access to them using standard accessor and modifier functions.

Using this scheme, the AI system would have to ask for the position of an object just like the graphics system would, and would also have to request a modification to position from the manager:

```
manager.getPos (OBJECT1, &x, &y);
manager.setPos (OBJECT1, &x+1, &y+1);
```

This is not much more complex than the client-server approach, and has a similar feel. There are few tools that support this model, and so discipline is needed to maintain it. There are few situations this scheme would avoid that the client-server scheme would not also avoid, and so that's not a distinction between the two.

Broadcast-listener

For a certain amount of overhead, we can change the client-server model into one in which modifications to state attributes can be sent to other subsystems by issuing an *event*. For example, when the position of an opponent changes, an opponent-change event can be sent to the graphics system, so that it may be drawn correctly. Given a system similar to the *Java* interface scheme that uses *listeners*, all subsystems interested in this change can be alerted at the same time. Objects or subsystems interested in a particular event, like position change of police cars for instance, would register with the listener so that they would receive the events.

So, using a *Java*-like syntax, we could have:

```
public class Z extends q implements BallListener
{
    ...
    t1.addBallListener(this);
    ...

    public void ballMotion (GameEvent e)
    {
        if (e.getSource() == t1) ...
    }
}
```

This shows the three essential parts of the set-up: declaring the use of the `BallListener` interface in the class header, adding this class instance to the list of those interested in receiving police motion events, and writing a handler (a callback, really) named `ballMotion` that will be called when a police motion event takes place. There is no direct communication between subsystems in this scheme. Information is sent to those interested, and only those, and is queued in the case where there are multiple events occurring simultaneously.

Now, this is pretty clever, and if you can make it work properly in a language not offering specific support, it is sure to give you a programmer rush. On almost all PC systems a process generally uses *only one CPU*. We can pretend that processes are independent if we like, but switching

between software processes takes time on our single CPU, and treating software events like variable modifications as if they were asynchronous processes is a little wasteful and obscures the flow of the code. This system works efficiently on a multiple processor console like the *PlayStation*. Still, some systems profit from using threads and such, and this is the way to deal with state on such systems.

Shared and Global Entities

This uses the inheritance characteristic of the language, object oriented of course, in which the game is implemented. Think of it as global state, but with references to classes and inheritance. Both the AI and the graphics system would have a reference (pointer) to a police car object, and could, by manipulating the accessor and modifier methods, get and change the object's position, orientation, and other attributes. This is the classic object-oriented way—cleaner than using globals.

Within this scheme there are many options: a single rooted hierarchy, ownership, multiple inheritance, and so on. This kind of thing has, in fact, become the standard practice in many colleges and Universities that teach programming, and as a result this method, or family of methods, has become the most common scheme for manipulating game objects and system state.

An important complaint with this set of schemes is that they tend to become dependent on a particular language, usually C++, and then decisions become *religious* (i.e. independent of designer intent). This is because of many an argument with software engineers over things like the proper use of multiple inheritance, for example. Since C++ is one of the few languages that permits multiple inheritance, a scheme that depends upon it has limited options for implementation. The problem is that a single rooted class hierarchy does not scale well, and the inheritance structure starts looking like nonsense after a while. Therefore, the most complex system (graphics, in general) tends to be able to specify the structure of the rest. Multiple inheritance scales better, but is hard to change later, and becomes hard to manage when the system gets complex enough. There are also performance issues in all of these schemes.

There is no best scheme, but one based on a client-server scheme or on managers can work well using object-oriented languages like C++ and Java. And even in C where object orientation can be hand coded. Each module contains a set of variables and data structures that cannot be accessed from

the other modules except through the accessor functions provided by the module. However, when needed for testing or while merging modules, there can be globals and shared entities—log file, for example, for dumping test information while debugging. There is nothing to prevent the user of this scheme in using *Java* or a scripting language like *Lua* as a tool for creating small specific purpose sub-modules, as the controller for an opponent, for instance.

However we do it, the management and control of the system state in a complex system like a computer game must be done carefully and with discipline. While the best way has yet to be determined with certainty, it is absolutely clear that modularity, planning, and discipline must be used to achieve success. Sitting down in front of the computer and starting to enter code at the beginning of the development process is sure to fail, later if not sooner.

Making a Game Function on a Computer

The discussion up to this point has been pretty abstract. Let's start building a very simple game in order to see how what we've learned so far can be put into practice.

An Example - *Hockey Pong*

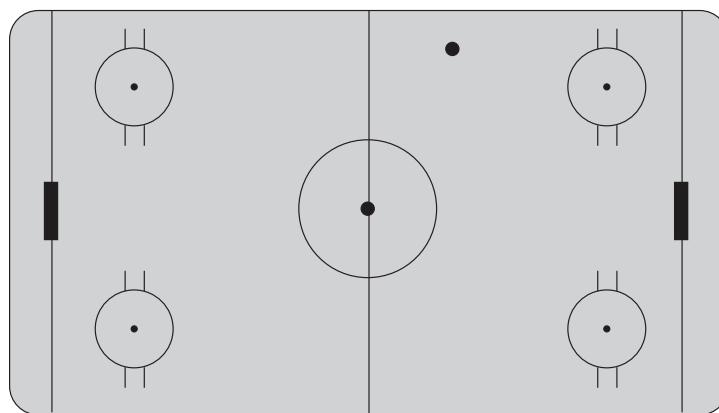


FIGURE 1.4 The playing area for the *Hockey Pong* game.

As a simple example of a game AI system, consider a basic two-dimensional game that looks a lot like *Pong*. Yes, this is the same game that appeared in arcades and bars in the 1970s, and was one of the first video

games that could be played in your home. It is a computer version (simulation) of ping-pong or table tennis. The game we are building will be played on a surface like the one shown in Figure 1.4, which looks like a hockey rink. Real table tennis is very much a three dimensional game, whereas the video version and the game to be proposed is more 2D. Therefore *Hockey Pong* is suggested, which is the same as regular Pong but takes place on the rink above. There are two paddles on opposite sides of the surface, which can be moved up and down under the control of players who use the keyboard to control them. A puck, represented by a small circle, moves left to right and up and down across this surface.

The idea, in case you never played the game, is to prevent the puck from going past your goal line by blocking it with your paddle. The puck bounces off the sides of the rink and off the paddle, but will leave the playing area on each end. When this happens, the other player scores a point.

The basic AI for this game is simple but possibly educational. Here is one implementation based on the idea that at each step in the game the puck occupies a location on the game surface. If that location is also occupied by a paddle, the puck will bounce. If the location is on the boundary (WALL) the puck will also bounce, or will score a point if it is one of the walls at the ends of the rink.

```
AI-step:  
Move_user: s = compute_next_puck_position();  
          piece = contents_of (s);  
  
          if (piece == WALL) puck_bounce_vertical();  
          else if (piece == PADDLE)  
              puck_bounce_paddle();  
          else if (piece == END_WALL)  
          {  
              score_point();  
              reset_play();  
          }
```

One earlier point should now be clear. The use of if-then-else to make decisions is all that is needed, and no fancy decision making algorithms have been used. Instead we make a lot of small decisions quickly. Any automatic opponent is missing at this point, but we will return to this game later on.

The design and implementation of even a simple game is a complex process, and there are some ways to make the process more transparent

and less error prone. One way is to have a very clear idea at the outset of what the game will do in all circumstances, and how it will be done—in other words, design documents. These are not just for software engineering or working in teams, but form a concise explanation of the operation and structure of the game, a blueprint if you like. No sensible builder would start construction on anything, not even a small shed, without some drawing, and no technician would think about building a new electronic circuit without a diagram. The description above is not sufficient for implementation, but it is the basic idea.

What documents should we have? Well, game developers each have their own detailed sequence of documents that they use, and these are sometimes specified by their publishers. We want the minimum needed for our task, which is to build a small game in our basement using only a few people—perhaps only one. We need, as a minimum, the following two documents:

High Concept Design

A professional HCD can be 25 to 30 pages, and is used to sell the game to a publisher, among other things. Ours will be 1 to 2 pages, and will be used mainly to crystallize our thoughts on how the game will be played and why you believe it to be fun.

Game Design Document

This document starts with the basic game objects and goals, and explains what objects are needed and how each of the game activities is implemented—from architectures down to data structures. This will be the major document that we'll use to build the game. Its existence prior to writing code is essential for success, since it provides the blueprint for the actual code. A professional design could be hundreds of pages, but for our purposes we can stick with one that is 10 to 25 pages.

A complete, if simple, High Concept Design for *Hockey Pong* appears at the end of this chapter. The game itself will be implemented in Chapter 4.

The Game Loop

We now know enough about the inner workings of a game to sketch the basic code. At the center, a game is a loop that checks for user input, moves each object that needs to be moved, schedules needed sounds and draws the next frame. This sounds simple, but the phrase *moves each object* could

require thousands of lines of code. This loop, called the *game main loop*, looks like this in pseudo-code:

```
do
{
    Check user input
    Move objects
    Draw the frame
    Play sounds
} while (game_continues)
```

There are many things wrong with this simple loop, but it does describe the operation of the game at one level of abstraction. In fact, this is pretty much what *any* game does, at its heart.

The above organization of the *game loop* is of the type referred to as monolithic, and this is appropriate only for a variety of games provided the control of time is appropriate, as we shall see. The game loop usually handles most of the game states, such as playing, paused, front-end, and so on. It must organize the correct order for the essential modules, like movement and collision tests. The monolithic scheme above is a clear way to organize, but discipline must be maintained during its design or it ends up having to know too much about lower software levels and variables.

Of course, no one scheme works for everything. For example, the audio subsystem is often a distinct thread. Once a sound is set playing, the game should not wait for it to finish. It is also common for file operations to be sent to a parallel thread, since input and output do not require the attention of the CPU. Waiting for data from a file would be a waste, while waiting in a thread is not.

A game frequently has a great deal of set-up before it can be played, and a degree of take-down afterwards so that the computer can be used for something else. In addition to the game loop, we may have:

```
Display startup screen
Read initial audio files
WHEN user types ENTER or startup screen is complete
Display initialization screen
Read initial graphics and other data files.

resume: loop
Allow user to set parameters
until user selects START_GAME
```

```

start:   execute main-loop
        confirm user exit, else goto resume
        save game state is needed
        free game resources: sound and graphics
        exit.

```

The code up to the label `start` is sometimes called the *front-end*. It serves the dual purposes of getting initial information from the user/player and loading the essential data from files. The program can read quite a lot of data from a file while the user finds a box on the screen with the mouse and clicks the button. So, it is true, there is some stalling going on. We want the user to appear to be doing something rather than just waiting for the game to start.

After the game is over it is normal practice to free up memory that the game allocated, close any open files, and so on. Some games allow you to save the state and resume from that point next time, and others keep a file of players and their high scores. All of this is done in the *back-end*, after the main loop is complete. The main loop is never complete until the user says so, or unless an error occurs.

The Control of Time

Any simulation has a means for controlling time. Time in a game or simulation does not move at the same rate as time in the real world, and so there must be a way to keep track of the simulated time and the corresponding real time. Things must appear to happen at a reasonable rate, a rate that does not depend on how fast your computer is.

Non-game simulations normally do not execute in real time, and so do not care what time it really is. Simulated events must synchronize in simulated time. A common simulation scheme is to claim that the current time is the time of the current event, and the next time is the time of the next event. This is clearly not suitable for a game because for one thing the display should not appear to freeze just because not much is going on. We could create *next frame* events that occur every 1/24 of a second. This is called the *next event* method in simulation jargon, and the *variable time step* method in games:

```

do
{
    currentTime = time of next event that occurs;
    deltaTime = currentTime - time;

```

```

        time = currentTime;
        evaluate (time);
} while (GAME_NOT_OVER);

```

However, some kinds of dynamic simulation break time into fixed duration increments, and examine the world every `t` seconds to see what things have to be updated. So, code for this could be:

```

do
{
    time = time + deltaTime;
    evaluate (time);
} while (GAME_NOT_OVER);

```

This has the huge advantage of simplicity, and if the time step is a nice multiple of a good frame rate, then the graphics will look good too. It also permits the programmer to debug the code, because the program should behave in a repeatable fashion. Debugging a simulation or game is a difficult problem, because there are necessarily random behaviors taking place, and random times between events. The technique above, the *fixed time step*, allows the simulation to be repeatable, provided we use a standard set of random number seeds.

There will be a problem if the `evaluate` function, whose job is to see what's going on now and implementing it, takes longer than `deltaTime` to execute. The longer the game runs, and the more objects there are, the more likely this is to occur. There is also a trade-off between accuracy, which is assisted by many small time steps, and efficiency, which would dictate a few large ones.

The two methods above can be combined into one very useful scheme sometimes called *multiple fixed steps*. If the time `current_time` is the time of the next significant event, perhaps a collision, then the time between now and then can be broken into fixed time intervals.

```

do
{
    current_time = time of next event;
    while (time < current_time)
    {
        time = time + deltaTime;
        evaluate (time);
    }
} while (GAME_NOT_OVER);

```

This is still fairly simple, but there is one obvious problem: if a lot of events occur in succession, we get a lot of events to process at once. This can happen in any of the schemes above, and it means that we have to choose which ones to deal with and which ones to ignore. We could change the scheme so that every time interval (i.e. between events) has at least one or two subintervals, or we could place a minimum value on the `deltaTime` value. In general, this seems to be a good compromise between complexity and efficiency, and appears to permit good graphics display rates.

Processing has a built-in time control scheme. All one-time initializations are placed in a function called `setup()`, and this is executed just once, when the program starts. Then, after every user defined fixed time interval, the function `draw()` is called to render something in a graphics window on the screen. The interval between calls to `draw` can be specified using a function called `frameRate(r)` which is passed the number of calls per second—30 would be typical. Experiments with *Processing* show that `draw()` will execute until it is complete, however long it takes. This means that the specified frame rate is a minimum, and some care should be taken to ensure that the frame rate remains somewhat stable, perhaps using a variation on the multiple fixed steps method.

Control of Objects

Any modern game contains a vast number of objects, also known as entities. *Hockey Pong*, our sample game, contains two basic types of objects, and only three on the screen at once, whereas some games have hundreds of objects and scores of types. The puck can move, and therefore can collide with other objects. We need a way to keep track of all of our objects.

It makes sense to have one place for moving objects and another for still ones (obstacles). Still objects cannot interact with each other, only with the moving ones. We could keep a table of objects, indexed by an integer code that is unique for each. The table will store properties of each object: position, velocity, perhaps orientation, perhaps a graphical rendition (sprite). If the object is one that can move, perhaps we need to store information about its current goals and behaviors. In shooters, we need health values and ammunition levels; we may also have skill levels in various categories and a collection of objects. The system we are building here—an *object management system or OMS*—may contain a lot of information, but the organization is nothing that a programmer with a few years of experience has not seen before.

We also need to be able to retrieve and store attributes of objects. There must be an interface to the table that permits us to inquire efficiently as to the position of the enemy vehicle. We will be implementing their motion, so we need to be able to update positions too. In an object-oriented language like *Java*, the object's attributes would be an integral part of the implementation for the object itself, and the table would simply contain references to objects organized in a convenient way. In a language like *C* or *Pascal*, the table would be spread out over multiple arrays, each containing a structure holding relevant data.

Nothing could be more relevant at this point than to give some pseudo-code as an example that shows one way to handle entity management. The *Hockey Pong* game is pretty simple, having only three objects. It could be coded without a general purpose object manager, but there are educational reasons to do it right.

The first thing to do is classify the objects into moving ones and stationary ones. The puck is a special object; there is only one, and the computer manages its motion. Objects can be coded as integers—which are indices into an array of objects—and the puck could be object zero. The two paddles will be objects 1 and 2. The walls need not be stored in the table, but can instead be identified as 2D board coordinates. These are the fundamental properties of the game objects.

Each object has a position on the screen—that is, in the graphics window:

```
int posx, posy; /* Position on the board */
```

Each object also has a direction of motion that can be expressed as a change in position during each time step:

```
int dx, dy; /* Direction: Where are we headed */
```

Objects can be created and destroyed. This is not true in *Hockey Pong*, but is certainly true in other games:

```
unsigned char exists; /* Still there? */
```

There are more components to a general object, but for now, this is all we need. This discussion has shown only code for a single object. There are many ways to generalize this depending on the language chosen. The most basic is to use arrays of these properties indexed by the object number:

```
int posx[], posy[]; /* Position on the board */
int dx[], dy[]; /* Direction: Where are we headed */
unsigned char exists[]; /* Still there? */
```

They could also be encapsulated in structures (records) or classes, as we'll in Chapter 8 *Processing* does support classes.

Hockey Pong has three objects

```
final int NOBJECTS = 3
final int PUCK = 0
final int PADDLE_1 = 1
final int PADDLE_2 = 2
```

Using this scheme, the X position of the puck at any particular time is `posx[PUCK]` and the Y position is `posy[PUCK]`.

A general object management system will use *accessor* and *modifier* functions to maintain the properties. An *accessor* function simply returns the value of a parameter. It is possible to simply use the parameter variable directly in many cases, but an accessor function gives more generality and flexibility. It can check boundary values, return standard codes for errors, and allow for ease of what software engineers now call *refactoring*—a technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. For example, if the array representation here were to be recoded using classes, it might be said to have been refactored.

A *modifier* function allows the programmer to change the value of a property. It is important to use such functions so that values assigned to properties can be checked to ensure they have sensible values, and so that all changes to properties are performed in a relatively small region of code. A lot of bugs are introduced into programs when variables are modified in many and diverse parts of the program.

Given the needs of *Hockey Pong*, the accessor functions are:

```
int getObjectX (int i); // Returns the X position of object i
int getObjectY (int i); // Returns the Y position of object i
int getObjectDx (int i); // Returns speed in the X direction
int getObjectDy (int i); // Returns speed in the Y direction
boolean getExists (int i); // Does object i exist yet/anymore?
```

The modifier functions are:

```
void setObjectPos (int i, int x, int y); // Set position
void setObjectDirection (int i, int dx, int dy); // Set speed
boolean setExists (int i, boolean t); // Set exist flag
```

These functions form the kernel of the game object management system. Using them, the position and speed of any object can be found, objects can be drawn at the right places, and collisions can be determined. At this point, the details of how the functions are implemented are not important. They can be used as building blocks with which to construct a game, and can be implemented in any number of ways in many languages.

Since we are looking at the AI subsystem right now, we may as well finish sketching it. The initialization steps are carried out by three simple functions:

```
void game_over ()
void initializeAI( void )
void terminateAI ( void )
```

Mostly these just allocate and free space needed by the AI. The `game_over` function can be called from anywhere and it calls all subsystem terminate functions and ends the program.

Front-End/Back-End

We expect there to be a front-end and back-end to a game, but frequently too little or too much effort is put into it. It is a common beginner's error to spend a whole lot of time on a really classy front-end animation that does not reflect the quality of the rest of the game. As we will see later, the tricky bit is to switch the graphics system between the rendering needs of the various subsystems. However, the basic function of the front-end remains to simply introduce the game, and sometimes to stall for time while files are being loaded.

The back-end, at least in our case, is used to display the player's score and time. The hardest part here was getting the message to be displayed. Graphics systems usually don't deal with text very well.

As anyone with a basic understanding of *Processing* knows, the function `draw` is called many times each second to update the screen. What this means is that the `draw` function must draw the correct screen each time, and this can be accomplished by using a state variable that tells us what screen is to be displayed. Initially the state is 0, and that means to display the front-end screen. It also means to process events (mouse clicks and key presses) as the front-end would. When the player somehow indicates that the game is to start, the state will become 1, and `draw` will draw the game-play screen. Finally, when the game ends the state will become 2, and the

back-end screen will be displayed. The basic structure of the draw function that represents this would be:

```
int state = 0;           // main state variable
...
void draw ()
{
    switch (state)
    {
        case 0: frontEnd();
        break;
        case 1: level1();
        break;
        case 2: backEnd();
        break;
    }
}
```

The structure of the mouse and keyboard functions will be similar, and will be dealt with for *Hockey Pong* in detail in Chapter 4. To start making the basic game, all that remains is for us to know now is how to do the graphics, which will be covered in Chapter 2.

A Sample Game Design Document

Hockey Pong: High Concept

(working title)

High Concept

A two person 2D hockey themed game in the style of the original **Pong**. Can you score more points than your opponent in three 2-minute periods?

Features

- Like real hockey, a timed game rather than first past the post.
- Select your team and their logo appears on the ice.
- Two players or one player against the computer.
- Sound effects and music.
- Playable on a PC, Mac, Linux, or online through a web page.

- Variable speeds. The puck changes speed to simulate tempo changes in real hockey.
- Replayable. As the player gets better, their score improves. The AI player has adjustable levels.
- Face-off after goals and at the start of each period.

Player Motivation

It's fun. Some people don't play table tennis, but do like hockey, and this style is different from the usual.

Genre

Arcade

Target Customer

Younger players. Causal players who are hockey fans. Potential advergame.

Competition

Pong variants, of which over 35 can be found on the Pippin Barr Website.

Unique Selling Points

- Playable in 1D form on a tablet
- Could be converted into a network 2D player game
- Timed play
- Face-offs

Target Hardware

Mac, Linux, PC. Browser on non-IOS systems.

Design Goals

- Simple and well-understood mechanics and obvious goals make the game easy to learn
- Adjustable AI makes the game hard to master
- Hockey theme appeals to certain classes of player
- Sound includes crowd noises to simulate a hockey experience

Story

What story? It's hockey!

Objects/Characters

Puck

The puck stays on the ice, moving from one end to the other. It bounces off the sides, sometimes changing speed as it does so. Also, there is a small portion at the ends of each rink where the puck will bounce off of the boards, and the puck also bounces off of the paddles.

Paddles

A typical *Pong* paddle, the left paddle moves up and down as the W and S keys are pressed. The right paddle is controlled by the arrow keys.

Timer

There are three periods of 2 minutes each, and the period ends exactly on time. This means that a goal can't be scored then, even though the puck still moves.

Face-Off

At the beginning of the period, the puck is dropped at center ice. In real hockey there is a face-off, and one side would win. A face-off happens after each goal. A whistle blows to restart the game at a face-off and the first player to type a key wins. The puck will then move in the direction away from the winner's goal.

Summary

In this chapter, you learned about the factors that make computer games interesting, and how computer games work as software. Computer graphics, sound, and Artificial Intelligence are key aspects of a computer game, but most important is the need to keep track of where game objects are and what they are doing.

Most computer games are about the passage of time, and what happens during time intervals. Time needs to be managed carefully, and can be thought of as a resource.

We looked at the High Concept Design of a simple game—*Hockey Pong*—and the factors that go into starting an implementation.

Exercises

The exercises below will apply your knowledge of the material in this chapter, and sometimes require that you do some more research before you are able to complete them. Solutions to selected exercises are available on the companion disc.



1. Give your own definition of *fun*. Don't think too long about it—try to be instinctive. Can fun be measured, using your definition? Is it possible to predict how much fun something will be, or is it only observable after the fact? How is *engagement* different from *fun*?
2. Explain the conflict/challenge in *Tetris*, *Mortal Kombat*, *Frogger*, and *Defender*. How is it maintained?
3. What is the goal of the game *Mario Kart*? *Pac Man*? *The Sims*?
4. Describe, as concisely as possible, the major mechanics of *Angry Birds*, *Portal*, and *Space Invaders*.
5. Examine ten or more Web-based games and list the controls (buttons, sliders) found on the *open* (splash) screen and on any *options* screen. List this in order of frequency, most common first.
6. Many games are *zero-sum* games, in which there is a win and a loss for each choice and situation. Each gain for one player is offset by a corresponding loss for another. Give three examples of a zero-sum game, and explain why they fall into this category.
7. Sketch a game loop for *Tetris*, stating in simple English what needs to be done at each step for this game in particular.
8. Write a two page High Concept Design for any computer game you have played, but do not name the game. Give this to another person; can they figure out what the game is?

Resources

Yale class on Game Theory: <http://oyc.yale.edu/economics/econ-159>

MIT course on Computer Games and Simulations for Investigation and Education:

<http://ocw.mit.edu/courses/urban-studies-and-planning/11-127j-computer-games-and-simulations-for-investigation-and-education-spring-2009/index.htm>

MIT Game Design course: <http://ocw.mit.edu/courses/comparative-media-studies/cms-608-game-design-fall-2010/>

Raph Koster, Theory of Fun for Game Design: <http://www.theoryoffun.com/theoryoffun.pdf>

Free download of GameMaker: <http://www.yoyogames.com/studio>

Bibliography

Barr, Pippin. *Pongs*. n.d. <http://www.pippinbarr.com/games/pongs/Pongs.html> (accessed June 27, 2013).

Dalmau, Daniel Sanchez-Crespo. *Core Techniques and Algorithms in Game Programming*. Indianapolis, IN: New Riders Publishing, 2004.

Fitzhenry & Whiteside. *Funk & Wagnalls Standard College Dictionary*. Markham, Ontario: Fitzhenry & Whiteside Ltd., 1978.

Graham, Mike McShaffry and David. *Game Coding Complete*. Boston, MA: Course Technology, 2013.

Järvinen, A. *Games without Frontiers: Theories and Methods for Game Studies and Design*. Tampere, Finland: Tampere University Press, 2008.

Morgenstern, John von Neumann and Oskar. *Theory of Games and Economic Behavior*. Second edition. Princeton, NJ: Princeton University Press, 1947.

Provancher, William. Multidirectional Controller with Shear Feedback. US Patent 13/269,948. October 10, 2011.

Schelling, Thomas. *The Strategy of Conflict*. Revised edition. Cambridge, MA: Harvard University Press, 1980.

Sicart, Miguel. "Defining Game Mechanics." *Game Studies* 8, no. 2 (December 2008).

Yap, Brian. *Analytical Perspectives in Game Design: Architecture*. 1999. <http://numbat.sourceforge.net/numbbatV2/architecture.html> (accessed 2013).

Zimmerman, K. Salen and E. *Rules of Play - Game Design Fundamentals*. Cambridge, MA: MIT Press, 2004.

CHAPTER 2

GRAPHICS AND IMAGES IN PROCESSING

In This Chapter

- A review of basic *Processing*
- A comparison of image types
- How to draw shapes and images

The *Processing* language was designed for use by artists. Visual feedback is a key aspect of the language, and the graphics window does this by showing the most recent version of a graphic as it is being constructed. Games do this too—a game is displayed as consecutive frames on the computer screen. It would appear that *Processing* is ideal for making games, at least in that sense.

Processing is a programming language, and the act of drawing in a window will require us to invoke code that draws things for us, code that has been written and included with the language. It serves the same function as the graphics part of a game engine, and in *Processing* it is mostly easy to use. There are two basic parts: library functions and types that operate on raster (pixel-based) images, and library functions that create graphical objects from simple components and renders them. The former is done by the provided *PImage* data type and related functions, while the latter is largely encapsulated into the *PGraphics* object.

For raster images, fundamental operations involve reading image files in common formats, displaying images on the screen, and accessing pixels in an image. However, the language was intended for use by artists, so ease of use was given priority over efficiency. Artists tend to manipulate images rather than pixels, so large-scale image manipulations are easy, pixel access is not.

Graphics are done at a simple level, allowing basic shapes like rectangles and ellipses to be drawn with ease, while more complex and less common operations are more difficult. This makes sense, and should be kept in mind when looking at the details. Also, since all graphics on a computer ultimately are displayed on a screen, all graphics will be converted into raster form for display. In a practical sense, this means that the *PGraphics* operations will draw into a *PImage* object before it can be viewed. So, let's look at basic operations and the *PImage* type first.

Review of Basic Processing

A *Processing* program has a `setup` part and a `draw` part. `setup` is a function used to initialize things once, at the beginning of the program execution. `Draw` is called multiple times per second to update the graphics in the display window. One of the important things that `setup` does is to initialize the display window, mainly specifying its size, as follows, where `width` and `height` are the size of the window in pixels:

```
size (width, height);
```

Now everything that the program draws will be drawn into that display window. *Processing* is an environment in which all programs result in a graphic being created, and it makes that process quite simple.

The `draw` function updates what can be seen in the display window each time it is called. The size of the window is now fixed at `width` by `height` pixels, and so drawings must fit into that range or they will be partly or completely unseen.

The `draw` function often erases the previous display window. If this does not occur, then the new drawing will be added to the existing one, rather than replacing it. Erasing the display window can be done by calling one of the versions of the function `background`:

```
background(0,255,0); // Sets the whole window to green
background (128); // Sets the background to a middle grey
background (image); // Sets the background to an image
```

As we will see later in this chapter, there are other ways to modify the pixels in the display window, but this is the easiest one to use. In this chapter, we will expand our understanding of graphics and images in *Processing*, and develop the *Hockey Pong* game more fully.

The `PImage` Type

`PImage` is in fact a `class` in *Processing*. *Processing* is based on the *Java* language, which allows a programmer to encapsulate data and functions that operate on that data as `class` objects, a common programming language feature. This means that a lot of the detail of how a `PImage` is implemented is hidden, and you only have access to the parts that you need to use. It also means that we can declare variables to be of type `PImage`, and can create and destroy instances.

An image is a very useful thing within a game. An image can be used as a background to the play, showing the fixed parts of the scene that place it in space. Think of it as the area on which the game is played. Images are also used as icons or sprites, which are graphical representations of objects within the game. Spaceships, balls, trees, cars—all have a simple, small graphical representation that can be moved around the playing area. An image can also represent the player (avatar) and other active non-player characters (NPCs) that make the game interesting. These are often animated as a sequence of small raster images that are displayed consecutively, like a flipbook. An avatar that appears to be walking is really a sequence of images, each showing a different stage of a gait and displayed in order at the right speed to make the illusion convincing.

We can start with some essential image operations, and move on to more specific details as needed. The simplest thing to do with an image is to display it in the window, and in order to do that we need to read the image from a file. This could take a lot of code in *C* or *C++*, but is nearly a **hello, world** program in *Processing*. Dissecting this code will teach a lot about basic image operations.

First, the image must be read from a file into a variable. Assume that the image is in the file named `image.gif`, although it could be any GIF, PNG, JPEG, or TIFF file. The image will be read into a `PImage` variable named, in this case, `image1`. The program also needs to create the display window and display the image. All of these things are done only once, and

so can be performed within the `setup()` function. Indeed, we don't need a `draw()` function at all here.

Here is the complete code needed to read and display the image:

```

PImage image1;          1
void setup()           2
{
    Image1 = loadImage ("image.jpg"); 3
    size (500, 500);      4
    image (image1, 0,0);   5
}

```

In line 1, there is the declaration of the `image1` variable as an instance of the `PImage` class. In this case it could have been declared within `setup ()`, but not always. Line 4 is where the image is read in using the *Processing* function `loadImage`. This function takes one argument—the name of the image file—parses the file format, and reads the pixels into memory. It returns a `PImage` value that can be used for display or other purposes. In this case, we assign the reference (pointer) to it to the `image1` variable. Line 5 creates a 500x500 display window.

Line 6 calls a function named `image`. This function can be used in a couple of different ways, but in this situation it is passed an image to be displayed in the display window and the X,Y coordinates at which to place it. Displaying the image at (0,0) as in this instance means to align the image to the upper left corner of the window. Running this program yields a graphics window that looks like the one in Figure 2.1.

A second form of the `image` function takes five parameters:

```
image (image2, x, y, width, height);
```

In this case, the image `image2` is drawn at display window position (`x,y`) and will be scaled to the specified `width` and `height`, in pixels. The variables `x`, `y`, `width`, and `height` are all floating point (real) variables, not integers. The ease with which images can be scaled is deceiving, because changing the size of an image is a complex task, involving interpolating pixel values in two dimensions. It is important to experiment with this function to see whether it is good enough for a specific application.

Still, it does present some interesting possibilities. For example, it is a simple matter to display multiple images of any size in a fixed size window. Displaying the same JPEG image as a 100x100 thumbnail in a 4x3 grid in



FIGURE 2.1 An image displayed in the graphics window.

the same display window is straightforward—place the images at x (horizontal) positions 50, 160, 270 and 380 and at y (vertical) positions 50, 160, and 270:

```
PImage image1;  
void setup()  
{  
    int i, j;  
    float x, y;  
  
    image1 = loadImage ("image.jpg");  
    size (500, 500);  
  
    y = 50;  
    for (j=0; j<3; j++) // display an image row  
    {  
        x = 50;  
        for (i=0; i<4; i++) // Display 4 images in the row  
        {  
            image (image1, x, y, 100, 100);  
            x = x + 110;  
        }  
    }  
}
```

```
    y = y + 110;  
}  
}
```

Here, the outer `for` loop computes the y coordinate for a row. The inner `for` loop computes the x coordinate for each of four images in a row and displays them. The resulting display window is shown in Figure 2.2. This is a very small program to accomplish this task, but does have some flaws. For one, the images are not well centered in the display; for another, the images are scaled to 100x100 pixels, which alters the aspect ratio. The thumbnails may have been stretched in the X or Y direction. This is fine for this image in this application, but the stretching would be obvious in some cases and, of course, measurements made on the scaled image might be wrong.

Being able to place an image at any point in the window is essential to creating a game. Being able to move images around as the game is played is also important. Being able to scale images is nice, but the images should be created with the correct size in the first place, as this improves their appearance.

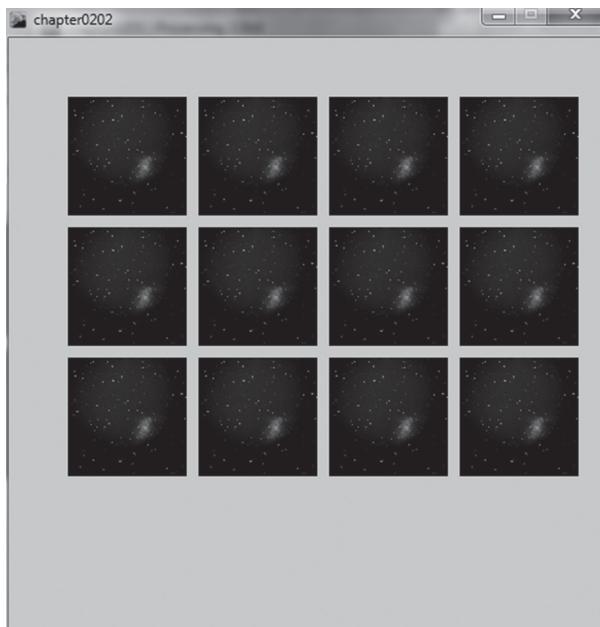


FIGURE 2.2 Displaying multiple images in a window.

Image Files

It's so easy to load and display images using *Processing* that one could forget how difficult it is to read image files. Image file formats contain a lot of information, and the various common formats have vastly different ways to do that. Let's take a quick look at the most common formats understood by *Processing* and see what they contain and what they are useful for.

Graphics Interchange Format (GIF)

This is one of the earliest file formats for images that is still in use. It is simple to include a transparent color, and it can hold a sequence of images that create an animated scene. These GIF animations are very common on the Internet to this day; such files end in **.gif**.

Each pixel in a GIF is 8 bits in size, and is the index into a table of colors. This means that any particular GIF can only contain up to 256 colors in all. The pixel data is compressed using the lossless LZW compression scheme, so this also helps keep the file size small.

A GIF file has the characters ‘GIF’ as the first three in the file, and this is followed by a header that contains useful information such as the number of rows and columns in the image.

Joint Photographic Experts Group (JPEG)

JPEG files end in **.jpg**, or **.jpeg**, or even **.jpe**, depending on what software created them. This format was designed with digital photography in mind, and it uses a lossy compression scheme that claims up to 10:1 compression with very little loss of quality. Any loss is unacceptable for scientific applications, and so this format is not suitable for scientific work. A JPEG image can be very large—up to $65,535 \times 65,535$ pixels—and can contain full 24-bit color values.

JPEG files can be quite small for a specific number of pixels. They do not allow for transparent color or no animation.

Tagged Image File Format (TIFF)

The TIFF format is popular in publishing and especially in astronomy. A key aspect of this format is a set of text *tags* that can be used to describe the data in the file, not unlike the tags in HTML files. It allows the user to specify compression methods (including no compression), type of color data, even byte order. Images can be composed of tiles or strips, and can

consist of multiple sub-files. Scientific users like the fact that metadata can be easily included. Photographers can add information about the camera and lens type, exposure, date, and time.

TIFF is likely the most flexible format that exists, but this flexibility also makes it more complex, and therefore harder to read and parse. Its complexity mitigates against its common use, and it is rare to see TIFFs in games.

Portable Network Graphics (PNG)

The PNG format was devised specifically to eliminate the need for GIF images after a patent on the GIF compression method was enforced. Like GIF, it uses a lossless compression scheme, and can use a color palette. Unlike GIF, it has full 32-bit RGBA color as an option. It supports a variety of transparency options, including a full alpha channel (a byte as in *Processing* colors) and a specific transparent color (as in GIF). PNG files end in **.png**. They do not allow animation, although there are extensions to the format that allow frame sequences.

Comparisons

GIFs are still useful when an animation is needed, and are good for sprites. They are not good for photos. On some images, GIF can achieve the greatest compression, and so the smallest file sizes.

BEST USES: sprites, especially animated ones; small animations (TV screens, windows).

JPEGs are excellent for photographs, but are not useful scientifically. They have no transparent color so are not good for sprites. They show artifacts along edges and blur color gradients. JPEGs are recompressed each time they are edited, and so compression errors multiply.

BEST USES: background images.

TIFFs are valuable for publishing and science, but are hard to parse and tend to be large in size. Most application software can read only a subset of possible TIFF file types.

BEST USES: documents, scientific data.

PNGs are generally good replacements for GIFs but, while usually smaller in size, still do not allow animations.

BEST USES: web images, some sprites, backgrounds.

Format	Lossy Compression	Color	Transparency	Size	Animation
GIF	No	256 colors	Yes—1 color	Small	Yes
JPEG	Yes	24 bit color	No	Smallest	No
TIFF	Yes or No	Normally 24 bits	No	Medium	No
PNG	No	Up to 32 bits	Yes—alpha or 1 color	Small	No

Pixels and Color

Colors are important in graphics because lines, shapes, and regions will be drawn in different colors. On a computer screen and in memory, graphics are drawn as individual pixels, even if we don't specify individual pixels directly. We know that a pixel is a color or intensity value that is measured or drawn at a particular location in an image. The key to representing a pixel is to represent the color, and color on a computer is usually specified by its red, green, and blue components (RGB). The choice of how to implement a pixel would seem to be made for us—it will be a collection of three values, one holding the amount of red at that location, one holding the amount of green, and one holding the amount of blue.

The built-in type `color` represents colors as described thus far, and so a pixel would be of that type. A variable of type `color` has a red, green, and blue component, each of which is an 8 bit unsigned integer. An 8-bit (one byte) integer implies that each component can have a value between 0 and 255. If the component were red, a value of 0 would mean that no red is present in the color, and a value of 255 would mean that the most red possible is present—this is also known as saturated.

A typical PC has either 32-bit words (4 bytes) or 64-bit words (8 bytes), but in either case a 32 bit unit can be accessed. If each 32-bit word contained either a red, green, or blue component then 75% of the space would be wasted, which is to say only 8 bits of the 32 would be a color value. To avoid this, the red, green, and blue (r,g,b) values are packed into a single 32 bit word, thus using 24 of the 32 bits. The `color` type in *Processing* is really just a 32-bit integer.

Getting and Setting Color Values

The people who created *Processing* provided some useful and simple functions for manipulating colors. To take r,g,b components and create something of type `color`, the function `color()` is given. For example:

```
color c;
int r=13, g=5, b=4;
c = color(r,g,b);
```

This code creates a `color` object and gives it a red, green, and blue value. The variable named `c` above can now be used as a pixel, and can be displayed on the screen or stored in an image. The same code as a declaration and using constants as parameters is:

```
color c = color(13, 5, 4);
```

This declares and initializes the variable in one statement.

It can be confusing that one word, such as `color`, can be used to mean both a type and a function. Computer languages can sort that out by context. If the word `color` starts a statement and is followed by some names, then the type `color` is meant; if the word `color` is followed by a “`(`” and some values, then the function `color` is meant.

Given a variable of type `color`, there are also functions for getting the red, green, and blue components. Setting the colors correctly is important. Some analysis involves examining pixel values and looking at their color. For example, merging texture at their edges might need this. It is also possible to use `color` to identify the location of an object on the screen—if it’s on a grey pixel then it’s on the road, if it is on a green one then it is on the lawn. This means that it will be important to be able to extract any pixel from an image and pull out the `r,g,b` components. In other words, the problem is, given a pixel `c`, what are the `r,g`, and `b` values?

Of course, *Processing* provides functions for doing this. They are convenient, clearly named, and easy to use. If you have an existing variable `c` of type `color`, the `r,g,b` components can be extracted using the following functions:

```
r = red(c);
g = green (c);
b = blue(c);
```

It really is that simple. We’ll see more details on how colors are represented later.

In addition to the red, green, and blue components of a pixel, there is a fourth property, called *alpha*. It refers to the degree to which the pixel is opaque or transparent. A pixel that is completely transparent would be effectively invisible, since anything drawn before (i.e. underneath) it would

show through. Conversely, a pixel that is fully opaque would cover anything beneath it. The use of an alpha value to control the degree of transparency allows objects underneath any pixel to be visible to a greater or lesser degree. This usually appears as a color change in overlap areas.

It is important to remember that an alpha of 0 means that the pixel is completely transparent. An alpha value of 255 means that the pixel is opaque. This is the opposite of what one would expect if alpha is the degree of transparency, since 0 transparency would seem to mean it is not transparent. Figure 2.3 shows the effect of changing the alpha value. The green (rightmost) circle in each overlapping pair has a specified alpha value of 255, 200, 128, and 60 (left to right). Note that the color of the other circle shows through more and more as the alpha of the green circle gets smaller.



FIGURE 2.3 The effect of the alpha value on pixel color.

Drawing Shapes

Looking at the *Hockey Pong* game described in Chapter 1, the graphical needs are simple. The puck is a small black circle, and the paddles would seem to be simple rectangles. *Processing* can handle that and more using a set of built-in functions that draw basic shapes. For example, a rectangle is drawn using the call:

```
rect (x, y, width, height);
```

In this call, `x` and `y` represent the position of the upper left corner of the rectangle, and `width` and `height` represent its extent, in pixels, in each direction. A triangle is also a basic shape, and is drawn by specifying the three vertices of the triangle as coordinates:

```
triangle(x1,y1,x2,y2,x3,y3);
```

A line is simple too. A line segment is drawn by specifying the two endpoints:

```
line(x1,y1,x2,y2);
```

A circle is a bit more complicated. *Processing* draws ellipses, and a circle is a special case of an ellipse. An ellipse has one long axis, the major axis, and one short one, the minor axis, as illustrated in Figure 2.4. In the case of a circle, these are the same length, that being the same as the radius. So, drawing a circle of radius r centered at (x, y) would be accomplished by:

```
ellipse (x, y, r, r);
```

The axes must be horizontal and vertical, so not all possible ellipses can be drawn this way. The first parameter following the x and y coordinates of the center of the ellipse represents the length of the horizontal axis, and the final parameter represents the length of the vertical axis. So, as seen in Figure 2.4, the ellipse drawn by the following code is very long and narrow:

```
ellipse (x, y, 30, 6);
```

Whereas reversing the last two numbers will give us a tall and thin ellipse:

```
ellipse (x, y, 6, 30);
```

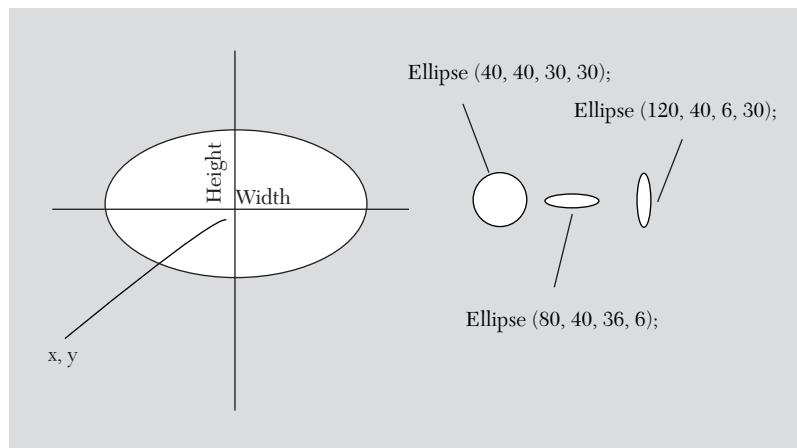


FIGURE 2.4 How an ellipse is specified. A circle has width and height equal. All basic ellipses have their axes aligned with the image.

Lines and Color

Rather than specifying a color each time something is drawn, *Processing*, like many other graphics systems and game engines, uses a default value. In fact, there are two default colors, and the user can set both. The *stroke* color is used for drawing lines and is set as in the following example:

```
stroke (255, 255, 0, 128);
```

This specific example results in all further lines being drawn in the color yellow, which is RGB = (255,255,0) and an alpha of 128 until the next call to `stroke`. The alpha is optional. If only one parameter is passed, then it will be considered a grey level; thus, one of the most common calls to `stroke` sets the line color to black:

```
stroke (0);
```

Looking at Figure 2.4 you can see that the ellipses there have a black outline drawn around them. This is the default. If you do not want an outline drawn then you can call the function `noStroke()`. No outlines will be drawn until the next call to `stroke()`.

The second default color in *Processing* is the `fill` color, which is the color used to fill the inside of circles and polygons. As expected, simply calling the function `fill` with a set of RGB values will set the fill color:

```
fill (0, 128, 128);
```

A fourth parameter could be passed, the alpha value. Again, if only one parameter is passed it is interpreted as a grey level. Finally, a call to `noFill()` will remove the fill color completely. This means that if both `noStroke()` and `noFill()` are called, then everything that is drawn from then on will be invisible.

Hockey Pong

The objects used in *Hockey Pong* are very simple, and could easily be drawn with what we now know about *Processing* graphics. Let's write a function named `renderObjects()` that draws all moving things. It will be called each time `draw()` is called during game play.

```
void renderObjects()
{
```

The puck is just a circle, and will be black:

```
    noStroke();
    fill (0);
    ellipse (posx[PUCK], posy[PUCK], puckRadius, puckRadius);
```

You may recall from Chapter 1 that the current position of objects is kept in a pair of arrays and is accessed using the object index—in this case the constant `PUCK`. The variable `puckRadius` is the size of the puck. Using accessor functions drawing the puck would be done as:

```
ellipse (getObjectX(PUCK), getObjectY(PUCK), puckRadius, puck-
Radius);
```

The paddles are just black rectangles. The fill color is already black, so we can go ahead and draw them using accessors:

```
rect (getObjectX(PADDLE_1), getObjectY(PADDLE_1), paddleWidth, paddleHeight);
rect (getObjectX(PADDLE_2), getObjectY(PADDLE_2), paddleWidth, paddleHeight);
}
```

Text and Fonts

When using a graphics system, text is drawn onto the screen. In a text window such as you would find in Microsoft Word, characters follow one another smoothly as they are typed, are the correct size and color, and are the proper distance from the previous character and from the one above. This takes a bit of work, which is invisible to the user unless it fails. When drawing characters the onus is on the person drawing them to make it happen correctly. A position is specified, a string is given, and a font and size is designated, then the character string is drawn.

As usual, the first steps are simple. Drawing a string is a matter of using a function named `text`:

```
text ("Hi there!", 100, 50);
```

This call will draw the string “Hi there!” at position x=100 and y=50 using the current font and size and color. There are variations on the `text` function call, but this is the basic idea. The default font is a basic sans-serif font, like *Calibri* or *Ariel*. Much of the complexity of dealing with text involves fonts and sizes.

The current font color is the current fill color, and we know how to set that. The font size can be changed using the `textSize()` function, for example `textSize(48)` sets the font size to 48 *pixels* (not 48 point). The use of pixels as a size unit makes it easier to draw them. This function scales the font images, so changing the size by a large amount results in poor quality text.

User-specified fonts in *Processing* are loaded from files. There are likely a great many font files on your PC if you have *Processing* installed, and the ones used by *Processing* have names ending in **.vlw**. These are in many ways like image files, and just as images are loaded into memory using `loadImage`, a font is loaded using the `loadFont` function. And, just as a `PImage` type is an internal representation of an image, the type `PFont` is

the internal form of a font. To load a font, create a variable of type `PFont` and then read the font file using `loadFont`, saving the returned value in the variable, for example:

```
PFont myFont;
.
.
myFont = loadFont ("Calibri-20.vlw");
```

The file **Calibri-20.vlw** would be found in a local directory named **data** in this instance. It represents the **Calibri** font in the 20 pixel size. A full path name can be used instead of just the file name, meaning that the font files can be located anywhere.

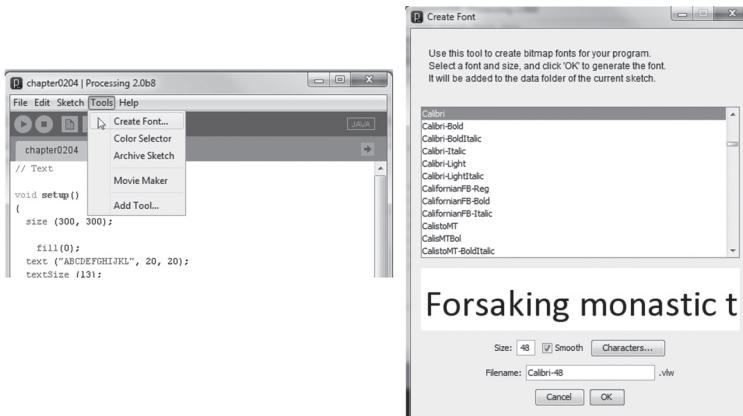


FIGURE 2.5 Using Processing to create a font file.

Processing allows you to create a font file with any available font and to save it in the data directory for use in your game. Simply click on the **Tools** menu item and select the **Create Font** item (Figure 2.5). The **Create Font** menu pops up and you can select the font and size using the mouse. When you finally click on **OK**, the file for the font you selected will be written to the data directory, and can be used immediately.

Internals of the Type `PImage`

The built-in type `PImage` is a container for everything that the program needs to know about an image. The inner structure of this type will remain hidden, as is the norm when using built-ins and pre-defined classes. We've already used a `PImage` variable to hold an image, and have read and displayed an image using built-in functions. We need to know more about how

to access the functions and values offered so that we can fully understand what is possible and how to use images effectively in a game. We're going to use some small sections of code in the explanation, so consider the following declarations to be in effect for this entire section:

```
// Some image variables to use for exposition
PImage thisImage, thatImage;
int i, j; // index variables that give pixel indices
int nRows, nCols; // holders for an image size
```

Image Size

A raster image has its size expressed as the number of rows and columns of pixels that comprise it. Knowing the extents of an image in both dimensions is critical, if for no other reason than to prevent exceeding the image bounds and getting an error. An image has a size with which it was initially created, whether by a camera or a drawing program. That size was saved with the original image in a file, PNG or GIF or JPEG. Whenever the image is modified by a program the size is required, and is saved along with the image data. It must be possible to access those values through a `PImage`.

`PImage` has two variables, `width` and `height`, that can be accessed directly. Using the declarations established above, after an image has been assigned to the variable `thisImage` then the number of pixels in each row is found in `thisImage.width`, and the number of rows is `thisImage.height`. These values can be used to code a loop to examine all pixels, or create a display window of precisely the same size as the image. Here is a `setup()` function that does that:

```
void setup()
{
    thisImage = loadImage ("test.jpg");
    size (thisImage.width, thisImage.height);
    image (thisImage, 0, 0);
}
```

Because `width` and `height` are fields inside of a `PImage`, an unfortunate property is that they can be modified. If set to an erroneous value there can be problems, since the number that the system thinks indicates the image size may not really be the image size. So, if we enter the following code, the image is 21 pixels wide for the purposes of this program:

```
thisImage.width = 21;
```

It won't make a proper sub-image, though: the first 21 pixels it sees will be the first row, the next 21 will be the next row, and so on.

There is a proper way to resize an image that uses a *method* (a function provided by the `PImage` type) named `resize`. It's straightforward—setting the size of `thisImage` to 50 rows by 80 columns would be done as follows:

```
thisImage.resize (50, 70);
```

As mentioned before, this kind of indiscriminate altering of an image size most often changes the aspect ratio, the ratio of height to width. This makes the image look compressed or stretched in one of the other direction. A way to avoid this is to pass a 0 parameter to `resize`. Either the width or the height parameter can be zero, but not both. The following call changes the image so that it has a width of 50 pixels and a height that will keep the aspect ratio the same as it was without distorting the image:

```
thisImage.resize (50, 0);
```

When an image is resized, the new version is an approximation. When shrinking an image, each new pixel becomes an average of a small set of pixels; when expanding an image, pixels are created between existing ones and their values are interpolated from their neighbors. Resizing an image more than once means approximating an approximation, and the defects introduced become visible quite quickly. Thus, if you need to resize an image many times for some reason, always start from the original.

Accessing Pixels

If we are going to be drawing objects and moving them around the graphics screen, then it would seem to be important to be able to look at individual pixels or small regions in an image or change their values. A `PImage` variable has access to an array named `pixels` that holds a copy of all the pixels in the image. The actual pixels are hidden, so we ask for an up-to-date copy and then we can start examining pixels and modifying them. When we're done, if necessary, we can copy the `pixels` array to wherever the actual pixels are stored and the change becomes permanent.

As an example, let's write some code that swaps the red value for the green one in every pixel. First, we need to update the `pixels` array; then we look at every pixel and extract the RGB values. The R value is written to the pixel in place of the G and vice versa, and then the image is saved.

The code is:

```
int r, g, b;
thisImage.loadPixels();      //Update our pixels array
// loop over all pixels
for (int i=0; i<thisImage.width*thisImage.height; i++)
{
    r = (int)red(thisImage.pixels[i]);
    g = (int)green(thisImage.pixels[i]);
    b = (int)blue(thisImage.pixels[i]);
    thisImage.pixels[i] = color (g, r, b);
}
thisImage.updatePixels();
```

The function `loadPixels` is called at the beginning to copy the pixel values into the `pixels` array. Then a loop is written that looks at each pixel, extracting the `r`, `g`, and `b` components using the `red`, `green`, and `blue` functions applied to each pixel value. A new color is created that has the red and green values interchanged, and this is assigned to the `pixels[i]` location, which is the pixel under consideration. After all pixels are modified, the function `updatePixels` copies the pixel values back into the image. Figure 2.6 shows the result of this code applied to a sample image, a photograph of the Victory Column in the Tiergarten Park in Berlin. The red car in the lower right becomes a lovely green after the transformation, and the sky becomes an unlikely mauve.



FIGURE 2.6 (Left) A famous landmark in Berlin. (Right) Color swap red for green in each pixel.

The array of pixels is linear, one dimensional, while an image is a two-dimensional array of pixels. It is common to access a specific pixel by specifying its row and column, and when using *Processing* we can do this using functions `get()` and `set()`. The `get(i, j)` function returns the pixel at row `j` and column `i` of an image. A call to `set(i, j, c)` assigns the color

`c` to the pixel at row `j` and column `i`. As an example, the code just written to swap red and green in each pixel could be re-written using `get` and `set` as:

```
thisImage.loadPixels();      //Update our pixels array
//  loop over all pixels
for (int i=0; i<thisImage.height;  i++)
    for (int j=0; j<thisImage.width;  j++)
    {
        c = thisImage.get(i,j);
        r = (int)red(c);
        g = (int)green(c);
        b = (int)blue(c);
        c = color(g, r, b);
        thisImage.set (i, j, c);
    }
thisImage.updatePixels();
```

The ability to modify pixels is the basis of computer graphics. We now have the ability to create an image containing any set of pixels we choose to draw. There are two problems with that. First, we need to be able to create a new, empty image of any size we like to use as a place to draw, a canvas if you like. Naturally, *Processing* solves that problem by providing a function. The following call creates a new `PImage` having 100 columns and 200 rows and assigns it to the `PImage` variable `thisImage`:

```
thisImage = createImage(100, 200, RGB);
```

The `RGB` variable that serves as the final parameter is the format of the image, and can be one of: `RGB`, `ARGB`, or `ALPHA`. `RGB`. Check the *Processing* documentation for details of the other formats.

The second problem is that all we can do right now is to read or modify one pixel at a time. The functions for drawing things, like `ellipse` and `rect`, do not draw into a `PImage`. It would be good to have some way to draw objects that are more complex into an image, not just onto the graphics window. This can be done, but it involves knowing about another new type: `PGraphics`.

The `PGraphics` Type

This is what many systems people call a *graphics context*. The drawing that *Processing* does to the graphics window, and all other graphics displays is done through `PGraphics`. This means that functions such as `ellipse`,

`rect`, `and` `stroke` belong to a `PGraphics` object that is created for you when the program starts to execute, and that displays the results to the graphics window. However, there can be other instances of `PGraphics` created, as many as you like. You can draw to these and then, if you like, display them to the screen.

To illustrate this, we'll create a `PGraphics` variable, draw a circle into it, and then display it on the graphics window. In `setup` we'll do the initialization:

```
PGraphics myGraphics;
void setup()
{
    size (400, 400);
    myGraphics = createGraphics(200, 200);
}
```

This code declares a `PGraphics` variable named `myGraphics` and then creates an instance by calling the `createGraphics` function, passing the size of the raster region to be drawn in. This is very much like creating a `PImage` using `createImage`. Before we can draw in this area, which has no display device associated with it, we need to initialize it further by calling `beginDraw()` to set up the drawing buffer. Then we can draw the circle using the drawing operations associated with the variable `myGraphics`, and finally call `endDraw()` to indicate that we're done drawing, and resources can be freed. At this point, the `myGraphics` variable contains our drawing. To display it on the graphics window we call the `image` function specifying the location in the window where we want the drawing to go. Coded as a `draw()` function this looks like:

```
void draw()
{
    myGraphics.beginDraw();
    myGraphics.stroke(255);
    myGraphics.ellipse(100, 100, 30, 30);
    myGraphics.endDraw();
    image(myGraphics, 10, 30);
    image(myGraphics, 120, 130);
}
```

This code will result in two circles being displayed in the graphics window, but only one has been rendered; the other is a copy.

Rendering to a `PImage`

The use of `PGraphics` objects like this is pretty common in games. One important use is to render complex scenes into an image rather than on the screen and then use `image` to flip the new scene onto the screen quickly (double buffering). This reduces flickering caused by slow updates to the screen. It is also used to draw semi-permanent things onto a background, which is really a `PGraphics` object, and have them remain there without having to remember where they are. Things like bullet holes, for instance, can be drawn onto a permanent background image.

A `PGraphics` object is treated by some functions as an image, and can be drawn to a `PImage` using a variation of the `get` function. It is true that `get(i, j)` returns the pixel at column `i` and row `j`, but it is also true that `get(i, j, w, h)` returns all of the pixels starting at `(i, j)` and ending at `(w, h)`. It returns these pixels as a `PImage`. Finally it is true that `PGraphics` contains the `get` function. So, the way to copy the graphics from a `PGraphics` to a `PImage` is:

```
myImage = myGraphics.get(0, 0, myGraphics.width, myGraphics.height);
```

Of course, `myImage` can be drawn in the display window using
`background(myimage);`

With this new information, we now have a way to draw things offline and display them all at once in the display window.

Summary

A game requires that objects are drawn on the computer screen, and that the screen can be updated quickly and repeatedly so that motion can be simulated. The `PImage` type represents an image, and can be read from a file and drawn on the screen using elementary *Processing* operations. Accessing individual pixels in an image allows the display to be manipulated at the lowest level, and a `PImage` type has operations that allow this. An object of the type `PGraphics` is an in-memory version of a display, and can have images and objects drawn into it for later display in a window.

Exercises



The exercises below will apply your knowledge of the material in this chapter, and sometimes require that you do some more research before you are able to complete them. Solutions to selected exercises are available on the companion disc.

1. Write a *Processing* sketch that sets the background color of the graphics window depending on the position of the mouse setting all three primary colors. For example, the far left of the window could have no red, and the far right could be full red, the top might have no green and the bottom could have full green.
2. Create a sketch that will display an image that is specified when a user types the file name of that image on their keyboard.
3. Create a sketch that will display a line on the graphics screen that begins when the mouse button is pressed and that ends when the button is released. The line should remain on the screen after the button is released. The sketch should allow for many lines to be drawn in this way, like a paint program.
4. Create a sketch that will magnify an existing image on the screen. When the mouse is clicked, a rectangle is drawn within which the image appears magnified by 2. This magnifying window should be able to be dragged around the screen, magnifying the image beneath it as it moves.
5. Build a sketch that reads an image and saves it as a GIF, JPG, and BMP file. Using an image display utility of your choice, examine those images at various degrees of magnification. What can you observe about the effect of different formats on the image data? (Not all images will illustrate the likely distortions of each format.)
6. Create a sketch that displays an image and allows the user to select a portion of it using the mouse—click, drag, and release will define a rectangular sub-image. This sub-image should now become the whole image, filling the window. When the **Return** key is pressed, the image displayed in the window should be saved as a file named **output.jpg**.
7. Create a sketch that makes bubbles that look more or less real. Bubbles in a liquid are spherical, and get larger as they rise to the surface, where they eventually burst. The bubbles should be created at random times,

having radii that are random within a small range of values. They should rise to the surface and increase in size as they do. (Look up the function `random`.)

8. Take a photograph of a television set, load it onto your computer and edit it so that it is about 200x200 pixels in size. Display this image in two locations in your graphics window. Create a static or *snow* effect and display it so that it appears in one of the television screens. On the other screen, display an image of your choice.

Resources

2D Game Graphics Tutorial: <http://gamebanana.com/tuts/11225>

Intro to 2D Graphics: <http://rbwhitaker.wikidot.com/introduction-to-2d-graphics>

Processing Documentation: <http://processing.org/reference/>

Techniques for Fancy and Lightweight 2D Graphics: <http://www.gameproducer.net/2008/03/03/techniques-for-fancy-and-lightweight-2d-graphics/>

Sprite Database: <http://spritedatabase.net/>

Open Game Art: <http://opengameart.org/>

Bibliography

Kelly, Charles. *Programming 2D Games*. Boca Raton, FL: A.K. Peters, (Taylor & Francis), 2012.

Parker, J. R. *Algorithms for image Processing and Computer Vision*. Second Edition. Indianapolis, IN: Wiley, 2011.

Pile, John Jr. *2D Graphics Programming for Games*. Boca Raton, FL: CRC Press (Taylor and Francis Group), 2013.

Sherrod, Allen. *Game Graphics Programming*. Boston, MA: Course Technology (Cengage Learning), 2008.

Shiffman, Daniel. *Learning Processing*. Burlington, MA: Morgan-Kaufman, 2008.

CHAPTER 3

SOUND

In This Chapter

- Basic audio concepts
- How to use *Minim*
- What to consider in a *Processing* game audio system
- How to use *simpleAudio*

Here is an interesting experiment: play a first person shooter, such as *Halo* or *Half-Life* with the sound off. It is amazing how much of the energy and emotional content is contained in the audio part of a game. The tempo of the music gets the blood racing, and the sounds of weapons fire and nearby explosions can guide you away from trouble or into situations where points can be won. It's not just shooters: play any of your favorite games with the sound off if you need convincing.

Computer games use sound for three basic things:

1. Music. A great deal of emotional content is contained in the music alone. Alfred Hitchcock knew this very well.
2. Sound effects. If a car crashes or a gun fires, we expect to hear that.
3. Speech. Many games tell a story by allowing you to listen in on conversations, or even participate. Your side is often typed in, and is not really understood, but the characters in the game speak and expect to be heard.

It is interesting that many programmers—even those with many years of experience and who know graphics and event-based programming—know almost nothing about how to manipulate and play sounds on a computer. It is especially interesting because sound programming is in many ways much like graphics programming; the goal is to display something. There is object positioning and rendering to be done, the viewer/listener's position affects the result, there are colors/frequencies to be handled, and a special device is at the heart of everything, the video card/sound card.

So it is with some trepidation, and perhaps some excitement, that we begin a trek into the dark, unknown world of computer audio. Like graphics, there can be a lot of math associated with sound—but unlike graphics, some of it is not necessary to perform simple reproduction of sound using a computer. Most games do not create sounds on the fly, but merely read sounds from files and play them at an appropriate moment. Games would be very dull indeed if the approach to graphics was the same. Graphical objects need to be moved, rotated, transformed, and tested for visibility and collisions. Audio objects basically turn on and off, get louder or softer, and perhaps move from the left to the right stereo channel. Display of sounds is in fact simpler than display of graphics. Expect this to change as more options present themselves.

Processing has no built-in scheme for audio display—that is done by a downloadable add-on. The fact that *Processing* is based on Java means that the add-on is coded in *Java* and uses the *JavaSound* API, and could be used with other purely *Java* code. This also means that when we discuss HTML5 in Chapter 9, we'll have to revise the audio display scheme.

There are a number of choices for audio systems, but for the purposes of this book we're going to use *Minim*, the most common option.

Basic Audio Concepts

Although there are similarities between our sense of vision and our sense of hearing, the differences are significant. Most important is the concept that objects that are seen normally reflect light from another source, rather than generating light on their own. Thus, we see by reflected light. Audio, on the other hand, is usually produced by the object that is being sensed; that is, an object that we hear is generating the sound, not reflecting it.

Of course sound reflections can be important, and contribute to the ambience of the sounds. The idea that sound sources are spatially localized is key to positional audio generation, but is less important in stereo and web-based games. In graphics, if we could only observe light sources and not reflections things would be much simpler and less interesting. Also, we don't really have an audio image—a two dimensional pattern that can be interpreted. Instead, we have our ears, two sound receptors, each of which perceives the sum of the sounds that reach them at any particular moment. This can also be thought of as another way that audio is simpler than graphics.

Sounds are essentially vibrations of the air. The intensity of the vibrations is called the *volume or loudness* of the sound. The duration between two consecutive peaks of the vibratory motion is called the *period*, and the number of peaks that occur in a second is called the *frequency*. (Figure 3.1).

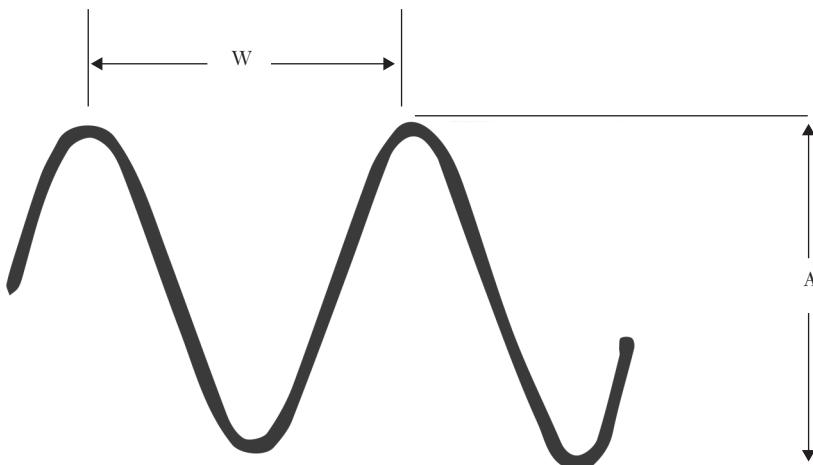


FIGURE 3.1 A sound wave viewed as a graph of intensity versus time. The wavelength **W** is the distance between two peaks; the period is the time between two peaks. The amplitude **A** is the distance between the peak and the trough (lowest point). The frequency is the number of peaks that pass by a stationary point in one second. High pitched sounds have a higher frequency than do lower pitched ones.

The unit of frequency is Hertz (Hz). This name will sound familiar; your computer has an execution speed that is also measured in Hertz—well, megahertz (MHz = million cycles) or gigahertz (GHz = Billion cycles)—and this refers to the number of clock cycles per second.

A typical human can hear sounds that have frequencies between 40 Hz and 15,000 Hz (1,000 = 1 kiloHertz or KHz). Frequencies above 15 KHz are not as important as the lower ones in computer games, as

computer speakers are generally not able to reproduce these sounds, and many people cannot hear them.

We have two ears, and normally any sound presents itself to both of them. There will normally be a slight time difference between the arrival at the left and right ears caused by the distance to the object and the distance between your ears. Essentially, it takes time for the sound to travel the short distance represented by the width of your skull. This is how you locate a sound. Most people can determine a fairly precise location for a sound even with their eyes closed, but only if both ears function properly. This fact is important in a game, because an object that looks like it is at the left side of the screen should also sound like it is to our left.

In day-to-day lives, we are surrounded by sound, and can actually detect much of it. What we hear is really the sum of all of the sounds that reach us at each moment in time. This makes audio rendering simpler than graphical rendering, because the screen requires that we compute the intensity and/or color of at least 640×480 pixels (places). For audio, we need to compute only two audio pixels, one for each ear. However, we need to compute these audio points more often than graphical ones. 24 frames per second is usually enough to realistically represent moving objects on a screen. To render audio realistically, we need to generate a new intensity value at a rate that is at least twice the frequency being created, or up to 30,000 times per second. Fortunately, the sound card can do a lot of the work.

In order to store a sound on a computer it has to be digitized, or *samped*. A standard sound card can do this, if you plug a microphone or other sound source into it. Sound is represented as electrical signals, which can look similar to Figure 3.1 when viewed on an oscilloscope. Sampling a signal involves making a measurement at a regular and frequent intervals. An electrical signal can be measured in voltages, for example, so to store a sound we could measure the voltage being sent to the speakers every millisecond and store this measurement as a binary number on a file. Playing the sound back requires that we can convert binary numbers into voltages again, and send them to an output device.

Without getting into too much detail about signal processing, we need to sample a sound at a rate that is at least twice the highest frequency that we want to reproduce. For example, if we want to be able to hear 15KHz sounds, we have to sample the signal 30,000 times per second. If a sample is an integer (16 bits), this means that a four minute song requires 4×60

$\times 30000 \times 4$ bytes to store (28 MB), and 56 MB if the song is in stereo. Of course, there are compression schemes and such to reduce the size as saved on disk. In any case, what we need is to store these samples as numbers, either integers or floats, and have a means to send them to the sound card.

A standard PC sound card can perform sampling at a high rate, and does so from both the line input and the microphone input. The device that does this is called an *analog to digital (A to D) converter* or *ADC*, and the sound card has some of these. The way the ADC works is much less relevant than the result and the implications. Figure 3.2 shows an example of the sampling process for the sine wave of Figure 3.1. After each sample interval the sound is measured as a voltage measurement and converted into a binary number for storage. Since these numbers are digital, they can't be saved perfectly and so are rounded to the nearest integer, which necessarily creates a small error in each. This is known as a sampling error.

The sound card can reverse the process too, taking a sequence of digital samples and converting them into voltages that can be sent to speakers or an amplifier for playback. Sounds in numeric form can be—and most often are—stored on files, retrieved, and played back as needed. The sound card is a complex and clumsy thing to program directly, and so a software library that does this is essential when developing a game.

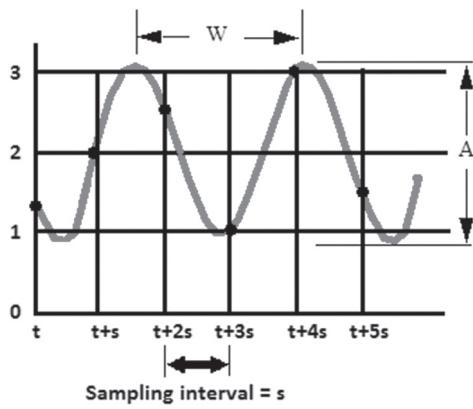


FIGURE 3.2 Sampling of a sine wave by an analog to digital converter. After each fixed interval s the sound is measured as a voltage and converted into a number for storage. Numbers at the bottom show the numeric values of the sound at each sample point.

This is where *Minim* comes in. We can create reasonable sounds for a typical game without needing to know too much of the math or physics of

sound. We also won't need to know too much about our sound card. What we do need to do is understand the paradigm used by the creators of the *Minim* system.

Introduction to *Minim*



Because *Minim* is an *add-on* or *plug-in*, the first thing we need to do is download it and install it so that Processing can use it. You can download *Minim* from the sites identified in the Resources section of this chapter or from the disc for this book. The file downloaded is an archive, probably a ZIP file named something like **minim-2.0.2-full.zip** where **2.0.2** is the number of the current version. This file can be unzipped into a directory of your choice.

Using *Minim* from *Processing* is simple. Run *Processing* and select the **Sketch** menu. From there select **Import Library**, and note that *Minim* is one of the choices. Simply select it.

Should that fail, manually copy the *minim* directory that you downloaded and unzipped into the *Processing* library directory—For example, on a Windows 7 computer this directory is:

```
C:\Program Files\processing-2.0b8\modes\java\libraries
```

The part of the path name that reads **2.0b8** refers to the version of *Processing*, and will change with time as new versions are released. Make certain that the directory is named **minim**, all lower case.

Declaring and Initializing *Minim*

For a *Processing* program to use *Minim*, it must be imported as a library. This is done using a simple declaration at the beginning of the program:

```
import ddf.minim.*;
```

This makes *Minim* available to your code by going to the libraries directory and connecting your program with the code found there. *Minim* is an object, just like *PImage* or *PGraphics*, so the next thing we need to do is create an instance of this object. This means declaring a variable of type **Minim**:

```
Minim minim;
```

Finally we will create the instance, or in other words allocate the storage and facilities for *Minim*:

```
minim = new Minim(this);
```

This statement could appear in `setup()`. Alternatively, the whole thing could be done in the declaration:

```
Minim minim = new Minim(this);
```

Minim is now ready to go.

Using Minim

The Minim package allows your processing code to do four basic things using sound: *play* a sound file, *create* sound dynamically and play it (synthesized audio), *monitor* sound input, and *record* sound to a file. Of these, playing sound files is by far the most common, and so we'll focus on that. You need some understanding of Java libraries here, as all of the operations that *Minim* does will use the Java sound libraries. Details of those can be found on the Java Websites and on educational and training sites listed in the Resources section of this chapter.

Playing Sounds

Sounds are played using the same object that Java uses to play sounds on web pages and in applications, the *AudioPlayer*. For each sound that is to be played, an *AudioPlayer* object must be created in the program. So, using the declarations for Minim in the previous section, let us assume that we have two sound effects to be played: a bouncing sound in a file named **puckBounce.mp3** and the sound of the audience in an arena cheering named **cheer.mp3**. These would obviously be useful for *Hockey Pong*. We need to declare an appropriate *player* for each of these:

```
AudioSnippet puckBounce;
AudioPlayer cheer;
```

Minim will create an appropriate *player* instance for us and read the sound data from the file, but needs to be told what kind of sound we want. We said earlier that sounds can be music, effects, or voice. These sounds are almost always located on files, and we want to play them on demand. The specific way that the files need to be used dictates the type of audio object that will be used to store the sound within the game. There are three types of object, and these types do not correspond with the types we've discussed.

Snippet

A *snippet* is a short sound that is loaded into memory completely at the outset of the program, and is played from there. Thus, large sounds (high

sample rates, long duration) will result in a lot of memory use. All sounds consist of a collection of samples, as an image consists of a collection of pixels, but when using a *snippet*, these samples cannot be accessed by the programmer. Also, while Minim can be used to apply audio effects to some sounds, that is not true of a *snippet*. All we can do to a *snippet* is to play it.

If the sound effect **puckBounce** is going to be implemented as a snippet, then the first thing to be done, probably within the setup function, is to read the file and convert into internal form. For the declarations seen so far, reading the snippet would be done by:

```
puckBounce = minim.loadSnippet ("puckBounce.mp3");
```

The **loadSnippet** function/method of the *Minim* object reads the data from the file. It is the fact that we called this specific function that makes this a snippet as opposed to any of the other kinds of sound. When the sound is to be played, we call the **play** method of the *AudioPlayer* instance **puckBounce**:

```
puckBounce.play();
```

The sound will be played all of the way through whenever **play** is called, and it called as many times as needed. When the game is over, or more importantly when a level is over and the sound is not needed any more, it is critical to return the resources to the system. If the use of **puckBounce** is at an end, call:

```
puckBounce.close();
```

When the need for sound is over, shut down *Minim* and return the resources it is using by calling:

```
Minim.stop();
```

Sample

If you are a musician you know what a *ampler* is—it is a musical device that records short samples of sound and then plays them back on cue, sometimes after being modified. A *sample* in *Minim* is rather like that. It is a short sound—read from a file—that is played when triggered and otherwise ignored. All of the audio is loaded from a file into memory, and then it can be played.

This sounds just like a *snippet* so far, but it differs from a *snippet* in that the individual sound samples are made available to the programmer. This

means that it is possible to modify the sound after each time it is played by changing the numeric values of the samples. *Another difference is that a sample does not use a **play** method, but one called **trigger** instead.* So, if we redeclare **puckBounce** to be a *sample*, we play it as follows:

```
AudioSample puckBounce;
. . .
puckBounce = minim.loadSample ("puckBounce.mp3");
puckBounce.trigger();
```

We still need to call `puckBounce.close()` when we're completely done with that sound.

If we want to access the audio samples we must request a copy, as we did with *PImage* data for pixels. The samples will be real numbers in an array, and there will be a left and a right channel. Let's do that:

```
. . .
// puckBounce is read in, as before.
float[] leftChannel =
puckBounce.getChannel(BufferSize.LEFT);
float[] rightChannel =
puckBounce.getChannel(BufferSize.RIGHT);
```

At this point the values in each array represent the sound sample as real numbers between 0.0 and 1.0 for the left and right stereo channels. If you modify them, the actual values change and will be heard the next time the sound is triggered. So, for instance, if they are divided by 2 the result is that the volume on playback is cut in half.

File

This type of sound object is for long sounds, such as music. The sound is not completely loaded into a buffer as with *snippets* and *samples*, but is instead streamed from its specified location, through a small buffer, and to the display device. This means that a continuous connection to the open file is needed, and if the sound resides on a remote site (I.E. Internet) then latency could be a problem.

The sound variable `cheer` declared above has type *AudioPlayer*, which is what is needed for a sound file. It is loaded using:

```
cheer = loadFile ("cheer.mp3");
```

A file on the Internet can be loaded simply by specifying the URL in place of the file name:

```
cheer=loadFile("http://cdn.epitonic.com/uploads/tracks/Miho_Hattori-Barracuda.mp3");
```

The sound is played and closed just as we did with a *snippet*:

```
cheer.play();
cheer.close();
```

The playing of the sound is done in parallel in a *Java thread*, so it is very important to close the player, otherwise the program can end while the player continues with no easy way to stop it.

Display of Synthesized Sounds

Synthesized sounds are made by filling in an array of data (numbers) in such a way that it appears to be a sound. Technically any data could be sent to the sound card, just as any collection of numbers could be made into an image, but the result is simply noise. It's actually pretty difficult to create a set of numbers that sounds meaningful when played, so *Minim* offers us a collection of classes that make it easier to synthesize sounds. Any one of these objects, referred to as *AudioSignal* objects, can be used to fill a data array with sampled sound data, which can then be played through the sound card. There are many such objects, but a simple and familiar one would be the *SineWave* object which creates a sine-shaped wave of specified amplitude and frequency.

In order to play a synthetic sound we need to create an object that can play what is essentially raw data. An *AudioOutput* object can do that. Any of the previously discussed sound objects come with a play method, but we need to create one to play synthetic sounds.

Let's write a short program to play a sine wave having frequency 440Hz (the musical note A):

```
// Need to import Minim and the signal library
import ddf.minim.*;
import ddf.minim.signals.*;
Minim minim;           // Root of the Minim system
AudioOutput out;        // The sound player
SineWave s;             // The sine wave signal generator
void setup()
{
```

```

    size (512, 200);
    minim = new Minim(this);
    // Create audio output, 2 channels with 4096 sample buffer
    out = minim.getLineOut(Minim.STEREO, 4096);
    // Generate a signal at 440 Hz, amplitude 0.5,
    // at a sample rate of 44100 samples per second
    s = new SineWave (440, .5, 44100);
    // Now 'play' this signal - there can be multiple signals
    out.addSignal (s);
}
// Must be a 'draw' function or program ends too soon.
void draw(){}
// Finalize the sound system
void stop()
{
    out.close();
    super.stop();
}

```

There are a few functions here that could use some illumination. First, creation of an output channel is done using **minim.getLineOut(Minim.STEREO, 4096)**. This output object was saved in a variable **out**, and is used for all sound display. A sine wave shaped signal, which is to say a set of data samples, is created by **s**, which is an instance of the *SineWave* class. We can't see the array of numbers but it does exist, and by specifying a frequency and amplitude we can create any musical note we choose. Playing of this signal is started with the call to **out.addSignal(s)**, meaning add the signal that **s** generates to the set of signals being played by **out**.

The output can play many signals at once. To play a chord, for example, replace the code:

```

s = new SineWave (440, .5, 44100);
out.addSignal (s);
with
s = new SineWave (440, .5, 44100);           // A
out.addSignal (s);
s = new SineWave (554.37, .5, 44100);       // C#
out.addSignal (s);
s = new SineWave (659.255, .5, 44100);      // E
out.addSignal (s);

```

This will play three notes at once: an A, a C#, and an E, which is an A major chord.

There are other synthetic sounds to use in addition to the sine wave: Oscillator, PinkNoise, PulseWave, SawWave, SignalChain, SineWave, SquareWave, TriangleWave, and WhiteNoise. Details of these can be found in the documentation for the `AudioSignal` class within the *Minim documentation* – see the Resources at the end of the chapter.

Monitoring Input

You may think of a desktop computer as having an *audio bus* through which all sound data ultimately flows to the speakers. The microphone might feed into this, as would the line input of the sound card and music playing from a Website or from a CD. *Minim* has access to this flow of sound data. This is one of the least useful aspects of *Minim*, but it is there and we'll look at it for a moment.

To examine the sound playing on the computer, have *Minim* return an `AudioInput` object, and then simply look at what it gives you. The following function call asks for one of these, and specifies that it is stereo (23 channels) and that you want a buffer 1024 samples in length:

```
in = minim.getLineIn(Minim.STEREO, 1024)
```

Data is now flowing into your buffer. The `AudioInput` variable `in` has a left and a right channel, and data can be retrieved using a call to the function `get()`. So, for example, `in.left.get(122)` will return the 122nd sample value in the left buffer right now.

As a complete example, here is a short program that monitors the audio and displays two rectangles, one for each of the left and right channels, whose height is the current sample value:

```
import ddf.minim.*;

Minim minim;
AudioInput in;

void setup()
{
    size(512, 300);
    minim = new Minim(this);
    in = minim.getLineIn(Minim.STEREO, 1024);
    stroke(255);
    fill (100);
}
```

```

void draw()
{
    float ly, ry;

    background(0);
    ly = ry = 0;
    ly += in.left.get(128)+ 1.0;
    ry += in.right.get(128)+1.0;
    rect (100, 200, 20, -ly*100);
    rect (200, 200, 20, -ry*100);
}

void stop()
{
    in.close();
    minim.stop();
    super.stop();
}

```

One of the values in each buffer is selected as a representative of the current time, and a filled rectangle is drawn using a multiple of that value as the height. The result looks like the volume bars commonly seen on amplifiers and receivers.

Recording Sounds to a File

This is something we almost never want to do, but in those rare cases where it is important, it's done using a few functions in addition to those used for monitoring. Most importantly, we need to create an *AudioRecorder* and feed data from an *AudioInput* to it. So, first:

```

AudioRecorder r;
AudioInput in;

```

Then create instances in *setup*:

```

in = minim (Minim.STEREO, 512);
r = minim.createRecorder (in, "myrecording.wav", true);

```

The *createRecorder* function creates and returns an *AudioRecorder*, in this case using *in* as the source of the sounds, and saving to a file named **myrecording.wav**. Begin recording audio by:

```
r.beginRecord();
```

Stop recording by:

```
r.endRecord();
```

Save to the file by:

```
r.save();
```

Don't forget to close after all is done.

Minim and Java Sound Support

Minim is coded in *Java* and is based on exiting *Java* classes. In particular, it is important to know something about the underlying *Java* support because some essential operations like volume and balance control are done using *Java*, not *Minim*, functions. The hierarchy of the sound system is not obvious. We have already seen many of the most important *Minim* functions, the ones for creating and loading *snippets* and *samples*.

The *AudioSample* class is the parent of *sample*, and *AudioSnippet* is the parent class of snippet. The *AudioSnippet* has a large set of tools for positioning the point in the sound that is currently playing. They are:

<code>cue(m)</code>	Set the play position to m milliseconds from the start
<code>isLooping()</code>	Return true if sound is looping, false otherwise
<code>isPlaying()</code>	Return true if this <i>snippet</i> is <i>playing</i>
<code>length()</code>	Return the total length of the snippet in milliseconds
<code>loop()</code>	Set looping and start playing
<code>loop(n)</code>	Set to loop N times
<code>loopCount()</code>	Return number of loops remaining
<code>pause()</code>	Stop playback and remember the current position
<code>play()</code>	Start playback from the current position (initially the start)
<code>play(m)</code>	Start playback m milliseconds from the start
<code>position()</code>	Return how much of the sound has been played
<code>rewind()</code>	Set play position to the beginning
<code>skip(m)</code>	Skip m milliseconds from the current position

The *AudioSample* has none of these, but *AudioPlayer* does. However, both of these classes extend the *AudioSource* class which is what gives us access to the raw sound data. In particular, *AudioSource* offers:

<code>buffersize ()</code>	Return the size of the buffer
<code>samplerate()</code>	Return the number of samples per second
<code>type()</code>	Return Minim.MONO or Minim.STEREO

There are other methods that refer to any effects that are being applied to the sounds.

A snippet extends a class named *Controller*; *AudioSource* extends *Controller*, and *AudioSample* and *AudioPlayer* extend *AudioSource*, so all of these ultimately provide the functionality of the *Controller* class. This gives all of them the following methods:

<code>float GetBalance()</code>	Return the balance (left/right) from -1 to +1
<code>float getGain()</code>	Return the gain. This is essentially volume, but the range may not be what you expect. Check by calling <code>printControls</code> .
<code>float getPan()</code>	Return the pan, essentially balance, from -1 to 1
<code>isMuted()</code>	Return true if the sound is muted
<code>mute()</code>	Mute this sound
<code>printControls()</code>	Print to the console the controls that exist and their ranges
<code>setBalance(b)</code>	Set the balance (left/right) from -1 to +1
<code>unmute()</code>	Unmute this sound

A Processing Game Audio System

If we were to design a system to support game audio, it would have to support the basic sound display operations in a portable fashion. That is—while we have looked specifically at *Minim* as the software underlying the sound display—a proper game audio system should be able to be used with any such package. The specifics of *Minim* should be hidden beneath the functional layer provided by our software. Let's look at one possible design for such a system.

What are the essential operations? We need to open a sound file, read it, and associate it with a name. The sounds can be classed as sound effects or music, and we should be able to decide whether we need access to the samples. We must be able to play, stop, and close a sound, optionally with position cueing. We should be able to set value, balance, and tone. And, finally, some degree of positional audio should be available.

Positional Audio

A positional audio display is an attempt to simulate the fact that a human can determine the location of a sound source. The person listening is located at some point in front of a pair of speakers. The simulation must present the sounds at those speakers as they would be at the listener's ears if they were present in the real scene. This is often done using relative volumes rather than time differences; that is, the source is nearer to one of the listener's ears than the other, and so the sound will be a little louder in that ear (or speaker).

It's also true that a sound that occurs nearby seems louder than one occurring a distance away, all other things being equal. We use this fact instinctively when judging the distances in real life, and it should be true in games as well. This is a basic aspect of positional audio, one that everyone perceives on a daily basis, and requiring only distance to the source and not the precise position.

What is needed for positional sound to be possible is the position of the sound event, which is to say the location within the game space of the thing that generates the sound. Collisions, for example, happen at a precise location. The position and facing direction of the player, or the avatar at least, is also known, and the relative intensities of the sound at each ear can be calculated and displayed at each speaker. It is clear that what's needed is the position of the sound, the position of the player (listener), and the direction the player is facing, in the game.

We can assume that the player's position is known, since it is being updated by the game each frame. The sound position is also known as most events have a known position within the game. The player's facing direction can be assumed to be the direction of motion, or if stationary then the last known direction of motion.

Example: Distance Attenuation

In computer graphics, clipping is the act of removing lines and polygons that are outside of the viewing volume. This includes lines that are too near the camera, and lines that are too far away. The near and far clipping planes are defined as distances; we shall do the same with sounds. At some sufficiently far distance d , a sound can no longer be heard, and will be *attenuated* (i.e. reduced in intensity) completely (100%). At some sufficiently near distance the sound will not be attenuated at all (0%), and at every distance in between the sound will be attenuated by some function of distance. In real life, the function is related to $1/d^2$ —the sound gets fainter as the square of the distance to it. The important thing in a game is that things seem correct rather than being correct, and using the true degree of attenuation may be seem too great. A linear function may seem more realistic.

Let's define variables `maxSoundDistance` and `minSoundDistance` to be the 100% and the 0% attenuation distances respectively. Then, if d is the actual distance to the sound, the attenuation a can be calculated as follows, where a will have a value between 0.0 and 1.0:

```
a = (d-minSoundDistance) / (maxSoundDistance-minSoundDistance)
```

Every sound will have a natural or intrinsic volume at which it is played, as well as a minimum and maximum. This intrinsic volume will be modified by multiplying by the attenuation before it is played.

In *Minim*, each device or sound type has a predetermined set of gain settings that apply to it, a `minGain` and `maxGain`. The attenuation will be used to set the gain to the proper value between these two points. Gain almost always has a minimum value of -80 (`minGain`) and a maximum of 6 (`maxGain`), for some reason. The gain setting would be, for any value of a :

```
g = (gainMax-gainMin) *a+gainMin
```

Finally, the gain for the sound is set for gain g and its `player` by the following, or something similar:

```
player.setGain(g)
```

If we want to play a sound, we could just call `play`. Or, to play a sound with distance attenuation we could use the following function:

```
// (x,y) is sound position; (lx, ly) is avatar position
void playDistance (int x, int y, int lx, int ly,
                    AudioPlayer player)
```

```
{
    float a, g, d;
    d = sqrt ((x-lx)*(x-lx) + (y-ly)*(y-ly)); // Distance
    a = (d-minSoundDistance)/
        (maxSoundDistance-minSoundDistance);
    g = (gainMax-gainMin)*a+gainMin;
    player.setGain(g);
}
```

This function would be need to be rewritten for the library to allow for sound objects and their positions, but is otherwise perfectly functional.

Example: 2D Positional Sound

A positional audio display for a computer game cannot use time differences to specify the source of a sound. A listener will be in an uncontrolled environment, sitting in an unknown position relative to the speakers. This has a significant effect on the time it takes for the sound to reach each ear. In addition, both of a listener's ears would hear the sound from both speakers, and the time delay might not be perceived or might be heard as an echo. Using a volume difference is not as accurate but does work well when the player is not wearing headphones.

A computer game knows where the sound source is, and what it attempts to calculate is how that should sound at your ears. It then sends those sounds to the left and right channels of a stereo sound system—this would be the N speakers of a 5 or 7 channel sound system, but for the purposes of this example, we will use stereo. The game sound system depends on the spatial separation of the electronic sound system and the ability to set volume levels on the left and right channels to *simulate* how the event should sound. We have to calculate those two sound levels.

To figure out how to do the calculation, we need to decide how to tell where the sound location is relative to the player's avatar and the direction that it is facing. We will find the angle between the player and the sound and use that to adjust the pan control. The player's location is known; the facing direction must be known too, and will be an angle in the same coordinate system. Remember that *Processing* uses a system that has 0 degrees as screen right, and 90 degrees as the screen up direction. With the player at position (x, y) we'll define a point (faceX, faceY) that corresponds to an imaginary point to that the player is facing. This point will be:

```
faceX = (int)(X + cos(facingDirection) * d)
faceY = (int)(Y + sin(facingDirection) * d)
```

The value of `d` above is a distance from the player, and can be anything that provides a long distance; the value `d=40` works pretty well.

Figure 3.3 shows something of the geometry of the situation. The left of Figure 3.3 shows a diagram of a player in two extreme situations—one where the sound is precisely to the right, the other where the sound is directly ahead. In the first case the sound is at maximum loudness in the right ear and the minimum in the left, and in the second the loudness should be the same in both ears. As the sound moves along an imaginary curve from the first point to the second, the pan between right and left channel also changes. This describes what we want to do. Figure 3.3 (right) shows a more abstract geometry, where the player and the facing direction are used to determine the relative angle to the sound.

The math and the programming is a bit complicated (See Appendix I for the relevant mathematics), but the basic steps for determining a pan value from the positions of the sound source and the listener are as follows:

1. Calculate the angle between the facing point, the listener, and the source. In the code provided on the Website for this chapter, the following function does this:

```
float angle_3pt (x1, y1, x2,y2, x3, y3);
```

It is past the coordinates of the three points that define the angle, with the listener being the center and returns an angle, in degrees.

2. Determine what side of the line defined by the listener and the facing point the sound source is on. This is done using the line equation and plugging in the x,y values of the source—if the result is positive it's on one side, negative and it's on the other. The function that does this is called `int whichSide`:

```
int whichSide (x1, y1, x2,y2, x3, y3);
```

It returns `-1` if the source is on the left, `+1` if it is on the right.

The combination of these two values tells everything we need to know about the orientation of the listener with respect to the source. The following function returns the product of `angle_3pt` and `whichSide`:

```
float sourceAngle (x1, y1, x2,y2, x3, y3);
```

This angle has to be mapped onto a pan value between -1 and +1—a `sourceAngle` value of -90 is a pan of value -1, a `sourceAngle` value of 0 is a pan of 0, `sourceAngle` value of 180 is a pan of 1, `sourceAngle` value of 360 is a pan of 0 again. Values in between can be interpolated, but the use of a pre-computed table can eliminate repeated calculation (the computation is done 30 times a second).

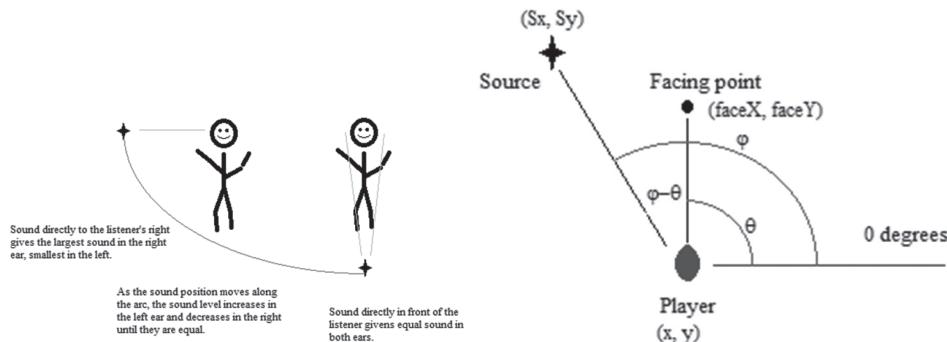


FIGURE 3.3 (Left) The listener geometry of positional sound. The ear that faces the source most directly gets a larger fraction of the sound. (Right) The more technical geometry of that situation. We need to determine the angle $\varphi-\theta$ and need to know what side (left or right) the source is on.

The `simpleAudio` Class

By using the facilities that *Minim* offers, and by using the basics of positional audio that we've figured out, it is possible to code a basic positional audio class for use in games. The architecture is based loosely on that of *openAL*, a cross platform open source sound library originally built by *Loki Software* so help with porting *Windows* games to *Linux*. A game implementation in *openAL* or *simpleAudio* has a *listener*, which represents the player or their avatar, and a collection of *sources*. The game loop must update the position of the listener after each iteration, and would have to update the facing direction as well. Sources can be more fixed but can also move if needed, and can have their value turned on or off and be subject to all of the operations allowed by *Minim*.

We need to open a sound file, read it, and associate it with a name. The sounds can be classed as sound effects or music, and we should be able to decide whether we need access to the samples. We must be able to play, stop, and close a sound, optionally with position cueing. We should be able to set value, balance, and tone. And, finally, some degree of positional audio should be available.

Using *simpleAudio*

The *simpleAudio* class uses *Minim*, so it has to be installed before anything can be expected to work. After installation, none of *Minim* has to be used directly. As an example, let's write a program to place a single sound at a particular point in 2D space, and allow a user to move around that point using keys. The facing direction can be changed too.

The first thing needed is an instance of *simpleAudio*, just as we have to create an instance of *Minim* in previous examples. Declare a global variable for the instance and then create one inside of `setup()`:

```
simpleAudio z;           // Audio system
void setup()
{
    size (500, 500);
    z = new simpleAudio(this);
}
Now create a sound source. simpleAudio provide a function for this:
sound1 = z.addSound ("a440.mp3", 100, 100,
                     source.STREAM);
```

The first parameter to *addSound* is the name of the sound file. Next is the position—on a 2D grid—of the sound. Then the type of the sound is given—in this case a *stream*, but it could also have been a *snippet*. The function returns an integer that identifies the sound, and is used to play it, stop it, and otherwise manipulate it.

After the sound is created the system knows about it, and it can be played:

```
z.loop(sound1);
```

This will play the sound as a loop. Only some kinds of sound can be looped, but *simpleAudio* knows what kind of sound was created and calls the correct function. The sound is played as if it was positioned in space at location (100, 100). It is useful to set clipping boundaries so it can't be heard past a certain distance. This is done as follows:

```
z.setClip (10, 1000);
```

This sets the volume levels to that at distances less than 10 the volume is a maximum, and at distances greater than 1000 it can't be heard at all. That's all of the initialization that needs to be performed, and `setup()` is complete.

The `draw()` function would normally call the game loop, but in this example program it does a few simple things: it draws a circle at the positions of the source and the listener, it updates the listener's position, and it calculates the facing point and draws a short line from the listener in that direction so that we can see the facing direction. That code is:

```
void draw ()
{
    float dtor = 3.1415926535/180.0;
    background (0);
    // Draw the source(s)
    ellipse (z.posX(sound1), z.posY(sound1), 20, 20);
    ellipse (z.posX(sound2), z.posY(sound2), 20, 20);

    // Draw listener
    ellipse (posX, posY, 20, 20);
    faceX = (int) (posX + cos(facingAngle*dtor) * 40 + 0.5);
    faceY = (int) (posY + sin(facingAngle*dtor) * 40 + 0.5);
    line (z.listenerX, z.listenerY, faceX, faceY);
    //Update the listener position.
    z.updateListener(posX, posY, faceX, faceY);
}
```

The only new function here is `updateListener`, which is passed the current listener position and facing direction or point. Finally there is a simple handler for the keyboard:

```
void keyPressed()
{
    if (key == CODED)
    {
        if (keyCode == LEFT) facingAngle += 5;
        else if (keyCode == RIGHT) facingAngle -= 5;
    }
    else if (key=='w') posY -= 5;
    else if (key=='s') posY += 5;
    else if (key=='d') posx += 5;
    else if (key=='a') posx -= 5;
}
```

This code changes the facing angle if the left or right arrow key is pressed, and changed the listener (or player) position as the W, A, S, and D keys are pressed. The player can move the listener around the source to hear the positional effect of the motion. The program is **chapter0308.pde**,

and should execute on any system with *Processing* and *Minim* properly installed.

Another way to illustrate the effect is to leave the listener in one place and move the sound. The program **chapter0309.pde** does exactly that. Sources can be repositioned by calling `updateSource (int i, int x, int y)`. The example program allows the user to change the position of the source rather than the listener by typing the same keys as before.

Exercises

The problems below will apply your knowledge of the material in this chapter, and sometimes require that you do some more research before you are able to complete them. Solutions to selected exercises are available on the companion disc.



- 1.** You have 5 minutes of stereo recording, sampled at 11,025 KHz and 16 bits per sample, uncompressed. How big is the file?
- 2.** Given any mp3 file, write a sketch that will read and play the file. Display the time remaining to be played on the screen.
- 3.** Implement a pan control for the solution to Exercise 2—when the mouse is on the left side of the screen, the sound should play only on the left speaker, and as the mouse is moved to the right, the sound is shared between the speakers, gradually moving to the right speaker.
- 4.** Create a simple keyboard that plays the basic notes starting at A (440 Hz). Each note should be played when an appropriate key is pressed: A, B, C etc. The note frequencies are: A (440) B(493.9) C(523.3) D(587.3) E(659.3) F(698.5) G(784.0).
- 5.** Finish the sound recorder example. When the R key is pressed, it should begin recording, and when pressed again it should stop recording. It should save the recorded sound to a file when the S key is pressed. Indicate that recording is taking place with a message or other obvious sign.
- 6.** Use *Minim* to construct a visual/auditory demo of distance attenuation. Let the sound source be represented by a circle and the position of the listener be represented by a second circle, drawn at the current mouse

position. The volume with which the sound is be played (any file you like) should be a function of the distance between the two circles.

7. Use *simpleAudio* to build a sketch similar to that of Exercise 6, but now have two sound sources indicated by two circles drawn a few hundred pixels apart. Both sounds should play simultaneously, and the volume of each should be a function of the distance between the mouse position and the circle representing that sound. You can mix the sound levels relative to each other by moving the mouse about.
8. Locate a recording of a hockey game on the Internet or record the sound from your television. Using Audacity, GoldWave, or a similar sound editor, locate a clean instance of a puck hitting the boards. Extract this into its own file, and clean it up using whatever filters you choose so that you think it sounds good. Edit the beginning and end so that the sound clip plays immediately when the file is started. Save this file for the exercises in the next chapter.

Resources

Minim download: <http://code.compartmental.net/tools/minim/>, <http://processing.org/reference/libraries/>

Top-level *Minim documentation:* <http://code.compartmental.net/minim/javadoc/>

Javasound documentation: http://docs.oracle.com/javase/6/docs/technotes/guides/sound/programmer_guide/contents.html

Minim audio signal documentation: <http://code.compartmental.net/minim/javadoc/ddf/minim/AudioSignal.html>

Audacity: <http://audacity.sourceforge.net/download/>

GoldWave: <http://www.goldwave.ca/>

Free sound effects: <http://www.grsites.com/archive/sounds/>

Bibliography

Collins, K. *Game Sound: An Introduction to the History, Theory and Practice of Video Game Music and Sound Design*. Cambridge, MA: MIT Press, 2008.

Crawford, C. *The Art of Computer Game Design*. Berkeley, CA: McGraw-Hill/Osborne Media, 1984.

- Friberg, Johnny, and Dan Gärdenfors. "Audio Games: New Perspectives On Game Audio." *Proceedings of the 2004 ACM SIGCHI International Conference on Advances in Computer Entertainment Technology*. 2004.
- Heerema, J., and J.R. Parker. "Music as a Game Controller." *IEEE International Games Innovation Conference 2013*. Vancouver, BC, 2013.
- Long, Ben. *The Insiders Guide to Music and Sound for Mobile Games*. 2012.
- Parker, J.R., and John Heerema. "Audio Interaction in Computer Mediated Games." *International Journal of Computer Game Technology* (International Journal of Computer Game Technology), 2008.
- Stevens, Richard, and Dave Raybould. *The Game Audio Tutorial: A Practical Guide to Sound and Music for Interactive Games*. Burlington, MA: Focal Press (Elsevier), 2011.

CHAPTER 4

HOCKEY PONG: A 2D GAME

In This Chapter

- Using prototypes to implement a game
- Testing
- An example Design Document

Having looked at the internal structure of a game, some basic graphics, and audio, we now have the tools at our disposal to build a complete 2D game. We did a high concept for *Hockey Pong* in Chapter 1, so let's complete that game as an example. One should never jump right into coding at the very onset of a project, because we don't know at that time where we are going. On the other hand a degree of organization and discipline are needed, and iterative prototyping is a good way to structure a project in game development—create a playable game as soon as possible, and then play it, taking note of merits and deficiencies. Then use that information to make a second improved version, and play it again, repeating until it is excellent, or until you run out of time.

Game developers work from documents. The high concept can be a sales device, not a working development document. The most important thing to have when building a game is a *game design document* (GDD), which is really a blueprint of what the proposed game will be. There are

many forms of GDD, but all have some basic things in common. They must describe the game in enough detail to implement it unambiguously. In most game development companies, there is a team building a game, and that team will each work from the same GDD. It defines the goal.

So, we should build a GDD for *Hockey Pong* and then stick to it when building the game, just as is done in real life. This is a simple game so the document will be short—the GDD for a major game can be hundreds of pages long. The GDD for *Hockey Pong* is at the end of this chapter, and should be referenced while the game is developed in the following sections.

Implementing the Game Using Prototypes

In traditional software development, it is not uncommon to have a complete design document before starting the coding part of the project. *Don't write any code until you have a spec* is what they teach at school. When developing a game, it can be very useful to have a malleable set of executable prototypes at the very beginning. These are executable but not functional, if that makes sense; the game implemented is a primitive one that has only the main feature or two working. The purpose is many-fold, but first: to allow the client, the person contracting for the game, to get a visual feel for what is being proposed. It's one thing to say that we're building a *Pong* variant in the style of a hockey game, but is quite another to see it on the screen.

The design document also gives us an idea how complex the project is. Many developers have the ability, after decades of practice, to conceptualize games in their heads. However, seeing everything mapped out—the game surface, the size of the parts, the speed of the objects, the colors—can spark new ideas, identify places where things could get difficult, and generally helps get the project off to a good start. Later prototypes allow the testing of new ideas, to address efficiency matters, and to try out new art and sound. Final versions are play tested so as to ensure that the final product is as much fun as possible.

Prototype 0

This first tentative version is mainly for an initial evaluation in-house. In this particular case, the basic code only took about 30 minutes to create. This version gives only a basic feel—the game play area is displayed, a puck moves around the surface, and it bounces off of the sides. Basic parameters

(puck position and direction) are displayed in a small text area under the playing surface. Figure 4.1 shows a screen capture from this version.

Does this look like what we want? Is the window big enough? Is the puck too small? There is no sound yet, no interaction, and only 27 lines of code.

This is pretty impressive, really. In *C* or even *Java* it would have been very difficult to create this in under an hour, and the number of lines of code would grow enormously. The things that *Processing* gives us are the things that are not as interesting to code and that take a lot of time: window management, graphics, animation, and interaction.

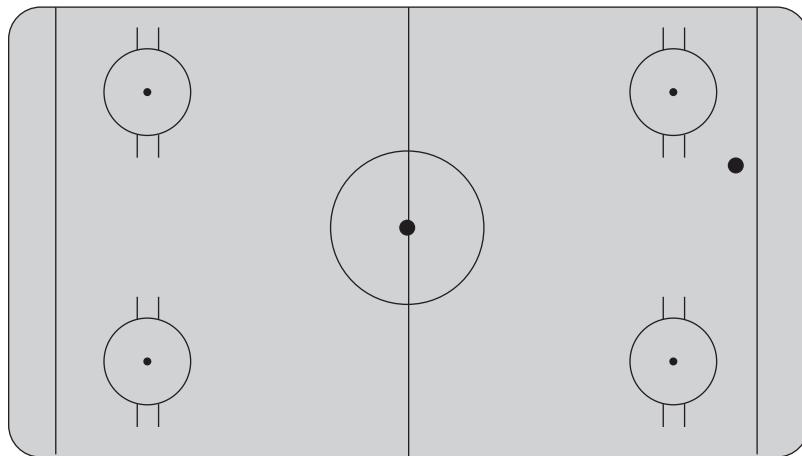


FIGURE 4.1 The first prototype game screen

Prototype 1

Prototype 1 is the first shown to the client. It has the suite of screens that will be used in the final game, if not the actual art that will be in place. It has paddles that can be controlled, and goal posts. This is a better example of how the game will look, and can be given to someone outside of the development group for comments.

The art for the screens exists, even if it is preliminary. The buttons on the screens work, and so transitions between screens can be illustrated. The game itself has not progressed much, but the entire system seems more

finished. The amount of code needed to implement the screens and the buttons is significant: prototype 1 has about eleven times the amount of code as did the previous version. It took over 18 hours to build, including the art. The artwork, it turns out, and positioning buttons took the lion's share of the time involved.

Screens

According to the game design document, there are to be four different screens used in the game—a start screen, an options screen, a play screen, and an exit screen.

The consequences for the code are that each screen corresponds to a different state in the display, and enables distinct activities in the game code itself. The keyboard, for instance, has no effect on any of the screens except the play screen, where it controls the paddles. The mouse has no impact on the play screen, but is used on the other screens to select an option or screen transition. And, of course, quite different graphics are displayed in each screen.

A simple finite state machine can be used to keep track of things. The state is the screen being displayed, and the transitions are controlled by the mouse and the game play itself. There is a screen transition diagram on page 110, but it is a small enough system to keep in your head. The **start** screen takes you to the **options**, **end**, or **play** screen. From each of those you can return to the **start** screen.

So, the screens can be numbered: start=0, options=1, play=2, and end =3. These are state numbers, and in the `draw` function we use the screen state number to display the correct screen. Each screen will be displayed by a distinct function (`startScreen()`, `optionScreen()`, etc) so the body of the `draw` function will look like this:

```
switch (screenState)
{
    case startState:
        startScreen();
        break;
    case optionState:
        optionScreen();
        break;
    case playState:
        playScreen(); break;
```

```

        case endState:
            endScreen();  break;
        default:
            error (NO SUCH SCREEN);  break;
    }
}

```

In this way the state we're in is used to draw the screen each time a new frame is drawn. Screen transitions are done in the mouse handler `mouseReleased`, and the code looks very much like that above. If the mouse is clicked in a button on a screen, the value of `screenState` is changed.

Buttons

A *button* is not a thing that *Processing* gives us, so we have to implement it ourselves. It's really just a region, usually rectangular, that responds in a particular way to a mouse click. The button has a label that reflects its function, so we speak of *start* or *play buttons*. When the mouse is pressed or released, *Processing* calls a function named `mousePressed` or `mouseReleased` respectively, if those functions are defined. So, if `mouseReleased` is called and the coordinates of the mouse are within the bounds of the rectangle defined by the button, then the button was said to have been pressed.

The start screen has three buttons—‘Options’, ‘Play’, and ‘Quit’. The ‘Options’ button has upper left window coordinates (56,398) and lower right coordinates (170, 428). The code that implements this mouse click within the `mouseReleased` function would look like this:

```

x = mouseX; y = mouseY;
if (y > 398 && y < 428)
    if (x>56 && x<170)
        // Button 'Options' was pressed

```

Each button in the game occurs on a specific screen, and has (x,y) coordinates for the upper left and lower right that define it uniquely.

Start Screen

When a player starts running the game, the *start* screen appears. This screen is illustrated in Figure 4.2. It shows a graphic background (a hockey player and an ice surface) and three buttons that allow transitions to the other screens. Many games and other interactive software that have buttons display when the button is *armed* (i.e. the mouse is over the button and a click will activate it) by changing the color or the font, or showing the fact graphically somehow. In order to accomplish that, the buttons should

be small images rather than simple text drawn on the screen. Each button has two images to represent it—one for the normal button, and one for the armed button. In Figure 4.2 the *Play* button is armed.



FIGURE 4.2 The Start screen.

When displaying the **Start** window, if the mouse coordinates lie inside a button then the armed version of the button is displayed, otherwise the normal version is used. This is helpful to the player because it indicates that the button is ready to be used.

When the mouse button is pressed while the coordinates of the cursor are inside of one of these buttons, a transition is made to another window simply by assigning a new value to the `screenState` variable.

Options Screen

When the player selects **Options** the game makes the transition to the options screen (`screenState = optionState`). This screen presents the player with the set of user selected parameters that can be chosen. This includes a choice of a 1 or 2 player game, the ability to turn the sound off, and the ability to select a home team from a small list.

Again, the buttons that allow a choice are small rectangular regions implemented as images. When the user clicks on the **Single Player** button, it is replaced by **2 Players**, and back if clicked again, so that the current selection is visible on the screen at all times. This does not work in the case of the team selection, because all teams have to be visible to make a choice, so the selected team's logo will be the first one in the list. Clicking

the **Back** button takes the player back to the **Start** screen (`screenState = startState`).

In the single player game, only the left paddle is under player control, and it responds to the W and S keys. Selecting the two-player game sets an internal flag that will turn off the AI player and enable the arrow keys of keys to function as the control of the right paddle. The AI is not implemented yet.



FIGURE 4.3 The Options screen.

When the sound is turned off, another flag is set (to false) which means that all calls to function that play sounds are disabled. In this prototype there are no sounds yet, so no actual implementation details are available.

The team selection again uses small images to indicate buttons. The button pressed here selects a logo to display on the ice (game surface). The one selected will move to the left side of the row and will increase in size on the window. The logos represent either actual very old teams or invented logos, so as to avoid copyright issues. In some games, a royalty is paid so that actual team logos can be used, but for a game of this magnitude that would seem impractical. The buttons are not active in this prototype.

Play Screen

The **Play** screen looks like Figure 4.1. No visible change has been made to the play of the game as implemented even though the design has advanced, because the changes have taken place in the subsidiary aspects—art, and screens for the most part. However, this is the screen that is displayed when the **Play** button is selected, where it was the only screen available in the initial prototype.

End Screen

The **End** screen is simply informative, giving game credits and contact information. A click anywhere in this window will terminate the game program (Figure 4.4).



FIGURE 4.4 The End screen, showing game credits.

Prototype 2

After prototype 1 has been assessed and agreed to by the client or the design team, then next step is to develop the game features in detail. This means that the screen development is considered to be complete, and all changes will be seen on the game screen only. There are three major issues to be addressed in this prototype: user control (including score), sound, and the game AI. There is also an omission to be corrected: the options screen does not have a **Back** button, and so is a dead end. One will be added.

User Control

This involves making the software connection between the key presses and the position of the paddles. It appears to be a simple matter, but there are important issues to resolve. Specifically, how fast should the paddle move, and what are the limits of its motion?

The motion is implemented within the `keyPressed` and `keyReleased` functions. For example, pressing the W key begins moving the left paddle upwards; the motion stops when the paddle reaches the upper limit of motion, or when the player releases the key.

Sound

Now that the user control system is being implemented it makes sense to assign audio events to events in the game. There are only a few audio events in this game, but it is important to give them sensible sound effects, ones that a hockey fan would recognize. In particular:

- There will be an ambient sound of an arena audience played for each period of the game.
- There will be sound effects of bounces against the rink boards.
- There will be sound effects of bounces from the paddles.
- There is a horn that indicates the start of the game, and one that indicates the end of the period.
- There are sounds associated with each face-off.
- There could be extra audience sounds, like horns and cheers, which can be played at random.

Ambience

The ambient sounds were obtained by recording sound at a hockey game and then editing it into relevant portions. Sound editing is done using *Audacity*, a full featured downloadable sound editor having all of the basic tools needed for creating game sound. Ambience is created by editing the sound into one minute blocks that contain few unique occurrences, like horns or whistles or people yelling. These can simply be cut out, because the sound of a crowd is like white noise, and the edits will be nearly inaudible. Figure 4.5 shows a screen from *Audacity* where a section of the audio track being edited has been highlighted for deletion. That section corresponds to someone in the audience yelling something incomprehensible. If it were not removed then it would be heard every couple of times the game was played, and combined with all of the other repetitive instances that would be there the repetition would be distracting.

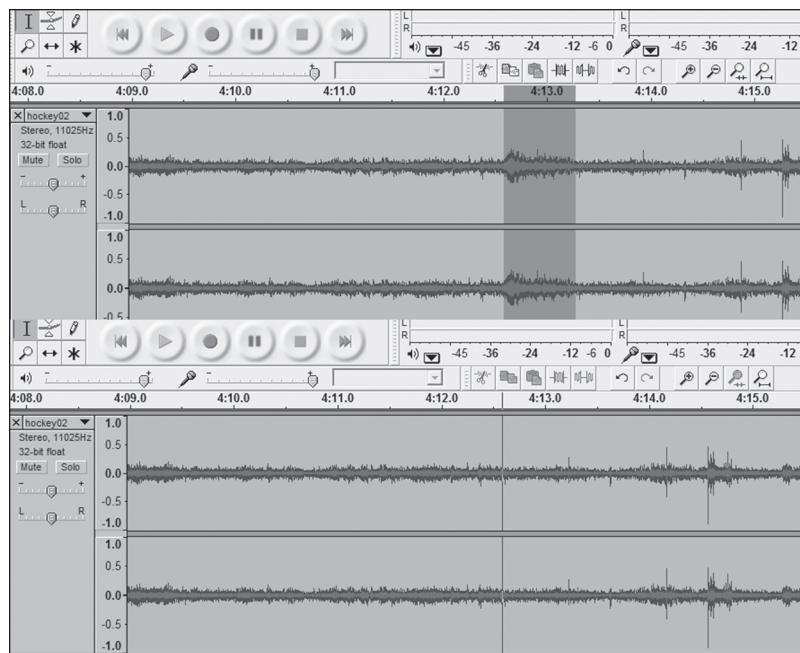


FIGURE 4.5 (Top) A small section of sound is marked for removal using the *Audacity* editor. (Bottom) After removal, the place where the piece was taken out can't be easily determined.

The files with ambient audience sound are each about one minute in length. When one is finished another one, selected at random, is started. The time remaining in a sound is:

```
t = (float)(z.length(ambient[playing]) -  
z.position(ambient[playing])) / 1000.0;
```

where `playing` is the index of the sound playing at the moment, and the functions `length` and `position` return the length of the sound and the current position being played, in milliseconds.

Any game needs six such files. This game has ten to minimize the effect of repetition. Each one-minute file is about 480 K in size.

Impacts

Sounds of impacts and face-offs were recorded in an empty arena using real players, some pucks, and a recording device (an *iPod* with a Belkin *TuneTalk* recording unit). When played over the ambient sound the impacts sound as if they were occurring during the game.

Each sound has multiple instances, and an instance is selected at random when playback is needed. This reduces the repetitive nature of sound. The same event rarely sounds exactly the same twice in real life, but frequently a game uses only one sound file for a particular event. So for example, there are ten distinct sounds used for the impact of the puck with the boards. If these sounds are stored in an array `soundImpact[]`, then any particular impact will result in the sound `soundImpact[(int)random(10)]` being played.

For example, when the puck hits the sideboards two things happen: the puck changes Y direction, and a sound is played. The code is:

```
if (puckY<3 || puckY>443)                                // Side boards?
{
    if (soundOn==0)
        z.play(puckBoards[(int)random(10)]); // play sound
    dy = -dy;                                              // Bounce
}
```

In the code above, the variable `z` is the *simpleAudio* instance created for the game, `puckBoards` is the array of sound indices for the sound of the puck hitting the boards, `puckY` is the vertical (Y) coordinate of the puck, and `dy` is the vertical speed of the puck.

Artificial Intelligence

The Artificial Intelligence portion of this game controls the right paddles in a single player game, and introduces randomness to bounces to make them less predictable. The other functions usually found in game AI are not required in a game as simple as this one.

Random Bounces

Whenever the puck hits the boards or a paddle it can change speed in both the x and y direction by a little. This means that the direction it is moving in can change if the change in x and y values are not the same. If a collision takes place, a random change in x and y speeds between -2 and +2 is generated. The change in x speed is coded as:

```
dx = dx + ((int)random(5)-2);
```

Then, if the resulting speed is greater than a specified maximum the speed is set to the maximum; the same but opposite happens if the speed becomes less than a specified minimum. The horizontal speed is not allowed to

become 0, being set to -1 or 1 at random if so. This prevents the puck from merely bouncing up and down the rink.

Collisions

There are only two kinds of collision that can happen in this game, and both can be determined using simple geometry rather than needing more complex collision detection methods. The puck can collide with the boards, which is the boundary of the rink, or the puck can collide with a paddle. Collisions with the boards involve a simple range check of the *x* and *y* values of the puck. If the puck is sufficiently near the boundary then a collision will be assumed to happen. As we saw, a collision with the boards results in a sound playing and a change in the *x* or *y* direction. The collision with a paddle is more complex.

The puck is moving faster than 1 pixel per frame in most cases. That means that collision detection can't be done by asking if the coordinates of the puck agree with coordinates of the paddle. The puck could go right through the paddle in a single frame, never having coordinates that agreed. What has to be done is to see if the puck is on one side of the paddle in one frame and is on the other side in the next frame. If so, then it had to have struck the paddle at some point in time between the two frames.

We don't need to be too precise here; we just want the bounce to occur in a visually correct fashion. So, remember that the puck is a circle 10 pixels across, and a paddle is a rectangle 10 pixels wide (*x*) and 30 pixels high (*y*). Figure 4.6 shows a schematic diagram of the situation and gives some simple code to check for the collision. As an example, the basic code for detecting a left paddle collision is:

```
if ((puckX<=lpaddlex+10 && ox>=lpaddlex+10) &&
    (puckY>=(lpaddley-4) && puckY<(lpaddley+34)) )
```

The offset of 4 pixels in the *y* direction allows for the ball to hit the paddle near its corner, as the ball coordinate is that of its center.

Right Paddle

At first it would seem that creating an automatic controller for the right paddle would be easy—simply move it so that the center of the paddle lines up vertically with the puck, *pucky* equals *rpaddleY*+15. This works too well—the right paddle will never miss, and that makes for a boring game. The human player wants a chance to score a point once in a while, and

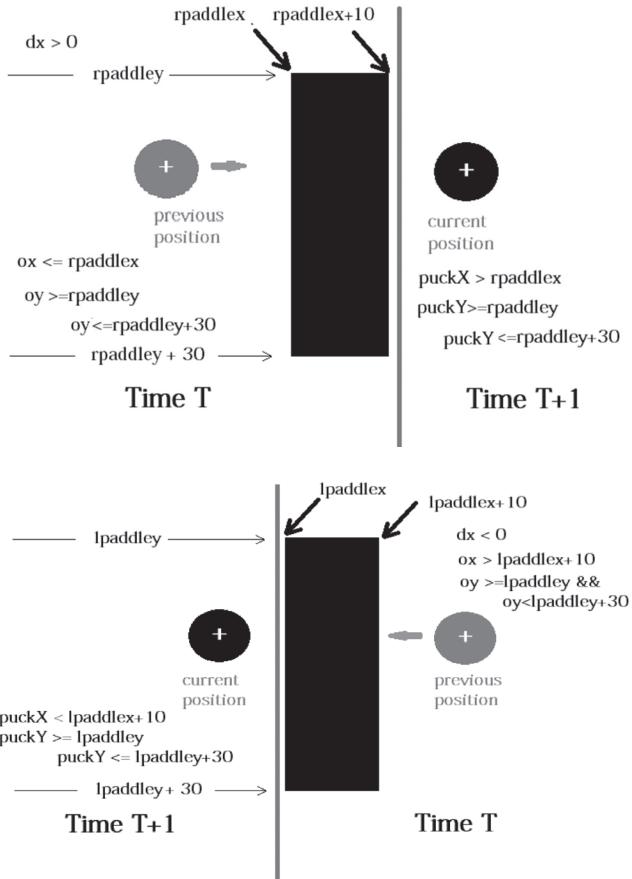


FIGURE 4.6 The top shows the collision with the RIGHT paddle, bottom is with the LEFT.

maybe even win a game. This is a part of game *balance*: to arrange play so that players all have a fair chance.

The first attempt at having the AI player—also called the Non-Player Character (NPC)—fail once in a while was to make the paddle too slow, but that's obvious and frustrating for the human player if they feel the NPC is throwing the game. So the next idea was to have the NPC stay still unless the puck is moving towards the right and has passed a specified X coordinate, initially 420. This looks better, but the NPC still always wins. Setting the motion point at X=600 seems realistic, but playtesting will tell. The relevant code is quite short:

```
if (players == 0 && dx>0) // NPC on and puck moving right
```

```

{
    if (puckX > 600)           // Movement threshold distance
    {
        if ((rpaddleY+15)-puckY > 8)      // Move paddle up
            rpaddleY = -paddleSpeed;
        else if ((rpaddleY+15)-puckY < 8)
            rpaddleY = paddleSpeed;    // Move paddle down
        else rpaddleY = 0;
    }
    rpaddleY += rpaddleY;          // Move the paddle
    if (rpaddleY>pmaxy || rpaddleY<pminy)    // limits?
        rpaddleY -= rpaddleY;          // Move it back
    rect (rpaddleX, rpaddleY, 10, 30);    // Draw the paddle
}

```

Other Minor Features

Some details were implemented after it was found that odd behaviors were observed in the game. Minimum and maximum speeds were implemented so that the pace of the game was maintained in situations where the puck is near one end of the rink behind the goal. It is then given a boost in speed in the x direction so that it is guaranteed to escape relatively soon. And, of course, we check every frame to ensure that the puck is still in the visible region of the window.

Prototype 3

The game has been playable since the code for the paddles was completed, but some of the key features of the play are still missing. This includes face-offs, the three periods, and of course keeping score. A common requirement of these features is a timer for negotiating transitions between parts of the game. We also need a set of states that identify the distinct parts of the game and allow us to specify the circumstances of the transitions.

For example, when the **Play** button on the start page is selected, the game play screen will be displayed. We'll be at the beginning of the first period, the score will be 0-0, and a face-off will be about to take place. This situation is summarized within the game as being in *game state 0*, and a function that implements this state can be implemented that encapsulates the actions needed and the transitions to other states that are possible.

There are 6 game states in *Hockey Pong*, as illustrated in Figure 4.7. They are named:

Game state 0

Begin the game

Game state 1	Begin a period
Game state 2	Face-off
Game state 3	Play
Game state 4	End period
Game state 5	End game

Each state has special requirements, initializes specific variables, and allows the game to progress in a specific direction. This is not a finite-state machine in the traditional sense, because state transitions include and depend on other state variables. For example, the face-offs at the beginning of each period do not have distinct states, but are all in state 1 with a distinct period number. At the heart of these states is a timer.

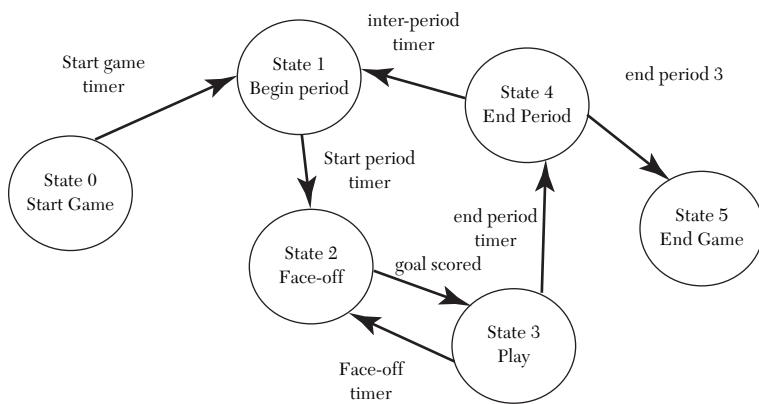


FIGURE 4.7 Game states for *Hockey Pong*.

A Timer Class

The idea of a timer is to permit specific time periods to pass between events; a hockey period is 2 minutes in the current game, and a face-off might need 10 seconds or so. Given this, the basic functions of a timer would be to start and to tell us how much time has passed.

There are two ways to keep track of time using *Processing*. One would be to use the language defined clock function `millis()`. This would give timers that functioned in real time, actual milliseconds passing and being counted. The function `millis` returns the number of milliseconds that have passed since the program started to run. Determining the time between two events is a matter of saving the time of the first event and subtracting it from the time of the second:

```
saved = millis();
```

```
// stuff happens here
duration = millis() - saved;
```

This works pretty well, is efficient, and can be used for other things too, like timing how long specific code takes to execute.

The other way to build a timer is to use the number of frames between two events. Since a game uses frames as the implicit time count this makes sense too. It is accomplished by counting frames between two events, so the timer has to do all of the work in this case. Each time `draw` is called the timer has to be updated; that is, a count of frames is incremented. If we want to know when 30 seconds has passed and the frame rate is 30 per second, then we simply count to 900. In general, $x = n/r$ where x is the number of seconds desired, n is the count of frames and r is the frame rate, which can vary in *Processing*.

This game needs a handful of timers and larger games may need more of them, so a timer class should have the ability to maintain multiple timers. Also, some should run forward and some backward, and perhaps timers should be reset rather than deleted and recreated. Timers therefore need the following interface:

<code>start (i, t)</code>	Start timer <i>i</i> counting <i>t</i> seconds. If <i>t</i> is negative it is a Countdown timer.
<code>time (i)</code>	Time elapsed (or remaining for a countdown) on timer <i>i</i> .
<code>finished (i)</code>	TRUE is the timer has counted to its limit.
<code>tick()</code>	Advance time by one frame.

Game State 0: Start the Game

The game play should not begin immediately after the **Play** button is selected. A player needs a few seconds to orient to the visual situation and prepare for their first actions in the game. The game software also needs a few milliseconds to initialize things like the score and the game clock and start the sounds.

So: after the **Play** button is pressed, a timer (`START_TIMER`, value 0) is started with a value of 5 seconds. Standard audience sound effects are played while initializations are done. After the 5 seconds a transition is made to state 1.

Game State 1: Start a Period

Every period begins with a whistle and audience sounds. This is a 5 second timer (`START_PERIOD_TIMER`, value 1). This gives a delay between periods and allows the period clock to be started. After the 5 seconds a transition to a face-off occurs and the period clock starts (`PERIOD_TIMER`, value 2, countdown).

Game State 2: Face-Off

The game gets quiet. At a random interval between 3-5 seconds (`FACEOFF_TIMER`, value 3) the puck is dropped. When it hits the ice the green light comes on at the ends of the rink and on the scoreboard (see below), the face-off sound effect occurs, and play can begin – the first player to type a movement key wins the face-off, and the puck moves towards their opponent's goal. The game proceeds as one would expect until a goal is scored or the period ends.

After the faceoff timer expires a special flag is set, and the players can type a movement key. The first player to type as indicated by the flag still being set when the key is pressed wins the faceoff; this clears the flag, and the puck moves towards the opponent's goal. The NPC can win a faceoff too, by a random number selection each frame averaging a time of 0.3 seconds.

Game State 3: Play

A transition into state 3 occurs upon any goal, and a face-off occurs. A transition to state 4 occurs when the period end (`PERIOD_TIMER`). The time remaining in the period is displayed at all times, as is the score. This is the only state where the puck moves, and the only state within which a goal can be scored.

Game State 4: End Period

This is a caretaker state that exists to handle the end of the game. From this state if the current period is 3 then the game is over, and we move to state 5. Otherwise, move to state 1 after incrementing the period count. A horn is sounded at the end of the period, and the clock stops.

Game State 5: End Game

After the game is over, the player can relax for a few seconds and appreciate their victory or examine their loss. Any mouse click in this state

causes a transition back to the start screen, where the player can choose to play again or quit. In any case, after 10 seconds (`END_TIMER`, value 4) the start screen transition is made.

The Scoreboard

Every hockey rink has a scoreboard upon which is displayed the score, the time remaining in the period, the penalties incurred and time remaining in them, and other such information as the audience needs to know. In *Hockey Pong*, the scoreboard will occupy the bottom third of the window and will be redrawn every frame. Two fonts are required: one for labeling the fields and one for the numeric values themselves. The time and score are displayed in the font *Quartz-Regular* because it has the appearance of a 7-segment electronic display. The labels are displayed using the *ArialMT* font, a clean non-serif font that is easy to read. Both fonts are available within *Processing* and are simply saved as files and used in the program using the method described in Chapter 2.

Displaying the score and period is simple. The time (integer) has to be broken down into minutes, tens of seconds, and seconds so that it is displayed correctly. For example, a time of 1 minute and 9 seconds would be displayed as **1:9** unless we intentionally display the **0** before the **9**. This is done as follows, where `T` is the timer and `myFrameRate` is the frame rate set by the program when the game begins:

```
textFont (clockFont, 70);
sec = (int) (T.time(PERIOD_TIMER) / myFrameRate + 0.5);
min = sec/60;
sec = sec%60;
sec0 = sec%10;
sec = sec/10;
text (""+min+":"+sec+"."+sec0, 320, 530);
```

The actual frame rate varies as the game is played, and using it here creates a flicker in the digit display.

The scoreboard also has a red/green light on it, indicating that play is live and a goal can be scored (green) or that play is dead and a faceoff is about to happen (red). Each time a goal is scored, the light turns red, and it turns green when the puck hits the ice at a faceoff. The same light appears at each end of the rink to ensure that it will be seen.

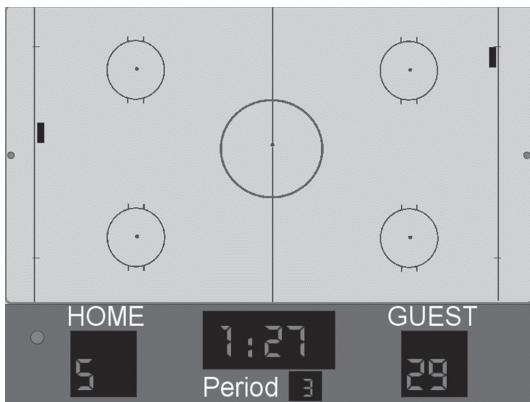


FIGURE 4.8 The final game screen with scoreboard.

Testing

When testing any game—as with testing many human built objects—there are two aspects to be considered. The first is, *does this object meet the criteria for being functional?* The second can be expressed as, *is this object a good example of its type?* Or, *is it art?* The former kind of testing for a game is often just called *game testing*, but is largely about testing the software that implements the game. The latter is called *playtesting*, and is intended to answer the question, *is it fun?*

Software testing is a time consuming and repetitive process. The program is executed again and again in an attempt to execute every line of code and make certain that it executes as designed. Code is tested against the design document, and against a set of standards. The design document answers questions about what the game should do at any particular point. The standards indicate correctness criteria—does this code do the correct thing when a file is not found, or when converting a real to an integer? Does it divide by zero? Are there any off by one errors in the loops? These are more technical questions, and ones that arise in any software.

Game testing is performed by the quality assurance (QA) department in some development companies, and sometimes by people outside of the company hired as testers. The tester must play through the game quite analytically, and when a problem is found, the nature of that problem must be carefully defined. The precise circumstances of the error have to be found by trial and error and given to the developer for correction. Then, after being fixed, the tester must make certain that the error is gone, and that the process of correcting it did not create any new problems.

Playtesting is a different process. It is like other kinds of product testing, where people are hired to try the product and are then asked questions. In game testing, players are recruited to play the game. They are asked questions before play to characterize their demographic identity, they play the game, and are later asked questions about it. Players are often recorded on video while playing to identify emotional reactions. All of the data collected goes to answer questions about whether the game is enjoyable, where it is fun, and where it is not. Iterations of this testing can be done, making design changes between each test, involving 6-12 people each time.

Beta testing is the final stage. Unlike beta testing for software systems, a game continues to have the code testing as something of a distinct process from that of the game play. The beta test involves a release of the game to the public, or a subset of the public, so many people can play it. They will find problems in the code if any remain, but the key element is play. Many people are playing, and report on their experience. Final tweaking can be done before the ultimate release of the game, which we hope will be the best we can do.

Hockey Pong Game Design Document



This is an abridged development document. Material will not be duplicated in detail between this and the material in this chapter. See the Game Design Document (GDD) on the CD for a more complete example.

Copyright Information © 2013 J. Parker

Version history

Time	Author	Changes
14:23 2013-08-22	J. Parker	Create initial document
9:30 2013-08-28	J. Parker	Rev 1 Prototype 0 and 1 Art detail Screens and transitions
10:15 2013-09-02	J. Parker	Rev 2. Prototype 2 Puck detail Paddle detail
9:45 2013-09-08	J. Parker	Rev 3 Prototype 3 Game FSA detail Score Faceoff Timer

Table of Contents

** For an example table of contents, refer to the GDD available on the companion disc.**



Game Overview

Game Concept

Hockey Pong is a 2D two player arcade type game in the style of the original Pong. The play elements of the game are quite similar to Pong: an animated ball (puck) moves left to right across the screen, bouncing off the sides of the playing area. If this ball passes through a goal area on the left or right end of the screen then a point is scored, and play starts again with a face-off.

Unlike the standard Pong, there will be three 2-minute long periods of play. The play ceases at the end of each period for a few seconds, followed by a face-off. Can you score more points than your opponent in three 2-minute periods?

Feature Set

- Like real hockey, a timed game rather than first past the post.
- Select your team and their logo appears on the ice.
- Two players or one player against the computer.
- Sound effects and (perhaps) music.
- Playable on a PC, Mac, Linux, or online through a web page.
- Variable speeds. The puck changes speed to simulate tempo changes in real hockey.
- Replayable. As the player gets better, their score improves. The AI player has adjustable levels.
- Face off after goals and at the start of each period.

Genre

Arcade, two players or one player against the computer.

Target Audience

Younger players (under 15) and casual gamers. Hockey or Pong fans of all ages. This has potential as an advergame, and could be played online through a browser or on a mobile device.

Game Flow

After an initial splash screen, the game begins with a traditional hockey face-off: the referee (invisible) will blow a whistle and drop the puck at center ice. The player who responds to that most quickly takes possession of the puck, and it moves towards the opponent's end. The players control a goalie using the keyboard, and can move him up and down the screen. The idea is to have the goalie block the puck from entering your own goal, at which point it bounces and moves towards the opposing goal. The speed of the puck varies with each collision, and the bounce direction is slightly random too. After two minutes the play stops, and a new period starts with a face-off. Goals result in a face-off as well.

The winner is the player with the most points after 3 periods of play. Players can select from a few parameters on the initial splash screen, including the home team logo to be placed on the ice (playing surface).

Look and Feel

The game is played on a flat surface made to look like a hockey rink. Key presses are used for control and sounds are used to convey the ambience of a hockey arena.

Project Scope

Locations

Just one.

Levels

Just one, but three periods.

Non-player Characters

No actual non-player characters, but the player may play against the computer, which will then take over the controls on the right side of the game.

Gameplay and Mechanics

The basic mechanic of both hockey and Pong is to get the ball/puck past a straight line in space that is being guarded by your opponent. In this game the player on the left attempts to get the puck past a vertical line on the right side of the screen, and the player on the right tries to get it past a line on the left of the screen.

Gameplay

There is a puck and two paddles, one paddle on each side of the screen. Each player can move a paddle, the left player by using the 'w' and 's' keys, and the right player by using the up and down arrow keys. The puck moves mainly left and right, but can bounce off of the sides of the 'rink' and will certainly bounce off the paddles or goalies.

Objectives

The player scores a point if the puck passes over their opponent's goal line. The player with the most points after six minutes of play, that is 3 two minute periods, wins the game.

Mechanics

The W and S keys control a paddle on the left side of the screen. The arrow keys control a similar paddle on the right. Each player tries to avoid having the puck pass their goal area by moving the paddle so that the puck bounces off of it. The puck will bounce back and forth until one player misses. The player who let the puck pass their scoring line does not get a point, the other player does.

There is no maximum score.

A goal cannot be scored after time has expired in any given period. A horn will sound to indicate that time is up.

Each goal has lights to indicate whether a goal has been scored. A green light indicates *yes*, red indicates *no*. The red light goes on after the time for any period is over.

What are the rules to the game, both implicit and explicit. This is the model of the universe that the game works under. Think of it as a simulation of a world, how do all the pieces interact? This actually can be a very large section.

Physics – How Does The Physical Universe Work?

This universe is 2D with no friction. The puck moves at various speeds but does not cease motion, as a real puck would, if it is not touched. There are no hockey players on the rink as there would be in real situations, so the bounces and speeds are somewhat random to account for this lack. The play is made less predictable by the existence of real players, so this game uses random speed and bounces to introduce that aspect to the play.

Movement

General Movement

The puck can move left and right, and to a lesser degree up and down, and does not stop moving unless a goal is scored or a period ends.

The puck starts moving after a face-off, in the general direction of the goal of the player who lost the face-off.

The puck will bounce off of the rink side and ends (the ‘boards’) in a somewhat predictable manner, but with small degrees of randomness introduced.

The paddles move only up and down, under user control.

Other Movement

Unlike the original *Pong*, the entire end of the rink is not the goal area. A goal is scored only if the puck passes one of the end red lines between a pair of ‘goal posts’ that represent the net of a hockey goal. If the puck hits the end boards it bounces, and if it hits the back of the goal it bounces.

The puck also bounces off the paddles, which represent the goalie.

Objects

Picking up Objects

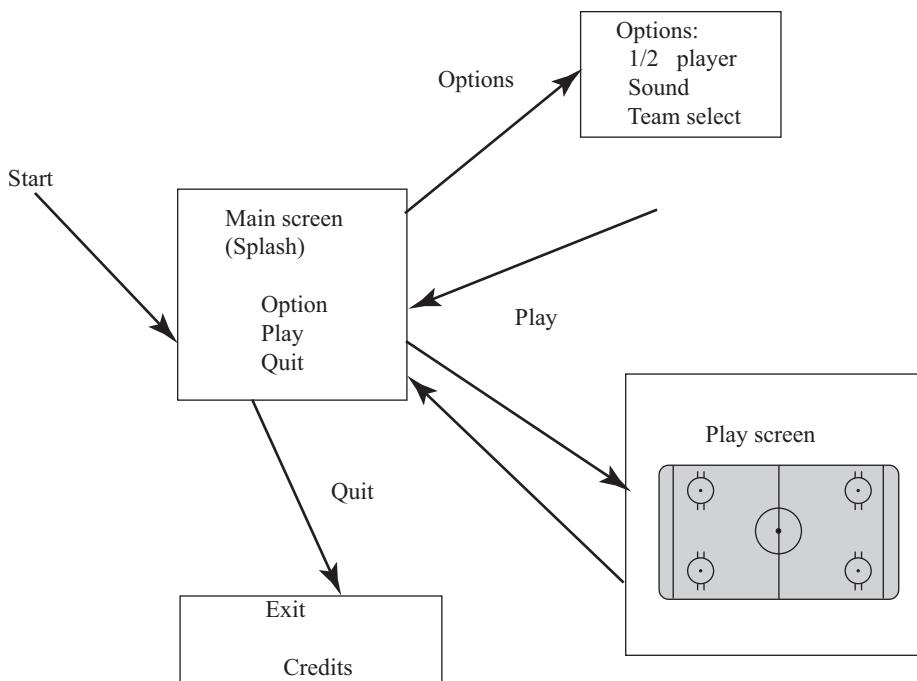
No objects in this game can be picked up.

Moving Objects

The objects that move are the paddles and the puck.

Screen Flow

Screen Flow Chart: A Graphical Description of How Each Screen Is Related To Every Other



Screen Descriptions

Main Menu Screen

The main menu screen is displayed when the game starts. It allows the player to navigate to the other screens, especially the play screen. All screens use the font *Kozuka Mincho Pro EL*, a standard font in Paint.

The main menu has three buttons for transitions; each button is an image, and has an armed and unarmed form. They are:

Button function	Unarmed	Armed	Coordinates
Transition to Options screen	options.gif	optionsA.gif	56, 400
Transition to Play screen	play.gif	playA.gif	341, 400
Transition to Quit screen	quit.gif	quitA.gif	630, 400

Options Screen

Permits the player to adjust the audio, select a team, and choose 1 or 2 players. This screen has two buttons for option specification and four logo

images that can be selected like radio buttons. Each button is an image, and has an armed and unarmed form. They are:

Button function	Unarmed	Armed	Coordinates
Select 2 player	single.gif	singleA.gif	400
Select 1 player	double.gif	doubleA.gif	341, 400
Sound On	soundOn.gif	soundOnA.gif	630, 400
Sound Off	soundOff.gif	soundOffA.gif	
Logo 1	OHC_logo.gif		461, 350
Logo 2	victorias.gif		536, 350
Logo 3	logo3.gif		611, 350
Logo 4	logo4.gif		686, 350

Play Screen

The game screen itself, showing the play surface and controls.

Exit Screen

Allows credits, advertising if an advergame, and could display a high score table.

Game Options

Option 1: Two Player/One Player

Obviously, if the player selects a one player game then the game itself has to play the other paddle. This requires a bit of intelligence, and more code, but insofar as game play is concerned the main issue is face-off management. Clearly the computer can always win the face off, and if it does the player will suspect it is cheating. Thus, in the one player game the winner of a face-off is determined by a coin toss (random number < 0.5).

Option 2: Music On/Off

Music here refers to background music, not sound effects. If music is present, it should be under user control. No consequences for game play.

Option 3: Home Team Selection

The player may select from a set of teams to be theirs, and the logo for that team will appear in the arena. In a real commercial game, a license would be required for the display of a team's logo within the game. There is no impact on play here.

Replaying and Saving

Games cannot be saved. It's only 6 minutes long.

Cheats and Easter Eggs

None.

Story, Setting and Character

** There is no story in this game, but in the games that have stories, the narrative needs to be defined carefully covering the elements listed below. Cut scenes are short pieces of animation that are used as a transition and narrative device. They don't affect play directly, but do convey story elements and information the player needs. They also indicate where the player is in the game, and what future actions might be important. **

- Story and Narrative
 - Back story
 - Plot Elements
 - Game Progression
 - License Considerations
 - Cut Scenes
 - Cut scene #1
 - Actors
 - Description
 - Storyboard
 - Script

Game World

Anyone who has been at a professional sporting event would understand the feel we're going for, but it is a two dimensional world so the space will be presented to the player in sound.

General look and feel of world

The nature of a hockey rink is well understood. The playing surface is ice, generally a blue white color. Pucks are black rubber. The area where

the audience sits is not visible, so the sounds of the audience must be made available.

** In games where there are multiple locations where play takes place, it is important to include the following details for one: **

- Area #1
- Description
- Characteristics
- Which levels use this area
- Transitions to other areas

Characters

** This game has no characters. In games where characters exist, the following should be provided for each character: **

- Back story
- Relevance to story
- Personal Traits
- Look
- Physical characteristics
- Special Abilities
- Relationships (to other characters)
- Characteristics

Levels

** This game has a single level. In multiple level games, the significance and characteristics of each level has to clearly described in detail. There are special designers called level designers who are responsible for individual levels, and who will fill in the details of these sections: **

- Level #1
 - Summary
 - Introductory/Transition Material for Player

- Objectives
- Physical Description
- Map
- Encounters/Tasks/Objects
- Walkthrough

Interface

** This game has a simple interface. In more complex games, there will be sections at this place in this document for special rendering systems, lighting, camera models, and special items like a heads-up display (HUD), mini-map, and other features. **

Control System

Controls, as with all arcade games, are simple. The motion keys are used for play. The mouse is only used (in the PC/Web versions) to select screens.

Audio

Sound is not necessarily high quality, but is an essential part of this game. We'll use the *simpleAudio* system (developed in Chapter 3) for sound display, and don't really need positional audio. There will be an ambient area/audience sound played at a low level while play is ongoing.

Music

Music, if present, should be limited to the main and exit screens, and will be a typical athletic anthem.

Sound Effects

This style of game tends to have many sound effects. This game in particular needs the following sounds:

- Puck bouncing off of the boards
- End of period horn
- Puck hitting a stick
- Puck hitting the goal post
- Referee whistle

- Puck hitting the ice/face-off
- Audience, ambient
- Audience, approval
- Audience, angry
- Audience, goal cheer
- Audience chant
- Horn in audience
- Organ, between face-offs
- Body check

Help System

None.

Artificial Intelligence

Opponent AI and Collision Detection

Technical

Target Hardware

Development Hardware and Software

Development Procedures and Standards

Game Engine

Network

Scripting Language

Game Art

Concept Art

Style Guides

Characters

Environments

Props (Tools, Equipment)**Cut scenes****Miscellaneous****Development Software****Editor, Installer, etc.****Management****Schedule (From start of project)**

High Concept	2 hours
Initial prototype	1 day
Producer meeting	2 hours
Screen design	1 day
Screen Art	1 day
Buttons	3 hours
Sound (level 0)	2 days
Game screen	3 hours
User control	1 day
AI control	4 hours
Game states	2 days
Sound (level 1)	1 day
Playtesting	3 days
Total	13 days 14 hours = 15 days
Response	2 days

Budget

Design	\$2,000
Art	\$ 800
Programming	\$2,000
Sound	\$ 900

Testing	\$1,200

Total	\$6,900

Localization, Risk, Special Considerations

IP: no current logos are used, no images from outside sources, and no music or sounds that have a copyright.

It is possible to include logos from any team, including professional leagues, national teams, or local pee-wee and youth leagues. This could yield a market for this as a promotional game with regional/local content.

Sound could be added that involved a short script (start screen and end screen), and local player names could easily be added to audience shouts.

Test Plan

First level testing will use the usual internal sources. Before the release, the game will be given to young people (ages 12-16) in an online forum using recorded audio.

Appendices

Asset List

Art

Model and Texture List

All art files in **2013/hockey/Art**

No models.

Ice texture ice.png

Rink rink.bmp (play surface)

rinkLarge.bmp (original art)

Animation List

None

Effects List

Interface Art List

Start screen screen0.png

Play button play.gif (unarmed) playA.gif (armed)

Options button	options.gif ((unarmed) optionsA.gif (armed)
Quit button	quit.gif (unarmed) quitA.gif (armed)
Option screen	screen1.png
Player button	double.gif (unarmed) doubleA.gif (unarmed)
	single.gif (unarmed) singleA.gif (armed)
Sound button	soundon.gif (unarmed) soundOnA.gif (armed)
	soundOff.gif (unarmed) soundOffA.gif (unarmed)
Team selection	OCH_logo.gif, victorias.gif, logo3.gif, logo4.gif
Back	back.gif (unarmed) backA.gif (armed)
Game screen	back.gif (unarmed) backA.gif (armed)
Exit screen	screen3.bmp

Cut Scene List

None

Sound

Environmental Sounds

Audience/arena ambience	amb-aud01.mp3	1:01	amb-aud05.mp3
	amb-aud02.mp3	1:02	amb-aud05.mp3
	amb-aud03.mp3	1:00	amb-aud07.mp3
	amb-aud04.mp3	1:03	1:01
Beginning of game	entry.mp3		

Event Sounds

Puck hitting boards	bang00.mp3 to bang10.mp3
End of period horn	endPeriod.mp3
Referee whistle	sfx-whistle.mp3

Interface Sounds

None

Music

None. (Ambient, Victory, Defeat, ...)

The game as developed has no music to avoid copyright issues.

Voice

** This section would include scripts and stage directions for any voiceover work. There is none here at this time.**

Summary

In this chapter, we completed the design of *Hockey Pong* and developed a working version of the game as four prototypes. We first created a basic demo of game play, then a set of screens and transitions between them were implemented using buttons. Then the game play was developed, sounds were added, and finally the ancillary features surrounding the game were completed.



The game is provided on the disc, and is playable on any PC that has *Processing* running on it.

Exercises

1. Select a Web-based game of your choice and document the following aspects:
 - a. Identify all screens and transitions.
 - b. Characterize all interface actions (key presses and mouse clicks).
 - c. Does the game possess internal states? Identify them.
2. Discuss the pros and cons of using the mouse as an interaction mechanism instead of keys on the keyboard. (Note: As we'll see later, touch screens often map onto mouse actions)

3. This game would be less like *Pong* if the puck could be moved about the rink and shot more as in the real game of hockey. How might this be done? What are the positive and negative consequences for game play?
4. The background image for the **Start** and **Options** screen is intended to look like an ice surface after it has been skated on. This image was created using *Processing*. Write a program to create a similar image, using no imported graphics or image files.
5. There are multiple sound effect files for many of the effects used in the game, and as described the system chooses one at random every time a sound is needed. This could result in the same sound being played many times in a row, defeating the purpose. Devise a scheme that makes it impossible for the same sound to be played twice in a row. Implement that scheme.
6. Create or download a sound effect that represents a typical audience sound, such as clapping, pounding, or a horn blowing. Edit the sound so that it is acceptable in the context of the game (i.e. adjust the pitch or duration, reduce the noise). Then have this sound played at random moments during game play.
7. Currently the game clock displays time as minutes and seconds remaining. It is common in real hockey games to display the time to hundredths of a second during the final seconds of each period. Modify the game clock to accomplish this for the final 10 seconds.
8. Write a short voiceover to begin each period—something like “The second period is about to start.” You can make this recording using any equipment available, such as a VOIP microphone, then add ambiance such as echo. Play these at the beginning of each period.
9. Insert the sound clip of the puck hitting the boards, created in Chapter 3, into the game. Add it to the set already there rather than replacing one of the existing sounds. Test it and make it possible to tell when your effect is being played in order to accomplish that test.

Resources

Sound Effects

The sounds from SoundBible.com that are labeled *public domain* or *creative commons* can be used without fee in games. Attribution should be given. <http://soundbible.com/>

Some of the sound effects for *Hockey Pong* were downloaded from FreeSFX, or were based on those effects. <http://www.freesfx.co.uk>

Sound Editing

Audacity is freely downloadable editing software with a high degree of functionality. If you need to save a file as an MP3, you'll have to download the *LAME MP3* encoder and install it. <http://audacity.sourceforge.net/>

Goldwave is a freely downloadable sound editor with a large set of audio formats in which sounds can be saved. <http://www.goldwave.com/>

LAME is MP3 encoding software. <http://lame.sourceforge.net/>

Graphics Editing

Paint Comes with Windows, and is a highly underestimated tool for putting together 2D images.

LView An image editor that is a valuable addition to Paint. It is especially useful for making backgrounds in GIF images transparent. Free download, but you should send them money if you like it. <http://www.lview.com/>

CHAPTER 5

GRAPHICS IN THREE DIMENSIONS

In This Chapter

- How 3D graphics work in a 3D environment
- Creating 3D basic 3D graphics

Lots of people think that computer games are all about graphics. Some games might be, but graphics are not what makes a game play well. Naturally, the use of some graphics is essential. You have to see to guide your avatar, and a rendering of the scenery or indoor volume and opponents is essential to play any fully 3D game. The degree to which a game depends on its graphics is one way to classify them, and it is quite obvious that there is a wide variety.

In any book about games, the concepts underlying 3D graphics has to be discussed someplace. As a practical exercise, we will write a simple 3D game, and in this chapter we will learn enough about practical graphics to begin the implementation of the game. It will be assumed that you knew *something* about graphics at one time, and references appear at the end of this chapter so that you can study on your own. In addition we've already discussed some of the 2D graphics operations in Chapter 2.

A modern desktop computer has a wide variety of multimedia devices attached that a game needs to use—the graphics card and the audio card are the key ones. Each specific brand of device works a little differently from the others. The designers of *Processing* have tried to isolate the users from machine dependencies as far as is possible, and so a typical programmer needs to know nothing much about the multimedia system, *Microsoft DirectX* or *OpenGL*, a more portable graphics system created by the now defunct *Silicon Graphics*. *OpenGL* has its difficult parts, but has a simpler interface than *DirectX*. It is the basis for the paradigm that *Processing* uses for much of their 3D implementation.

As we learned in Chapter 2, a typical computer screen consists of hundreds of thousands of dots in a rectangular array. Each dot (picture elements, or *pixels*) can be made to light up in color specified as RGB coordinates and is used as a component of a bigger picture. Computer graphics is really all about connecting pixels into geometric shapes that look like what we are trying to draw.

A goal of most graphics systems is to distance the artist or programmer from the idea of pixels. Most people don't think of images in terms of their pixels, and pictures are not generally drawn that way. Pixels are never how we really think of pictures, and is never how we would draw them with a pencil or a brush. The *OpenGL* and *DirectX* systems have very few operations for drawing pixels specifically—when we draw a polygon, we specify the coordinates of the vertices, a color, and maybe even a texture to be mapped onto it, and *OpenGL* converts that into a set of low level pixel operations. We do not manipulate pixels directly. High-level systems like game engines convert our rendering requests and lighting setup—via a set of function calls—into a raster image. The mechanism is hidden from us, for the most part.

We want to think of the scene as consisting of high-level objects that we move about, not as pixels. The basic components of all images we will create are polygons.

Using Polygons to Build Objects

Any object we wish to draw can be created from simple polygons, usually triangles. Figure 5.1 shows a sphere constructed out of triangles. From left to right, the number of triangles used increases, and the representation looks more and more like a sphere. There is a limit, of course, but gener-

ally the more polygons used to represent an object, the better it looks. The act of building an object from triangles is called *tessellation*, or sometimes *polygonization*, and the polygonalized object is sometimes referred to as a *mesh*.

Most games use polygons to represent objects because it is easy and fast. Algorithms for determining whether something can be seen (visibility) are easy when using polygons. Polygon intersections can be calculated simply and quickly, as can whether a point is in or outside of a polygon. Polygons can have shading applied so that a collection of them appears to be a smooth surface, and they can have textures applied very easily. Simplicity and speed are common reasons for doing things in a computer game.

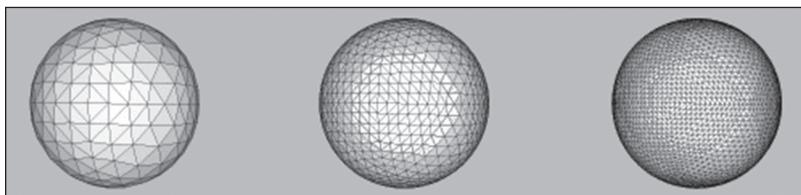


FIGURE 5.1 A sphere polygonalized at various resolutions.

Meshes can be created in a few distinct ways. Easiest of all, we can use a drawing tool to create the object and simply save it as a mesh. Most tools used for building graphical models, like *Maya* and *3D Studio Max*, will save objects as a polygonal mesh. The other main way to create a mesh is to define a surface somehow, perhaps by defining it mathematically or perhaps by digitizing a real object, and then running the data through a special polygonalization algorithm. Both methods are complex in their own way, and so we will simply assume from now on that any objects to be drawn into our game exist as a polygon mesh—however it was created—or can be drawn using *Processing* primitives. One can get models of many objects from websites that offer free 3D models. A list of some of these sites can be found at the end of the chapter.

Storing and Drawing Polygons

Let's consider storing and drawing triangles as a special case. This is what we'll probably be doing anyhow, in practice. A triangle has three sides, or *edges* and three points where edges meet, also called *vertices*. In three dimensions each vertex has three coordinates: X, Y, and Z. This would usually be stored as a three dimensional vector, such as:

```
float vec3d[3];
```

A triangle will need three of these, so you could save a triangle as:

```
float triangle[3][3];
```

So, `triangle[1][2]` represents the second vertex (the first has index 0) and the Z coordinate of that vertex (The X coordinate would be 0, the Y would be 1).

Of course, any object will consist of a collection of triangles. We won't always know how many triangles in advance, and not all objects will have the same number. Let's create a structure:

```
class simpleObject
{
    int Ntriangles;
    float vertices[][] = new float[3][3];
    simpleObject (int N)
    {
        Ntriangles = N;
        vertices = new float[N][3];
    }
}
```

Each object has its own instance of this class, which is just big enough to hold its polygons. There can be as many objects as needed, and these can also be stored in an array.

Drawing them is easy, especially because we are not going to write the whole program. We have a procedure that draws two-dimensional lines—if we had a 3D version we could draw a triangle stored in the array `T` by simply doing the following:

```
line (T[0][0],T[0][1],T[0][2], T[1][0],T[1][1],T[1][2]);
line (T[1][0],T[1][1],T[1][2], T[2][0],T[2][1],T[2][2]);
line (T[2][0],T[2][1],T[2][2], T[0][0],T[0][1],T[0][2]);
```

Now, *Processing* has a 3D `line` function, and this will actually work. So all that we need to understand is how to represent our universe (the things we wish to draw) as polygons, and everything else falls into place. The 3D line function does have some prerequisites, though. We need to tell *Processing* that we want to draw in 3D, because the default is 2D. That's easy to do but has consequences.

In 3D, we need to have an appreciation of other aspects of the scene.

Lighting, point of view, scale, and perspective all affect the way things will look. Fortunately, it is not necessary to do all of these things ourselves. They have been implemented by the graphics system—all we need is an understanding that these aspects exist, an appreciation of how these things affect the graphical version of the scene, and a knowledge of how the graphics system implements them.

Introduction to *OpenGL* Graphics in *Processing*

The *OpenGL* system consists of many parts, including a window system and performance tools, but from the perspective of a *Processing* programmer, there is only its graphics system. It has facilities for color manipulation, texture mapping, geometry (rotation and translation), and rendering polygons—the basic operations needed to draw cars and buildings. While *OpenGL* is a pretty complete basic graphics system, it does not draw complex objects—they must be built out of simpler ones, like polygons, and drawn them using the basic operators it provides.

In order to use *OpenGL* within *Processing*, you must indicate this to the system in the `setup()` function. In the call to `size`, simply add the word `OPENGL` as the third parameter:

```
size (640, 480, OPENGL);
```

Without the third parameter, *Processing* assumes you want to do 2D graphics. Also, because *OpenGL* is a library external to *Processing*, you must import it, as we did with *Minim*. At the top of the *Processing* code put:

```
import processing.opengl.*
```

We can now use the *OpenGL* library.

Triangles

Let's get back to drawing 3D triangles. Triangles are the simplest kind of polygon, and games often draw a lot of them. Being able to draw them is important. We now know how to declare that we're using 3D and we know that the `line` function can take 3D coordinates. A complete *Processing* program to draw a 3D triangle is given on the right. It follows the description above in every respect, and is a complete program that compiles and executes.

This triangle is 3D, although the third coordinate is always 0. All triangles define a plane, in that any three points have a unique plane passing

through them—this is not true of four points, as we'll see. The points that define the triangle, or *vertices*, here are (100,100,0), (100, 200, 0), and (200, 200, 0).

```
// 3D triangles
int T[][] = new int[3][3];

void setup()
{
    size(300, 300, OPENGL);
    stroke(0); fill (255,0,0);
    T[0][0] = 100;  T[0][1] = 100;  T[0][2] = 0;
    T[1][0] = 100;  T[1][1] = 200;  T[1][2] = 0;
    T[2][0] = 200;  T[2][1] = 200;  T[2][2] = 0;
}

void draw()
{
    line (T[0][0],T[0][1],T[0][2], T[1][0],T[1][1],T[1][2]);
    line (T[1][0],T[1][1],T[1][2], T[2][0],T[2][1],T[2][2]);
    line (T[2][0],T[2][1],T[2][2], T[0][0],T[0][1],T[0][2]);
}
```

If this were the only way to draw a triangle, then complex objects would be tedious to define and draw. *OpenGL* allows us to define a more general *shape* that consists of a set of vertices in 3D connected by lines. Shapes nearly always consist of more than one polygon, but we're just starting out, and will draw the same triangle as before using a *shape*. What we need to do is to start the shape by calling a function named `beginShape` and passing one parameter—the kind of thing the shape will consist of, in this case `triangles`. This is followed by a set of calls to a function named `vertex`, each specifying one point (3 coordinates) of, in this case, a triangle. Since a triangle has three vertices there must be three calls to `vertex`; in general, for objects consisting of triangles, there will be a multiple of three calls to `vertex`.

```
// 3D triangles
int T[][] = new int[3][3];

void setup()
{
    size(300, 300, OPENGL);
```

```

stroke(0);
T[0][0] = 100; T[0][1] = 100; T[0][2] = 0;
T[1][0] = 100; T[1][1] = 200; T[1][2] = 0;
T[2][0] = 200; T[2][1] = 200; T[2][2] = 0;
}

void draw()
{
beginShape(TRIANGLE);
vertex (T[0][0],T[0][1],T[0][2], T[1][0],T[1][1]);
vertex (T[1][0],T[1][1],T[1][2], T[2][0],T[2][1]);
vertex (T[2][0],T[2][1],T[2][2], T[0][0],T[0][1]);
endShape(CLOSE);
}

```

The code above shows this. The initialization of the triangle coordinates is the same, but the `draw` function now uses a *shape block*, which is the code between the calls to `beginShape` and `endShape`. Note that the call to the `beginShape` function passes `TRIANGLE`, which is a constant that tells the system that the vertices that follow are to be formed into triangles. There are other choices, as we shall see. The function `endShape` is passed a constant `CLOSE`, which means that the system is to complete the polygon by connecting the last vertex to the first. This is a good idea if the polygon is generated by code or consists of real numbers, since rounding errors can create gaps in polygons otherwise.

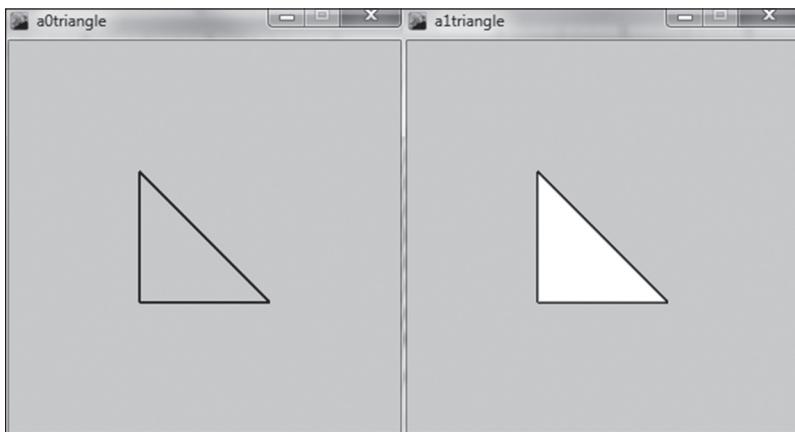


FIGURE 5.2 (Left) A triangle drawn by three calls to the `line` function. (Right) A triangle drawn using `shapeBegin`-`shapeEnd`, which is known by the system to be a polygon, and can hence be filled with a color (in this case, white).

Figure 5.2 shows the triangles drawn by these two small programs. This new way to draw a triangle, or in general a polygon, may seem just as time consuming as the use of the `line` function, but in fact allows greater functionality. Specifically, each group of vertices is grouped into specific polygons or sets of polygons that can represent objects. This fact can be inferred from the figure: the triangle on the left consists of three lines, and the system does not understand it to be a polygon. It is not shaded. The triangle on the right is filled with white, and is known by the system to be a polygon. These objects can be manipulated as a unit; they can be rotated, scaled, shaded or given textures. This can't be done if we draw objects as lines.

At this point, there's no way to tell whether this is really a 3D triangle. It's drawn in 2D, as computer images always must be. How can it be proven that these are being drawn in a 3D space? We must be able to see all side of the triangles, to be able to move around them in three dimensions and view them. To do that we have to create a *viewing transformation*—we need to establish a point in space from which we are viewing the triangles, a set of transformations that define how the coordinates of the vertices are projected onto a 2D thing like a screen on the retina of the eye, and naturally the scene has to be illuminated somehow. These are all pretty complicated ideas, and result in some complex math. However, the code will function whether you understand the math or not.

Viewing

A *viewing transformation* takes the objects and the lights that have been defined by the programmer and creates a two-dimensional projection as seen from a particular point in space. This projection can then be displayed on the computer screen as pixels—the goal of any graphics program. The viewing transformation needs to know where all of the polygons are, where the viewer is (in three dimensional space) and what the parameters of the view are—perspective or orthographic, the direction the viewer/camera is looking, the focal length, and so on.

The most commonly used projection for games is the *perspective* projection, because it is the nearest to what we would expect in real life. For example, more distant objects appear smaller than closer ones. The position of the viewer can change, even while the game is being played and the perspective needs to be consistent with the viewpoint. We have to be flexible and allow for multiple viewing positions. Any projection that does not show

distant objects as seeming smaller than close ones would be rejected by our common sense, and would be difficult to incorporate into a 3D game.

Terms like *projection*, *viewing position*, and *perspective* have been tossed about as if they are already understood. In fact the whole concept of taking three dimensional objects in a virtual space and creating a planar representation as would be seen from a virtual position is conceptually very difficult. At least, many people have trouble with it, even engineering students. There are two common ways to cover the needed material—the entire thing could be abstracted and described as geometry and algebra, or it could be described as images where the intention is to create 3D mental models.

Why do this at all? *OpenGL* will do the work for us, so why not just explain how it does it rather than show all of the details? Because experience, again, shows that the result of getting the *OpenGL* calls wrong is often a view of nothing at all. This does not tell you what you did wrong, and trying to get it right by trial and error is a bad idea. Although you may get lucky, there are just too many ways to get nothing drawn on the screen. So here are the basics in a couple of pages, and you can follow up using resources listed in the reference section of this chapter if you wish to know more.

Projecting the Image of a Scene onto a Plane

Imagine that the scene consists of a simple prism and that the center of the front face of the prism is distance d from your eye. This situation is diagrammed in Figure 5.3.

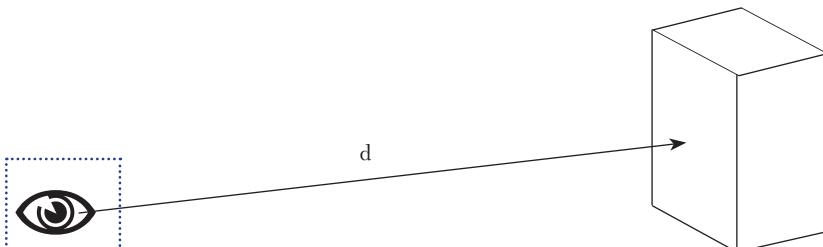


FIGURE 5.3 Geometry of viewing a scene.

Simply put, the projection of the scene, or the prism here, is a two-dimensional view obtained by placing a plane in between the viewer and the scene. Rays drawn from all points on the prism to the eye will intersect the plane, and those places are indicated by a change of color on the plane. This will create an image of the scene, as seen in Figure 5.4.

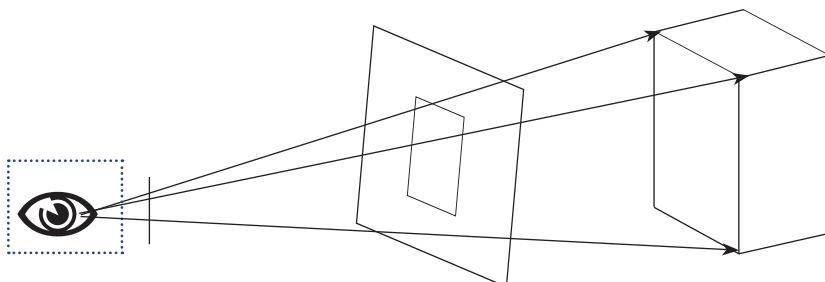


FIGURE 5.4 Scene viewing geometry determines how 3D scenes convert to 2D images.

From the game writer's point of view, all we really want is the projection. The screen is two dimensional, and *is* the plane onto which the scene is projected. We wish the projection to have the basic properties of the real situation. If we move the plane nearer to the prism, the projection of the prism gets larger until it becomes the same size as the prism itself. Moving the plane away from the object reduces the size of the image projected. This is correct, so far.

The plane in question is referred to as the *viewing plane* or *projection plane* (PP), and the location of the eye above is often called the *center of projection* (COP). We need to have a simple numerical description of what's happening so we can calculate the view at any time in our game.

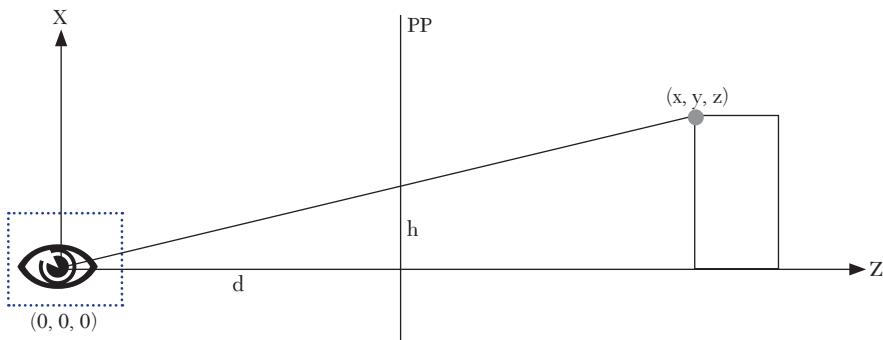


FIGURE 5.5 The projection plane and an eye-centered coordinate system.

We will want to determine what the player sees at each time, and display that for the player in close to real time. In particular, we need to know how big things should appear on the PP when objects are a specific distance from it. If we look at the geometry of the situation it is surprisingly simple. In the illustration in Figure 5.5, we want to be able to compute the height

of the prism in the projection, **h**. Its actual height is **X**, and its distance from the viewpoint is **Z**.

This is what is called a *similar triangle*: the triangle defined by the viewpoint and the projected top and bottom of the prism has the same angles as the one defined by the viewpoint and the actual top and bottom of the prism. This means that the ratios of the height to distance (from 0,0,0) are constant; or in other words:

$$\frac{h}{d} = \frac{x}{z}$$

It is possible to calculate apparent sizes quite simply. We know the coordinates of the view point and the object, and so have **d**, **x**, and **z**.

The discussion so far has concerned itself with the *perspective* transformation. If the viewpoint (COP) is moved away so that **d** is infinite, then the lines from COP to the object become parallel and we get an *orthographic* or an *oblique* projection, depending on other details. There is no decrease of size with distance here (**d** is so large that, in real life, you would not see anything at all), and these are of limited use in games. If you are looking for more details on parallel projections, *Introduction to Computer Graphics* by Foley and van Dam is a good resource.

Perspective Projection in Processing

The perspective projection is a mathematical transformation that is applied to all vertices drawn. It is set up once, at the beginning of rendering, and remains in effect for the entire program. The function that sets this up is `perspective`:

```
perspective(fovy, aspect, zNear, zFar);
```

In this function, `fovy` is the vertical field of view angle (in radians), `aspect` is the aspect ratio, the ration of width to height of the view, and `zNear` and `zFar` are the near and far clipping distances or planes. This is illustrated in Figure 5.6.

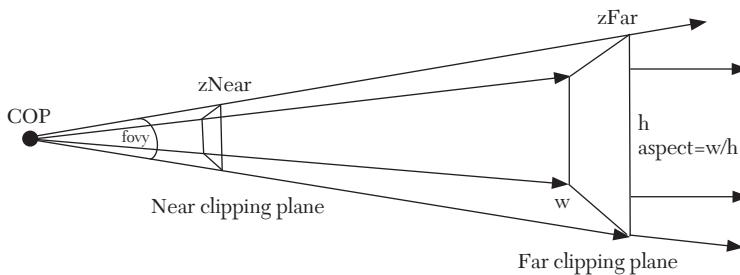


FIGURE 5.6 Perspective projection geometry.

It takes a bit of time experimenting with the parameters to get a real feel for their effect on the image. A call to `perspective` without passing any parameters means that *Processing* will use the default values where `cameraZ = (height/2)/tan(60.0°)`:

```
fovy = π/3
aspect = screen width/screen height
zNear=cameraZ/10
zFar=cameraZ*10
```

This sets up the viewing volume, and establishes the foreshortening that makes distant objects appear smaller than near ones.

```
void draw()
{
    background(255);
    beginShape(TRIANGLE);
    vertex (T[0][0],T[0][1],T[0][2], T[1][0],T[1][1]);
    vertex (T[1][0],T[1][1],T[1][2], T[2][0],T[2][1]);
    vertex (T[2][0],T[2][1],T[2][2], T[0][0],T[0][1]);
    endShape(CLOSE);
    d = d + dd;
    if ((d< -1000) || (d>1000)) dd = -dd;
    T[0][2] = d; T[1][2] = d; T[2][2] = d;
}
```



Applying this to the program that draws the triangle is illustrative. On the disc this program is **a3triangle.pde**, and the code for `draw` is on the right. This program changes the Z value (distance to the triangle) from -1000 to +1000 and back while displaying the triangle. When the program is executed the triangle appears to change size from quite small to filling the screen. This is evidence, for the first time, that the triangle has three dimensional properties, and that foreshortening is taking place.

Eliminating Things You Can't See Anyhow

There are always things that you cannot see in the real world from a particular place. They might be hidden behind something else, or they may simply not be in your field of view. If you hold your arms out from your sides you will be aware that your hands are to your left and right, but probably will not be able to see them. If you wiggle your fingers you can detect the motion, but not count the fingers themselves. Moving your hands towards each other in front of you, you will be able to see them clearly at some point. There is an area in front of your eyes, and for a certain angle on each side, that can be clearly seen. Simulating this involves *clipping* the objects outside of this area.

In fact, it's not an area, it is a volume, and it's defined by the truncated pyramid that we used to define the perspective transformation. We have a volume that contains things you can see, and all objects outside of that volume should be ignored. This volume is cone shaped, but in computer graphics we usually create a polygonal viewing volume of the sort illustrated in Figure 5.6.

Every polygon, vector, and point that lies outside of the truncated prism which is the viewing volume need not be drawn. However, some objects will reside partly inside the volume and partly outside, and so any method that rejects vectors that can't be seen must also clip them so that the visible part is present and the part outside of the view volume is not drawn. This is not a simple task, but fortunately we don't have to get into the details. *Processing* will clip for us once we define the viewing volume.

Hidden Object/Surface Removal

There are two ways in which an object can be not seen—when it lies outside of the viewing volume, and when it is hidden by another object. The subject of hidden surface removal used to be discussed by academics and practitioners until the development of the very popular Z-buffer algorithm. This is only practical when a relatively large amount of memory is available, but fortunately memory has become cheap and most graphics cards these days come with hundreds of megabytes or more.

So, a very quick introduction to the Z-buffer algorithm is relevant here. Not only is it used for hidden surface removal, but similar methods are used by *Processing* and most other graphics systems for performing other tasks, like transparency, shadows, and blurs.

Simply put, a Z-buffer or *depth buffer* is a two-dimensional array the same size as the viewing area. Every pixel displayed on the screen has an entry in the Z-buffer. When a pixel is drawn on the display, it is associated with a distance from the viewer (COP). Objects are usually drawn one at a time, and if the Z-buffer has no entry in it for a pixel to be drawn, we save the distance from the viewpoint to the object at that point in the Z-buffer. If the Z-buffer has a distance in it already, we compare that against the distance of the pixel we are drawing; if it is greater than the current one, we draw the current pixel and save its distance. We never draw a pixel that is farther away than the value in the Z-buffer. Thus, any object that is behind another will not appear on the screen.

One thing that can happen in practice is *Z-fighting* or *Z-flicker*, where polygons seem to flick in and out of existence rapidly as the point of view changes. This is the result of approximation errors in the pixel depth values. Frequently, and this includes *Processing*, depths are stored as floating point numbers in a narrow range, and these are sometimes converted from integer user coordinates. Increasing the accuracy of the values of the buffer usually solves the problem, so if a 16-bit Z-buffer is being used, changing to a 24-bit or 32-bit buffer will solve the problem. *Processing* does suffer from Z-flicker.

Viewpoint

A computer game world is seen from the location of the player's avatar, just as the real world is seen from any person's real position. The player's position changes as they move about in this world, allowing them to see different sides of 3D objects and navigate the world to solve the game. *Processing* allows the specification of the player's position and direction that the player is facing, which are really the parameters that define the viewpoint. The function that sets these parameters is called `camera`:

```
camera(eyeX, eyeY, eyeZ, centerX, centerY, centerZ, upX, upY, upZ)
```

The parameters (`eyeX`, `eyeY`, `eyeZ`) represent the 3D coordinates of the camera or eye itself, the point from which viewing takes place. The 3D coordinates (`centerX`, `centerY`, `centerZ`) represent the center of the scene being viewed, or the direction that the player is looking. The vector (`upX`, `upY`, `upZ`) defines the **up** direction, and is usually something like (0, -1, 0). Like `perspective`, if `camera` is called without any parameters it sets them to the default values, which are:

```

eye = (width/2.0, height/2.0, (height/2.0) / tan( $\pi * 30.0 / 180.0$ )
center = (width/2.0, height/2.0, 0)
up = (0, 1, 0)

```

In most games the eye position will be essentially the avatar position with a little height added. Each time the player moves, this position will change. The center position will be the direction that the player faces, so when the player changes the orientation of the avatar these values will change too. The up vector normally stays the same throughout the game. It's important to note that setting these parameters incorrectly often results in important objects not being seen at all, and this makes it hard to correct the mistake by trial and error. If you are looking in the wrong direction, you won't see the object, and there are many wrong directions.

```

void draw()
{
    background(255);
    perspective();
    camera(eyex, eyey, eyez, cx, cy, cz, 0, 1, 0);

    beginShape(TRIANGLE);
    vertex (T[0][0], T[0][1], T[0][2], T[1][0], T[1][1]);
    vertex (T[1][0], T[1][1], T[1][2], T[2][0], T[2][1]);
    vertex (T[2][0], T[2][1], T[2][2], T[0][0], T[0][1]);
    endShape(CLOSE);
}

void keyPressed()
{
    if (key=='d') eyex += 5;
    else if (key=='a') eyex -= 5;
    else if (key=='w') eyez += 5;
    else if (key=='s') eyez -= 5;
}

```

Let's look at the 3D triangle again. To make certain that we always see it, set the center vector to the center of the triangle: (125, 125, 0). This is cheating and would fix our gaze in a real game, but it is just an example. The up vector will be (0,1,0) which defines the Y axis to be vertical, as usual. The eye position will be at (100, 100, 100) and will be modified by pressing the W, A, S and D keys so that W and S move the viewpoint in the Z direction

and A and D move in the X direction. Running this program (**a4triangle.pde**) one can now move the viewpoint at will, and see that the triangle is in fact in 3D space, although it is still a plane and has no thickness.

The viewpoint is normally not fixed. The way things operate in most games is that the keyboard does not control the player in the X and Z directions as was done here either, but moves *forward* and *backward* and changes orientation to the *left* or *right* as keys are pressed. The forward direction is not the X or Z axis, but is the direction that we're facing, and can be any direction. This means that the facing direction needs to be recomputed each time the player moves. We did something like this when discussing positional audio in Chapter 3, since we needed to know the facing direction to figure out where the sound should appear to come from.

To implement this new movement scheme, we start with a facing direction `FA` that is known. We could define that direction to be 0 degrees, and it will initially look straight down the Z axis. We need the coordinates of the center point, and this can be calculated as follows for some distance `d`:

```
centerX = eyeX + d * sin (FA);
centerZ = eyeZ + d * cos (FA);
```

The idea is to have the avatar looking at some point in the distance `d` straight ahead. When the player types an A or S, the action is to change the facing angle `FA` by a constant turn increment, say 10 degrees and then re-compute the center point. Moving ahead is, in fact, computed the same way, replacing the large `d` value with a smaller step size `s`, to get the distance the character moves in the X and Z directions. The center point must also be recalculated when moving forward and backward or it will change distance from the viewer.

A program that does this is **a5triangle.pde**. The difference between this program and the previous one is the way that the movement is computed, so the new version of the `keyPressed` function is shown to the right, giving the calculation for updating the avatar position and facing vector. The variable `deg10rad` is simply the value of ten degrees in radians, because the trigonometric functions expect to be passed angles specified in radians.

```
void keyPressed()
{
    if (key=='d')    fa -= deg10rad;
    else if (key=='a')  fa += deg10rad;
    else if (key=='w')
```

```

{
    eyex = eyex+(int)(step*sin(fa)+0.5);
    eyez = eyez+(int)(step*cos(fa)+0.5);
}
else if (key=='s')
{
    eyex = eyex-(int)(step*sin(fa)+0.5);
    eyez = eyez-(int)(step*cos(fa)+0.5);
}
if (eyex < 0) eyex = 0;
if (eyeze < 0) eyeze = 0;
cx = eyex + (int)(1000*sin(fa)+0.5);
cz = eyez + (int)(1000*cos(fa)+0.5);
}

```

A Prism

Although the triangle we've been using as an example does use 3D vertices, it is otherwise a 2D object. Moving around it, we can see foreshortening take place, but it lacks depth. The next step in our exploration of the 3D graphical offerings of *Processing* would be to build and use some real 3D objects. Rectangular prisms are simple examples that will expand our knowledge, and are very useful in real life—they can be used as buildings in games.

A cube is an example of a rectangular prism. The sides, top, and bottom are all rectangles. In order to draw such a thing in *Processing*, we need to learn just a little more about drawing polygons in 3D. The `beginShape` function is more flexible than we know, as it allows not just triangles to be drawn but many other groupings of vertices, including `QUADS`, which are rectangles. When drawing a `QUAD` the system expects four vertices per `QUAD`, and the vertices should be specified in either clockwise or counterclockwise order.

```

void draw ()
{
    background (255);
    perspective();
    camera (eyex,eyey,eyez, cx,cy,cz, 0,1,0);
    beginShape (QUAD);
    vertex (0., 0., 0., 0.);
    vertex (sx, 0., 0., 0.);

```

```

    vertex (sx, sy, 0., 0.);
    vertex (0., sy, 0., 0.);
    endShape (CLOSE);
}

```

The code above will draw a single QUAD, a rectangle. The vertices are specified in counterclockwise order. The rectangle is `sx` units in the X direction and `sy` units high, the Y direction. The program **a6quad.pde** uses this draw function and will show the rectangle in the graphics window. To make a prism, we need three more of these as walls, one as the top, and one as the bottom or floor. These rectangles need to be connected—that is, joined along at least one edge. Specifying the coordinates that allow the connection between the rectangles of a prism can be tricky, and so it can be helpful to draw the prism on paper with the vertices specified before transforming the prism into actual code. Figure 5.7 shows just such a pencil sketch, the design drawing for the prism we’re going to draw using *Processing*.

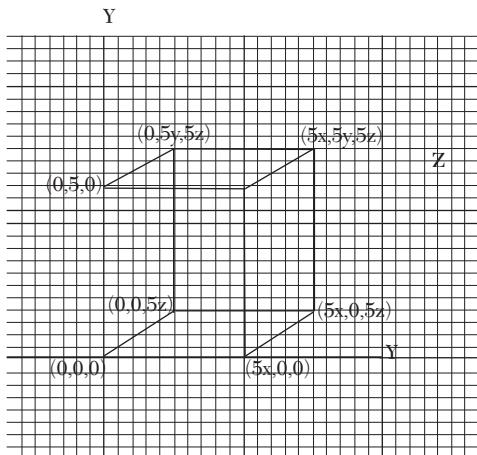


FIGURE 5.7 A sketch of a prism in 3D showing the coordinates of the vertices. It is important to go through a design phase for even simple 3D objects.

```

beginShape (QUAD);
vertex (0., 0., 0.);
vertex (sx, 0., 0.);
vertex (sx, sy, 0.);
vertex (0., sy, 0.);

vertex (sx, 0., 0.);
vertex (sx, 0., sz);

```

```

vertex (sx, sy, sz);
vertex (sx, sy, 0.);

vertex (sx, 0., sz);
vertex (0., 0., sz);
vertex (0., sy, sz);
vertex (sx, sy, sz);

vertex (0., 0., 0.);
vertex (0., sy, 0.);
vertex (0., sy, sz);
vertex (0., 0., sz);
endShape (CLOSE);

```

The near face of the prism in Figure 5.7 is the one we just drew. The next one, on the right side as we see it, would have vertices $(sx, 0, 0)$, $(sx, 0, sz)$, (sx, sy, sz) and $(sx, sy, 0)$. Coding all of the sides is now pretty simple, as can be seen above. Even so, a coding error in the first draft of this code resulted in a triangle being drawn for one of the sides. It's an error prone process at first.

One of the basic ideas in programming is the fact that we can define something in computer code one time and then use it many times. Making the first one may be hard, but then the next ten million are easy. That's true here too. The prism created here is intentionally parametric on the values of `sx` (width), `sy` (height) and `sz` (depth). Any of these values can be changed, and the result will be a prism of a different size. The code on the right is from **a6quad.pde**, and allows you to move around a single prism.

What if we want many prisms? This code can draw as many as we like, so long as they are located at the origin $(0,0,0)$. That's not good enough, but fortunately there's an easy way to draw prisms anywhere we like.

Geometry: Translation

The term *geometry* can be used to describe of all of the manipulations of an object that affect its position and orientation in space. Unfortunately there is some math involved; details can be found in the math tutorial (Appendix I). These operations will be implemented by the *Processing* graphics system, of course, and so they will be described here in those terms. All geometric operators work by systematically changing the coordinates of the

vertices of the polygons that comprise an object. In any 3D graphics system multiple coordinate systems are possible, and this is the source of much confusion. Let's try and keep it as simple as possible.

Translation is the act of changing the position of an object in three dimensional space. Let's align the X-axis to what we think of as the horizontal and the Y axis to the vertical, and imagine a triangle with vertices $(0,0,0)$, $(0,1,0)$, $(1, 1, 0)$; this situation is shown in Figure 5.8. A translation in the X direction by 1 unit corresponds to simply adding 1 to all of the X coordinates in the polygon. We can translate an object by different amounts in all three coordinate directions at the same time by simply adding a different amount to each coordinate. The new (x,y,z) coordinates in terms of the old ones and the three translations could be written out as:

$$\begin{aligned}X_{\text{new}} &= X_{\text{old}} + Tx \\Y_{\text{new}} &= Y_{\text{old}} + Ty \\Z_{\text{new}} &= Z_{\text{old}} + Tz\end{aligned}$$

In Processing a translation is implemented using the function `translate`:

```
translate (Tx, Ty, Tz);
```

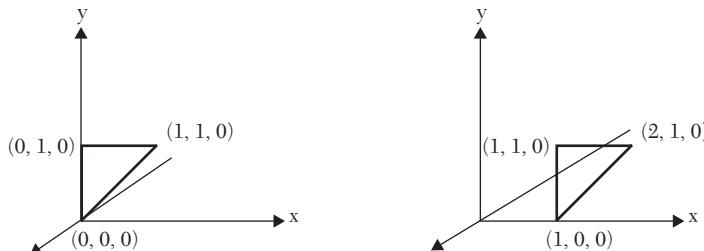


FIGURE 5.8 Translation of a simple shape.

When this function is called, all successive vertices are translated by the specified values. A further (second) call to translate combines the two specified translations, so the call `translate (1,1,1)` followed by `translate (2,2,2)` results in the same thing as a single call to `translate (3,3,3)`. There are two ways to stop performing the translation, one of which would be to call `translate` with negative values of the parameters called in the first place—this effectively translates them back to where they were.

Translation addresses the issue of multiple instances by allowing us to translate to a particular position, draw a prism, then translate to a new position and draw another, as many times as needed. One implementation is

to write a function such as `drawPrism` that takes `sx`, `sy`, and `sz` as parameters and draws a prism with those dimensions at (0,0,0). Then use this to implement the function `drawPrismAt` that translates to a specific (x,y,z) locations, draws a prism there, then translates back. Now we can draw as many prisms as we choose, wherever we like, of any dimensions we like. The program **a7prisms.pde** allows you to move around a group of seven prisms drawn this way.

```
void drawPrismAt (float x, float y, float z,
                  float sx, float sy, float sz)
{
    translate (x, y, z);
    drawPrism (sx, sy, sz);
    translate (-x, -y, -z);
}
```

Geometry: Scaling

A scaling transformation increases or reduces the size of the object as it appears on the screen. It seems obvious that if we want an object to get bigger, say twice as large, then we could multiply each coordinate in each vertex by two. We could also scale each dimension by a different amount by multiplying each dimension by a different value. If the object is not centered at the origin then scaling it also appears to translate it—the multiplication by a scale factor makes the coordinates larger or smaller, and this changes their distance from the origin by the same factor.

When using *Processing*, a call to `scale` changes the current scale value, which defaults to 1.0. The call `scale (2.0)` multiplies all successive vertices by a factor of 2, doubling an object's size by a factor of 2 in every dimension (as shown in Figure 5.9). The call `scale (1.5, 2.0, 2.5)` increases the scale in each dimension by a different amount.

Mathematically, the new coordinates can be expressed in terms of the old ones as:

$$X_{\text{new}} = S_x * X_{\text{old}}$$

$$Y_{\text{new}} = S_y * Y_{\text{old}}$$

$$Z_{\text{new}} = S_z * Z_{\text{old}}$$

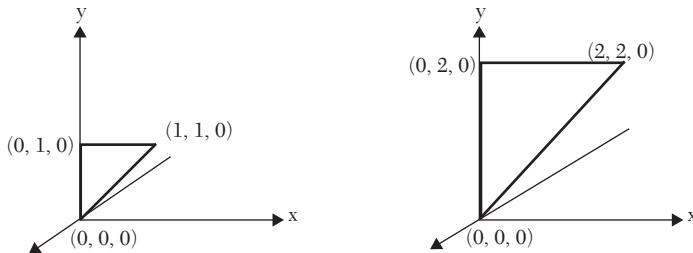


FIGURE 5.9 Scaling a triangle by a factor of 2 by calling `scale(2.0)`.

Geometry: Rotation

Rotation is a circular motion about a specified point or axis by a specified angle. It is easier to explain rotation in two dimensions first, and that's the situation shown in Figure 5.10. On the left side of the figure we see an X-Y coordinate axis with a pixel drawn at the point (2, 0). On the right we see the same axis, but now the pixel has been rotated about the origin (0,0) point by 45 degrees. What are the coordinates of the rotated point? To answer this, it would be easiest to use some basic trigonometry.

In this case the angle θ is 45 degrees, and the length of the line joining the origin to the point (the hypotenuse) is still 2 units. So the following trigonometric equations hold:

$$\begin{aligned}\sin \theta &= y_\theta / 2 \\ \cos \theta &= x_\theta / 2\end{aligned}$$

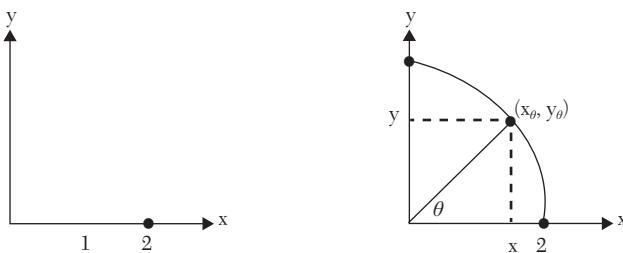


FIGURE 5.10 Rotation of a point in two dimensions.

So we can rearrange these to find the point (x_θ, y_θ) as $(2\sin \theta, 2\cos \theta)$. However, this only works if we're rotating about the origin. If we are instead rotating about some point P, we can move the object to the origin (translate), rotate, and then move it back.

In Processing rotations are done around the origin by the function `rotate(angle)` for any angle in radians. As with `translate` and `scale`, the

rotation angle specified is applied to all further vertices until it is changed, and further calls to `rotate` add the new rotation angle to the old one in the order that they were applied. Order matters now.

The general equations needed to rotate any point (x,y) about the origin by θ degrees are:

$$\begin{aligned}x' &= x \cos \theta - y \sin \theta \\y' &= x \sin \theta + y \cos \theta\end{aligned}$$

In three dimensions a number of things are different. First, we don't rotate about a point, but about a line, usually one of the three coordinate axes. Second, there are three coordinates, not two, and so there are three equations. Finally, a rotation about an arbitrary line can be implemented as a translation and a number of rotations about the coordinate axes, up to three of them. *Processing* allows rotations about the origin with respect to any coordinate axis. The calls are:

```
rotateX (a1);
rotateY (a2);
rotateZ (a3);
```

Colors, Shading, and Textures

So far the 3D graphical environments that we've built have been pretty simplistic, but all of that is about to change. The polygons that have been rendered look simply like rectangles, with no other structure. The first and easiest thing to do is add color. The function `fill` works as it did before, and if a fill color is specified it will be used as the color of any polygon drawn in successive `beginShape`-`endShape` blocks. Figure 5.11 shows a set of prisms as drawn by **a7prisms.pde**, and the same set obtained after adding a different fill color to each face. It's not *Myst* yet, but it shows what can be done.

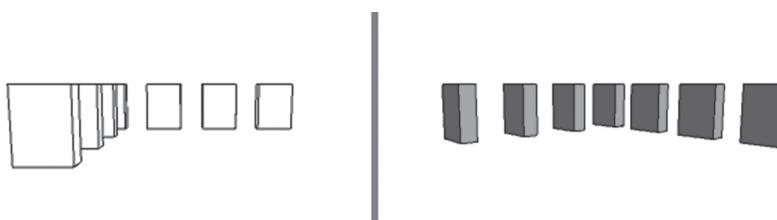


FIGURE 5.11 Colors on faces of the prisms.

3D graphics and game graphics generally are about reality. High-resolution graphics are not beyond the capabilities of *Processing*, but they are not what it was designed for, and the computing requirements might be more than it can handle. It can deal with graphics in the middle and low range of resolution and with a good degree of realism. The next step in our discussion would be the use of shading and textures to make the graphics look more realistic.

We've seen that a sphere can be built out of polygons, and it has been implied that any object can be reduced to a set of polygons that can be manipulated by simple geometrical operations. However, the results looked like a wire frame mesh of lines. What we want is for the polygonal surface to present the effect of a solid object. This is accomplished by shading the polygons so that they appear to be a smooth surface, or by applying a real-world texture to make the polygonal surface look like a real one. Let's look at texturing and shading separately.

Texturing

A texture is a characteristic visual pattern seen on real-world objects. Textures sometimes repeat the visual effect as a function of distance, but often the repetition is not perfect, and sometimes is nearly random. Wood grain, concrete, and metal are all textures that we might use in a game. For example, if we wish to build a wooden fence around a yard in a computer game, we could make the fence out of polygons and cover the polygons with a wood grain texture. Making a building involves constructing a prism and then covering it with textures of concrete, paint, doors, and windows. These are visual simulations of real objects, hollow imitations of the real things; but that's what a game is. It's a simulation, in which objects are not real and things need to look good rather than be correct. This is not a criticism, but an observation that developers need to understand.

A texture is an image. So, how do we map the pixels of an image onto the locations occupied by the polygons that represent an object? The theory is interesting, but let's watch it happen first. Convincing *Processing* to do texture mapping onto polygons has three steps to it. First, we have to read the texture into memory; textures are images, and are stored in an image file format like GIF or JPEG. We already know how to read images using *Processing*. Next we must map the vertices of the polygon onto texture coordinates by hand—that is, every polygon vertex we will be plotting must be associated with a row and column coordinate in the texture image. Remem-

ber, the polygons are in 3D space and the texture is in 2D space. It's also important to understand that a perspective transformation is being done, and so the texture needs to have that transformation applied to it as well.

Finally the polygons are drawn, covering them in the process with portions of the texture. Once we have defined the mapping then this part is almost automatic, taking place when we draw the polygons because *Processing* kept track of the polygons that have textures and what they were. So really, texture mapping is all about setting things up so it can be done automatically. A polygon, even a 3D polygon, is two dimensional, so drawing the texture onto it is possible. It's no different from drawing an image on the computer screen, except that the pixels have to pass through the perspective transformation first.

We do need to define how the mapping of texture image pixels to polygon pixels is going to take place. It is easier for a QUAD, since both an image and a QUAD are rectangular. In this case, let the upper left coordinate of the texture corresponds to (0,0) on the polygon. The next corners to connect should be in a counterclockwise rotation, so it will be the lower left on the texture image. The vertex this corresponds with on the polygon depends on the orientation of the polygon in space, so for the purposes of this example assume that it lies on the X-axis. Then, based on the size of the rectangle being `sx` width and `sy` high, the coordinate corresponding to the lower left of the texture will be (0, `sy`, 0). Assignment of texture to polygon vertices continues in a counterclockwise direction, and depends on how the **up** vector was defined in the call to `camera`.

The way the assignment is performed is by using a new version of the `vertex` function that has two extra parameters: the coordinates on the texture image that correspond to this vertex. The first vertex call, the mapping of the upper left coordinates, would be done as:

```
vertex (0., 0., 0., 0., 0.);
```

This explains very little since all coordinates are zero. The next vertex call would map (0, `sy`, 0) on the polygon to a texture coordinate, and might be:

```
vertex (0., sy, 0., 0., 340.);
```

The first three parameters of the vertex call are as before, the 3D coordinates of that vertex in space. The final two parameters are the x and y coordinates of the corresponding pixel in the texture image. In this case the

number 340 would be the height in pixels of the texture image, or the index of the final row of pixels therein. The use of the actual size of the texture image is sometimes awkward. Actual image sizes will not be the same from texture to texture, and things can get confusing easily. As a result graphics people have defined what they *call texture coordinates* (**u**, **v**), which are simply x and y coordinates of pixels in the texture image. Values of **u** and **v** are real, and run from 0.0 to 1.0. This means that all images use the same texture coordinates, which map onto any actual pixel coordinates. The use of 0-1 texture coordinates in *Processing* is specified by a call, just once, to the function `textureMode (NORMAL)`, where **NORMAL** means normalized texture coordinates as we've just explained.

A polygon having a texture mapped onto it can be specified using the following code:

```
textureMode (NORMAL);
beginShape (QUAD);
texture (texture1);
vertex (0., 0., 0., 0., 0.);           // Upper Left
vertex (0., sy, 0., 0., 1.);          // Lower Left
vertex (sx, sy, 0., 1., 1.);          // Lower Right
vertex (sx, 0., 0., 1., 0.);          // Upper Right
endShape (CLOSE);
```

The call `texture(texture1)` defines the image `texture1` as the image to be drawn over the polygon, which will probably be a `PImage`. The following four vertex calls define how the corners of the texture image map onto the polygon, and by the way they also draw the polygon. The situation is complex enough that it is worth drawing a sketch and marking the polygon and texture coordinates on it before writing the code. In particular, the 3D coordinates of the polygon might not be orthogonal to the axes, and this complicates the mapping to 2D texture coordinates.

```
void drawBuilding (float x, float z)
{
    translate (x, 0., z);

    beginShape (QUAD);           // Front
    texture (textureFront);
    vertex (0., 0., 0., 0., 0.);
    vertex (0., sy, 0., 0., 1.);
```

```

vertex (sx, sy, 0., 1., 1.);
vertex (sx, 0., 0., 1., 0.);
endShape(CLOSE);

beginShape (QUAD);           // Side
texture(textureSide);
vertex (sx, 0., 0., 0., 0.);
vertex (sx, 0., sz, 0., 1.);
vertex (sx, sy, sz, 1., 1.);
vertex (sx, sy, 0., 1., 0.);
endShape(CLOSE);

beginShape (QUAD);           // Rear
texture(textureBack);
vertex (0., 0., sz, 0., 0.);
vertex (sx, 0., sz, 0., 1.);
vertex (sx, sy, sz, 1., 1.);
vertex (0., sy, sz, 1., 0.);
endShape (CLOSE);

beginShape (QUAD);           // Side
texture(textureSide);
vertex (0., 0., 0., 0., 0.);
vertex (0., 0., sz, 0., 1.);
vertex (0., sy, sz, 1., 1.);
vertex (0., sy, 0., 1., 0.);
endShape (CLOSE);

translate (-x, 0., -z);
}

```

To make a large jump in complexity, let's make a bunch of buildings. This means texturing the four sides of a prism, then creating many instances of that. We need textures for the front, sides, and back of the buildings. *Paint* was used to make some (**Front.jpg**, **Side.jpg**, and **Rear.jpg**), but photos of real buildings work, and other tools can be used to make textures. The program **a10buildings.pde** (above) makes a building by drawing a rectangular prism and texturing each face with one of these textures. A function called `drawBuilding` was written that—like `drawPrismAt`—draws a prism at a particular (x,z) location in space and textures the faces. Then a

20×20 array of these buildings was drawn in multiple calls of `drawBuilding`, leaving some spaces between buildings for the avatar to walk through.

This has many flaws as game code, not the least of which is that all of the buildings are the same and they don't face the street properly. It does show how a large virtual space can be created with relatively little code—there are 400 buildings here (Figure 5.12).

Theory

Texture mapping involves a lot of work, and a certain amount of math. Here's how texture mapping works.

Images are usually rectangular arrays of pixels, and so mapping them onto rectangles is especially easy. If the texture image T is N rows and M columns and the rectangle R being mapped onto is N' by M' pixels then the map is essentially a scaling operation, and can be implemented as an interpolation. For example, a pixel in row i of the texture is i/N of the vertical distance from the top of the image. The corresponding position in rectangle R is $(i/N)*N'$ from row 0.

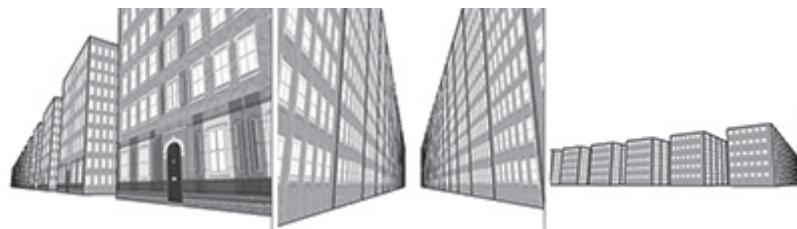


FIGURE 5.12 Texture mapping images (buildings) onto polyhedrons.

This mapping is actually backward, because by going from texture to object we may have object pixels that are never set. Going from object pixels to texture pixels will ensure that every pixel visible to a viewer with a reasonable value.

The essential code for mapping a texture onto a rectangle is:

```
for (i=0; i<N'; i++)
{
    y = (float)i/(float)N';
    I = y*N;
    for (j=0; j<=M'; j++)
    {
        x = (float)j/(float)M';
        J = x*M;
```

```

    mapped[i][j] = texture[I][J];
}
}

```

If the rectangle being mapped is in 3D space, it may be viewed from an oblique angle. If this happens the rectangle will look like a parallelogram or a rhombus, and the mapping from the texture image (a rectangle) seems harder than before.

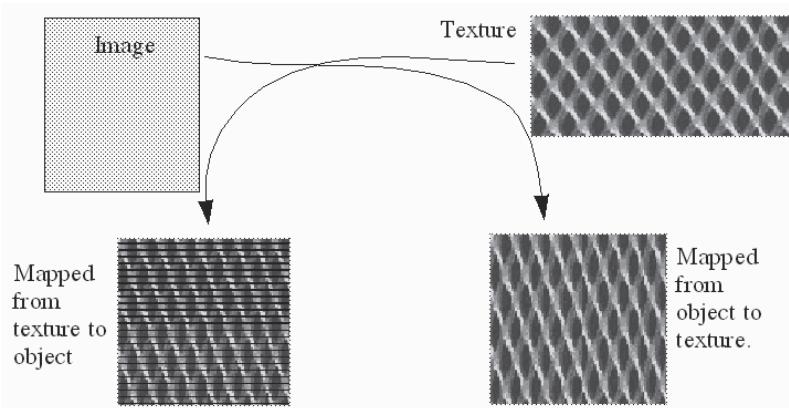


FIGURE 5.13 Simple texture mapping. Mapping from object to texture works, but from object to texture leaves holes.

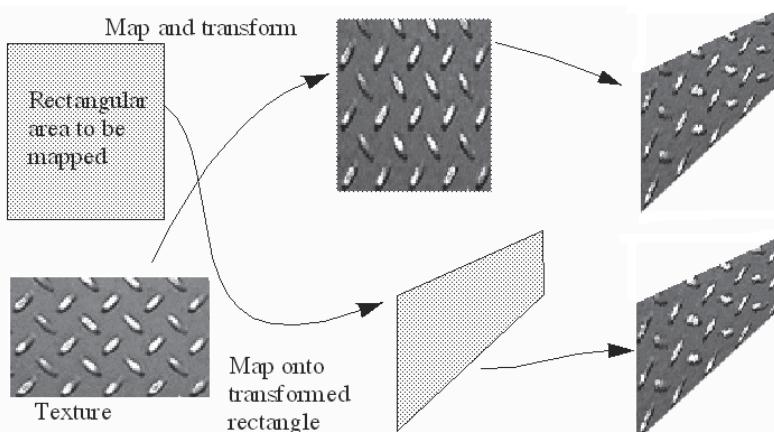


FIGURE 5.14 Mapping a texture onto a rectangle that has been transformed by a perspective transformation.

However, the object coordinates can map pixels from the texture before the perspective transformation is done—the texture pixel is transformed by

the viewing transformation in the same way as any other pixels, and we don't have to do anything special.

On the other hand, mapping a texture onto a sheared rectangle is not difficult. The following code will do it, assuming that the function `I(j)` returns the minimum index in column j and that `II(j)` returns the maximum index in column j :

```
for (j=j0; j<jn; j++)
{
    i0 = I(j);
    i1 = II(j);
    x = j/M;
    jj = x*M
    for (i=i0; j<=i1; i++)
    {
        y = (i-i0)/(float)(i1-i0);
        ii = y * N;
        mapped[i][j] = texture[ii][jj];
    }
}
```

Texture Mapping To a Triangle

The polygons most likely to be used to represent objects in a typical game are rectangles and triangles. We have seen how to map a texture onto a rectangle and a rhombus, so it remains to do a triangle. As before, the idea is to do an interpolation, but the sides of a triangle are not parallel to each other, so there is a slight variation.

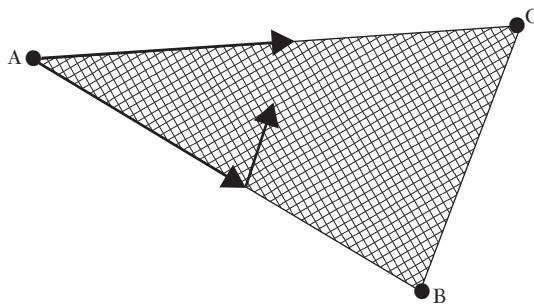


FIGURE 5.15 Texture mapping to a triangle.

As seen in Figure 5.15, a first step is to determine the texture pixel values at two points along two of the triangle's sides. This is done by interpo-

lating the values between the common point **A** and the two other vertices. Then we look in the BC direction and interpolate between the two points we just computed on the sides.

It is sufficient to see how this can be done. *Processing* performs the details of the mapping, and does it by asking the graphics card to do it if that facility is available.

Object Models

At the beginning of this chapter we mentioned that any object can be built from collections of 3D polygons, and showed an example of spheres of various degrees of detail (Figure 5.1). No attention has been paid to *how* these models are constructed, because that falls into a different department in the game development team: the art department. There are programs that enable artists to build 3D objects and save them as files, just as we save textures as BMP or JPG files. And, just as with images, there are many file formats in use for saving models as files, and each tool will use only a few of them. The most common model creation tools include *3D Studio Max* (3DS Max), which uses files having the suffix .3ds, and *Maya*, which uses .obj files.

```
// Cycle created by Herminio Nieves @2013
// for comercial and non comercial use.
```

```
PShape obj;
final float PI = 3.14159265;
float sx=30., sy=40., sz=12.;
int eyex= 280, eyey= -50, eyez=44;
float fa = 4.5;
int cx=24, cy=0, cz=0;
float deg10rad;
int step = 20;

void setup()
{
    size(600, 400, OPENGL);
    deg10rad = ((PI*2)/360.0)*10.0;
    obj = loadShape("cycle-x.obj");
    sx = sx*5; sy=sy*5; sz=sz*5;
    cx = eyex + (int)(1000*sin(fa));
    cz = eyez + (int)(1000*cos(fa));
}
```

```

void draw()
{
    background(200);
    perspective();
    camera(eyex, eyey, eyez, cx, cy, cz, 0, 1, 0);
    rotateZ(PI);
    shape(obj, 0, 0);
    obj.translate(200, 20, 20);
    shape(obj);
}

```

These programs can be costly, and are worth it if you are a professional game developer. If you are looking for a free program, *Blender* is competent. The way these programs work is not important; that they export a file containing 3D polygons is critical, because a modern game requires a model for every object it uses and has to import them (read the model file) when the game begins. It also has to move the models into place in the 3D world, and some of the models are under the control of the player, and so can move within the game space.

Can *Processing* use models? Well, sort-of. There exists an object model importer for *Processing* that was created by interested third parties and that can be installed so that *Processing* can easily use it. It is the *ObjLoader* program. The bad news is that many models these days have a lot of polygons, and the *ObjLoader* loader can be slow. It also does not deal with all formats, just .obj files, and not all versions of those. Still, it can load and manipulate some models and that gives a functionality we can use.

The *ObjLoader* program can be downloaded and made available to *Processing* in about five minutes. After that, using it involves three steps.

1. Import **processing.opengl.*** at the beginning of the program, and specify it in the `setup` call.
2. Load the shape or shapes. If there are not too many this can be done at the beginning, also in `setup`. If they take too much memory then you may have to devise a scheme for loading them when needed. Load an object as follows:

```
obj = loadShape("cycle-x.obj");
```

where **cycle-x.obj** is the name of the object file, and **obj** is a variable of type **PShape**; that is, declare:

```
PShape obj;
```

3. After setting up the projection and camera, display the object by calling:
`shape (obj);`

The program **a11object.pde** is a *Processing* program that loads and displays a futuristic motorcycle stored in the .obj file mentioned above. One thing to note is how slowly the screen is updated after moving the point of view. By way of explanation, this model consists of 14,085 triangles, more than you probably expected. It takes time to draw and texture this many polygons. Figure 5.16 shows what this cycle looks like when rendered properly with good illumination and a high quality renderer, and also how it appears in the *Processing* window. There is a distinct difference, only part of which is the better illumination.

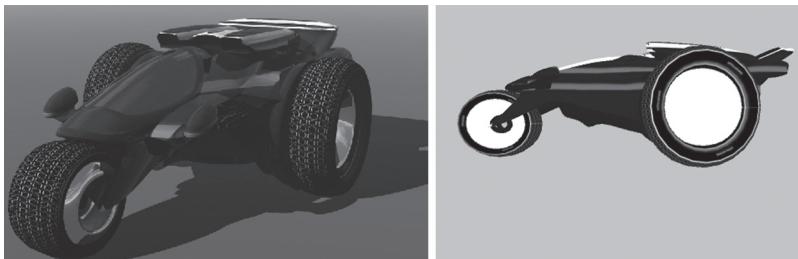


FIGURE 5.16 (Left) The futuristic cycle rendered in high quality form. (Right) The same model rendered in *Processing* using *ObjLoader*. (Cycle created by Herminio Nieves ©2013)

When this program was running, one entire processor of a 4-core CPU system, running at 3.2 GHz was totally occupied running Java.

It is certain that the speed of the rendering could be drastically improved, and that the quality could be as well. It should be remembered that *Processing* is free, and that few people are paid for the development of enhancements and tools.

PShape

A `PShape` object is a general structure for holding arbitrary shapes. Shapes are either loaded from files (using the `loadShape` function) or created by the user using drawing operations (using the `createShape` function). There are also a set of built-in shapes, like `SPHERE` and `RECT` that can be created.

Once shapes exists, there are a variety of operations that are possible that work on those shapes. They can be rotated, scaled, translated, and textured. They possess a local set of parameters, and can have their stroke and fill color set as distinct from the global values. And of course they can be drawn.

Complete documentation for his type can be found online, as a ‘Java-Doc.’

Summary

We have covered quite a lot in only a few pages, and it seems that this material is not related to all games. Games all have a foundation in graphics, audio, AI, and programming that is extensive, and teacher’s job is to make sure that the entire class has at least the same basic background. This is not all we need to know about graphics, and the references will have much to say about more advanced subjects such as performance issues, mip-mapping, bump maps.

At this point, anyone who understands what has been explained so far will be able to start designing and writing a 3D game.

Exercises



The exercises below will apply your knowledge of the material in this chapter, and sometimes require that you do some more research before you are able to complete them. Solutions to selected exercises are available on the companion disc.

1. Write a Processing function named **mycube** that draws a cube that is 200 pixels on a side, and with the lower left corner at (0,0,0). Use calls to the **line** function to draw the edges. View the cube by calling **mycube** from **draw** using a perspective projection with a viewpoint of (100, 100, 400) and a viewing center of (100,100,100), the center of the cube. Which lines of the cube represent the X, Y, and Z coordinate axes?
2. The processing function **sphere(r)** draws a sphere of radius **r** at coordinates (0, 0, 0). Draw a sphere of radius 12 inside the cube created for Exercise 1 above. A call to the function **lights()** will make the sphere look shaded, and a call to **noStroke()** will eliminate the edges of the polygons. Make the sphere move inside the box, bouncing off the sides.

3. A **PShape** object can be used as a sphere too. Read the online documentation for **PShape** and repeat the solution to Exercise 2 using **PShape**. Map a texture of the planet Earth onto the sphere—this requires the use of the **PShape** functions `setStroke` and `setTexture`.
4. Draw a rotating sphere, textured with a map of Earth, centered in the field of view.
5. A panoramic photograph gives a flat 360° view of a place. Find a panoramic photograph online by searching for “panorama interior” and download it. Write a *Processing* sketch that places the viewer inside of a sphere onto which the panorama has been mapped. Have the view direction change in x or y as the up/down (Y) or left/right (X) arrow keys are pressed.
6. Use modeling software of your choice (*Blender* is free) to create a sphere. Using this, build a dumbbell (two spheres joined by a bar). Import this into *Processing* and display it.
7. Exercise 2 involves having a sphere bounce in a box. If there were two spheres, then it is possible that they could collide with each other. Suggest ways that such collisions could be detected and implement one if possible: have a box with two spheres moving within it, and handle sphere-sphere collisions.
8. A *particle* system uses a large number of small particles (circles and ellipses) to simulate more complex graphical phenomena like fire and smoke. An example of a simple 2D particle system can be found on the *Processing* Website (<http://processing.org/examples/simpleparticlesystem.html>). Using **PShape** and the built-in SPHERE shape, create a simple 3D particle system for smoke. Create many spheres of various sizes and velocities and with small color (grey) variations and have them move upward in the field of view.

Resources

Blender web site: <http://www.blender.org/>

Blender 3D Design course, online. <http://ocw.tufts.edu/Course/57/Coursehome>

PShape JavaDoc: <http://www.processing.org/reference/javadoc/core/processing/core/PShape.html>

ObjLoader (Saito object loader) Website: <http://code.google.com/p/saitoobjloader/>

ObjLoader documentation: <http://thequietvoid.com/client/objloader/reference/index.html>

Textures for mapping onto spheres for use as planets: <http://freebitmaps.blogspot.ca/2010/11/srgb-planet-serendip.html>

Bibliography

Dunn, F., and I. Parberry. *3D Math Primer for Graphics and Game Development, Second Edition*. Boca Raton, FL: Taylor & Francis Group (A.K. Peters), 2011.

Foley, J., A. van Dam, S. Feiner, and J. Hughes. *Introduction to Computer Graphics*. Addison-Wesley Professional, 1993.

Gortler, Steven J. *Foundations of 3D Computer Graphics*. Cambridge, MA: MIT Press, 2012.

Shirley, P., M. Ashikhmin, and S. Marschner. *Fundamentals of Computer Graphics*. Boca Raton, FL: Taylor & Francis Group (A.K. Peters), 2010.

Xiang, Z., and R. Plastock. *Schaum's Outline of Computer Graphics*. New York, NY: McGraw-Hill, 2000.

CHAPTER 6

GAME AI: COLLISIONS

In This Chapter

- How to make collisions look realistic
- Collision detection packages

The words *artificial intelligence* bring to mind a host of advanced technology. Often our first exposure to AI, as it has come to be known, is through science fiction—the computers on *Star Trek* can speak fluent English, and the robots on *Star Wars* can serve drinks and pilot spacecraft. In truth, AI has not advanced nearly this far, and although computers can now defeat humans at chess and checkers, this is a far cry from the scenes we see on television and movies.

So what is AI really? Historically this subject has been called *cognitive simulation*, and that is probably a more descriptive phrase for what is happening. AI is an effort to simulate the actions and responses of an intelligent creature. Why would a game want to have simulated intelligent creatures? To simulate other intelligent creatures, allies and opponents, of course. At a high level, AI is used in games to implement other people performing intelligent tasks. These simulated persons, sometimes called *bots* or simply

opponents, are expected to behave in a manner that would be normal for a person. They do not have to be actually intelligent.

At a lower, more practical level, the AI in a game keeps track of things—cars, people, trees, roads, and such. One of the most important tasks of the AI system is to determine when two things collide. This is because collisions are often key points in a game. A missile collides with its target, and the target is destroyed. A hockey player collides with another player and loses the puck. A car collides with a concrete bridge support and takes damage and changes direction. All of these require that collisions be detected, and that the location of the impact and its exact time be known.

It is also true that a good portion of the game AI system can be occupied with performing physics calculations. After a collision takes place, the result is that something breaks, or changes direction, or falls down. A good approximation of real-world physics is essential for a realistic looking game, especially a sports or driving game, and accurately determining the properties of collisions is a crucial first step in correctly simulating physics.

A professional programmer or engineering student would have some knowledge of academic AI, which can be a very exciting subject. However, there is very rarely time enough to use those methods in a computer game. Yes, there may be time to use a neural network that has been trained in advance to accomplish a particular task, but there will not be time to train such a network to handle changing situations. A game programmer must have a practical view of AI, and in a game this means speed and simplicity. Most decisions in a game are made using simple look-up tables or decision trees.

All of the parts of the AI system will be discussed here, including opponents, collisions, plans, and physics simulation. We'll start with collision detection because it is the basic thing that we must get right. If a game includes collisions then collision detection must be fast and accurate if that game is to be playable, and we can't add it on at the last minute—it will be an integral part of the game from the start. There are packages that will do the hard work for us, but it is always dangerous to have code in your program that you don't have a basic understanding of, so it is worth at least skimming this part before downloading someone else's code. Having said this, there are entire books written on the subject, and this chapter merely touches on some of the basic ideas. You should use an existing collision detection system until you feel you want do it from scratch.

Collision Detection

At the outset, the nature of the problem needs to be understood. In the real world, when two objects collide the result is a physical response: sound, heat, energy transfer, and so on. In a game objects are not real, they are numbers representing how an object looks and where it is in a virtual space. When they collide, it is a virtual collision with no effect unless we detect that collision and simulate an effect. This is one way in which computer games are in fact simulations.

There are two major problems associated with collision detection in computer games. The first is complexity. A small game may have 100 or so objects active at a time. If they are all moving, we have to look at each pair to see if they will hit each other, which means about 100×100 , or 10000 tests. We do this each time interval, which is usually the time between two frames - say 24 times per second. This implies a quarter of a million tests for collision per second in a small game, and over a million in a fair sized one. Any solution we implement must be fast.

The second problem is that time in a game is *quantized*. Let's say we again have 24 images per second displayed on the screen, which is the speed of a motion picture. It is possible, even likely, that two objects approaching each other fast enough will pass each other in the time between two frames, 1/24 of a second. So they would have collided, but at time T they were some distance apart, and at **T+1** they had changed places. This happened in *Hockey Pong* with respect to the puck and the paddles. We can do two things in this case: look ahead (predict the time at which they will collide), or look back (to the past time when they did collide). Even a game can't easily roll time backward, so the former solution is best. We must figure out when they would collide, figure out the results of that collision, and draw only the *result* at time T+1.

There are other troubles, some dependent on the algorithm we choose, but these are the big ones. Let's pick an easy problem first.

One-Dimensional Collisions

We will start with a one-dimensional case, which can give us some intuitive insight into the more complex ones. In this case we have motion of particles along a line, and to make it even easier we will have only two particles. There are really only a few possible situations that can occur:

Particle 1			Particle 2		
Case	Position	Speed	Position	Speed	Name
1	0	+2	4	-2	Approach
2	0	+2	4	+1	Overtake
3	0	+2	4	+3	Escape
4	0	-2	4	+2	Diverge

Position is a distance along a line, let's say the X axis. *Time* is in integral units, frames perhaps. In case 1 above, the two particles are heading toward each other and a collision will occur. In this specific situation, the speed of the particles dictates that the collision will take place in exactly one step, since particle 1 will be at position 2 then, and so will particle 2. This is unusual—if particle 2 had any other speed, then the time of the collision would fall between two time units.

Case 2 is also a collision, since at some point in time, easily computed, the faster particle (1) will overtake the slower one, colliding from behind. In the other two cases there is no collision, either because the particles are moving in opposite directions (4) or because one is receding from the other (3).

The collision problem—as would be found in an imaginary one-dimensional game—is to determine whether a collision will occur between any two particles between time T and time T+1, and to compute the precise time of the collision. A precise statement of the problem is essential in finding a correct solution.

A brute force solution might be a good first step, and corresponds to a simulation that you could perform with some graph paper and some small objects representing the two particles. It is now time **T**; for each particle **i**, having position **P_i**, calculate the value of $\mathbf{D}_{ij} = \mathbf{P}_i - \mathbf{P}_j$ for all other particles **j**. Think of this as a distance to all other particles. Now compute the position of the particles at time **T+1**, and call these positions **P'**. Calculate $\mathbf{D}'_{ij} = \mathbf{P}'_i - \mathbf{P}'_j$ for all **i** and **j**. If the sign of \mathbf{D}_{ij} is not the same as that of \mathbf{D}'_{ij} then particles **i** and **j** collide between time **T** and **T + 1**. What this means is that if at one time particle **i** is to the left of particle **j**, and then later on **j** is to the left of **i**, then they had to have passed each other, and in one dimension this means they would have collided.

If $\mathbf{D}_{ij} = \mathbf{0}$ the two particles are in contact at time **T**, and if $\mathbf{D}'_{ij} = \mathbf{0}$ they are in contact at precisely time **T + 1**. Otherwise, if a collision occurs we still have the problem of figuring out when; of course if no collision happens,

this second step is not needed. Imagine that we have two particles **A** and **B**, and that particle **A** is at position $\mathbf{P}_{A\text{initial}} = \mathbf{0}$ with velocity $\mathbf{V}_A = 2$ and particle **B** is at position $\mathbf{P}_{B\text{initial}} = 4$ with velocity $\mathbf{V}_B = -3$. Right now, at time T , $\mathbf{P}_A - \mathbf{P}_B = \mathbf{0} - 4 = -4$. At the next time, $T + 1$, $\mathbf{P}_A = 2$ and $\mathbf{P}_B = 1$, so $\mathbf{P}_A - \mathbf{P}_B = 1$, and the sign changes implying that a collision happens.

When does it happen? Simple physics says that the position of particle **A** is given by $\mathbf{P}_A = \mathbf{P}_{A\text{initial}} + \mathbf{V}_A t$, for any time t ; similarly, $\mathbf{P}_B = \mathbf{P}_{B\text{initial}} + \mathbf{V}_B t$. When they collide, $\mathbf{P}_A = \mathbf{P}_B$, so simple algebra says:

$$\begin{aligned} \mathbf{P}_{A\text{initial}} + \mathbf{V}_A t &= \mathbf{P}_{B\text{initial}} + \mathbf{V}_B t \\ \rightarrow \quad \mathbf{P}_{A\text{initial}} - \mathbf{P}_{B\text{initial}} &= (\mathbf{V}_B - \mathbf{V}_A)t \\ \rightarrow \quad t &= (\mathbf{P}_{A\text{initial}} - \mathbf{P}_{B\text{initial}}) / (\mathbf{V}_B - \mathbf{V}_A) \end{aligned}$$

This is the time at which the collision takes place. The value of t is between 0 and 1, and is relative to the initial time T . If the collision is perfectly elastic, meaning that no energy is lost, then the final step is to figure out where the particles will be at time $T + 1$.

At time t both particles are at position $\mathbf{X} = \mathbf{P}_{B\text{initial}} + \mathbf{V}_B t$. Then an elastic collision happens, and they reverse direction, we will assume with the same speed. Now, time $T + 1$ is 1 time unit from time T , and is $1 - t$ units from the collision time t . In that time, both particles move: particle **A** moves with velocity -2 from position \mathbf{X} for time $1 - t$ units, particle **B** moves with velocity $+3$ from position \mathbf{X} for $1-t$ time units.

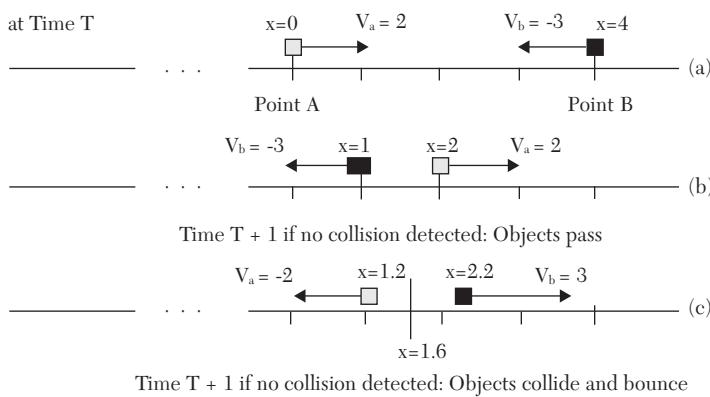


FIGURE 6.1 A one-dimensional collision. (a) The initial situation, time T . (b) The locations of the objects after 1 time unit if no collision occurs (i.e. the objects pass through one another) (c) Locations of the objects after they collide at time $T+0.8$ at location $x=1.6$.

Plugging in the numbers we get $t = 4/5 = 0.8$, $X = 1.6$, $1-t = 0.2$. The final position of **A** is $\mathbf{X} \cdot \mathbf{V}_A(1-t) = 1.2$ and of **B** will be $\mathbf{X} \cdot \mathbf{V}_B(1-t) = 2.2$. Newton never lies (he just approximates).

In summary, a computer program that displays the positions of moving points in one dimension would have, in two consecutive frames:

Time T: point A at $P_A = 0$ $V_A = 2$ point B at $P_B = 4$ $V_B = -3$

Time T+1: point A at $P_A = 1.2$ $V_A = -2$ point B at $P_B = 2.2$ $V_B = +3$

The points would never be seen to touch, since that occurred between two frames, and so would not have been rendered.

This simple one-dimensional (1D) situation is not going to help us in a game, but it does give an idea of how to solve the same problem in higher dimensions. A simulation of 1D collisions is provided on the companion disc. It's called **collision1d**, and shows two particles nearing each other along a line. When they collide, the simulation halts and permits you to step through a short time interval, forward and backward, so you can see the problem. The program is written in *Processing*, so you can modify it, change speeds or sizes, and explore the issues as you choose. Figure 6.2 shows a sample set of screen shots from this program.

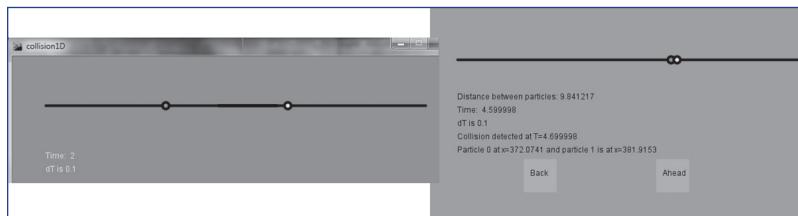


FIGURE 6.2 Output from the 1D collision simulator.

Two-Dimensional Collisions

The two-dimensional algorithm equivalent to the collision detection method above is quite similar, but there are complications due to the extra dimension. In particular, just because two particles have changed places does not mean that they collided. Now each particle occupies an area, and they collide only if the areas would also overlap. So we have a two-step test: first determine whether collision was possible, then determine whether it actually happened. In a real game, most objects will not be involved in collisions during any particular time period, so a fast test that rejects most potential collisions is a good thing.

In 1D we were concerned with points, but mathematical points cannot collide in 2D—they have no area, technically. So we will be concerned with moving polygons on a plane, and in detecting collisions between these. Again, let's name two polygons **A** and **B**, and give them positions and velocities at time **T**. This time, a position is a 2D vector. So is a velocity, and each component of a velocity vector is the speed in a direction, X or Y. Instead of computing a simple distance to determine whether a collision is possible, we must do something more complicated. Since a polygon contains N vertices and all of them are moving at the same speed and direction, we can easily define a straight line that corresponds to the path taken by each vertex in the time between **T** and **T+1**. We can also easily compute the position of all of these vertices at both times. A simple rule that excludes a collision between **A** and **B** is: if we compute a line **L** that represents the path of a vertex of **A**, then a collision cannot have taken place between that vertex of **A** and polygon **B** if **B** is on the same side of that line at both time **T** and **T+1**.

Take a look at Figure 6.3. Here we see a couple of polygons at two points in time, and with any luck at all the explanation above will make sense in the context of the figure.

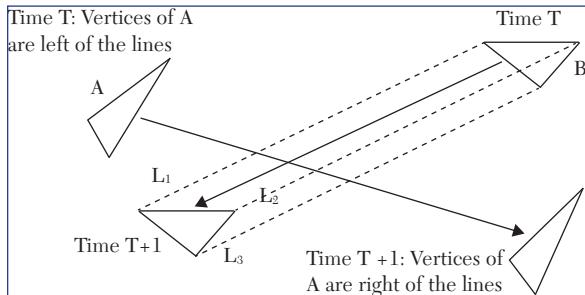


FIGURE 6.3 Polygons **A** and **B** move in the direction of the arrows between times **T** and **T+1**. If any of the vertices of polygon **A** are on one side of any of the lines **L₁**, **L₂**, or **L₃** at time **T** and are on the opposite side of the same line at time **T+1** then a collision is possible.

Basically, if any of the vertices of polygon **A** cross any of the lines **L₁** to **L₃** then a collision *may* have taken place. This is the same basic method we used for the 1D situation, and has a similar solution.

The line called **L₁** in Figure 6.3 is crossed by the triangle **A** if any vertex is on one side of the line at time **T** and on the other side at **T+1**. Every line has an equation that defines it, and this can take one of many mathematical forms. Let's use the *standard normal form*

$$ax + by + c = 0$$

A point (x,y) that satisfies this equation is *on the line*, and there are two other possibilities: $\mathbf{ax + by + c}$ is greater than zero, in which case the point (x,y) is *above* the line, and $\mathbf{ax + by + c}$ is less than zero, meaning that the point is below the line. Thus, if any vertex is above \mathbf{L}_1 at time \mathbf{T} and below it at $\mathbf{T + 1}$ then there could be a collision, otherwise there cannot be.

This is an example of *broad phase* collision detection, the elimination of objects that cannot possibly collide. The idea is to do this quickly, without using up too much CPU time. The next step, or the *narrow phase*, is about accurately detecting collision events and determining the time and position of the collision. Breaking the problem into these two parts is about performance. The idea is to do the detailed and time consuming narrow phase only if there is a chance the objects could collide in the first place. This has the effect of making the whole process more flexible, too. There are many ways to solve each problem, and the solutions that are decided upon can be combined almost arbitrarily.

Let's look at each phase independently, making sure that there are ways to combine them. After that, there are single phase collision detection methods, and we'll look at one or two of those. We'll also jump to 3D now, since we have covered the basics and games take place almost exclusively in 3D these days.

Broad Phase Collision Detection

It is in this phase where the greatest saving of time can be created by carefully selecting an algorithm and implementing it efficiently. This phase is about rejecting objects that cannot collide, eliminating collision tests that cost time but cannot yield fruit.

Operational Methods

The phrase operational has been used in a similar context with respect to security. It represents an examination of how the system operates at a general, perhaps even superficial level, to see if efficiencies can be created or obvious flaws found. For example, in operational security they may determine that a computer operator's screen can be seen through a window, and so the first thing to do in improving system security is to move the monitor away from the window.

In a game, there are many objects that can be colliding or collided with, and the general collision problem takes an amount of time in proportion to

the square of the number of objects (which a computer scientist would call $O(n^2)$). If there are four cars, a hundred trees, and twenty buildings, that is 124 objects, and up to 15,376 collision tests.

However, testing A for collision with B is the same as testing B for collision with A, and we never need to test A for collision with A. This reduces to a total of $(N^2 - N)/2$ tests, which is a lot fewer than N^2 . This gives us 7,626 tests for the above example.

Now consider static objects such as trees and buildings that cannot collide with one another. Only the cars need to be tested against all other objects. This represents $(4 \times 3)/2$ tests between cars (=6) and $4 \times 120 = 480$ tests with static objects, for a grand total of 486 tests. Any further efforts at broad phase detection will almost certainly not yield such significant savings but could still find more savings.

Most games have many classes of objects, and computation time can be reduced significantly this way. Alien missiles, for example, need not be checked against alien spacecraft, and sometimes asteroids or meteors need not be checked against each other. It's not as exciting as advanced code optimization, but a little good sense gives a much greater benefit than almost anything else we can do.

You should always look very carefully to make certain that you are not making any unnecessary tests. Then, you do the more difficult things, and combine the two methods for a joint time savings.

Geometric Tests

When we checked to see which side of the line the start and end point of a polygon vertex was on at times T and T+1, we were using geometry to eliminate possible collisions. This is a pretty efficient process, and we can do it in three dimensions too. In 3D, a line becomes a plane, and the test becomes, *is a particular polygon or node on one side of a plane at time T and on the other at time T+1?* If so, a collision is possible. If not, one has been eliminated.

In more detail, let us imagine that two objects in our game are to be tested to see if they collide with each other. We know that we're going to draw objects as collections of polygons, since that's how modeling programs define them, and it's how graphics cards draw things.

So the question is, *do any of the polygons in the set A (object A) collide with any of those in set B (object B)?* Each polygon is, in fact, a part of a

plane. A mathematical plane is infinite in extent and a polygon is not, but in the broad phase we extend a polygon in one object to become the plane in which it is embedded and ask whether any of the vertices or polygons in another object are on opposite sides at times **T** and **T+1**. If so, a collision is possible, otherwise one has been eliminated.

This can be done quickly, in a manner similar to that already described in the section above of 2D collisions. Two triangles A and B have vertices A0, A1, A2 and B0, B1, and B2. Both belong to different moving objects, and the question of collision is at issue. Instead of just plugging coordinates into the equation of a plane, it is possible to use vector math to do the equivalent thing, but in a manner that is faster.

Imagine a two-dimensional triangle floating in 3D space. The 3D orientation of the triangle is minimally defined by the three corners. A line can pass through the triangle from any angle. When the intersection of the line and the triangle is exactly 90 degrees, it is defined as normal to the triangle. With a normal relationship between a line (vector) and a triangle (polygon), we can calculate the equation that defines the plane on which the triangle exists.

The final chapter of this book contains a summary of the math we need and a primer on key subjects. It also provides basic implementations of the important operations. It turns out that a simple-to-calculate operation known as the *cross product* takes any two vectors and creates a vector that is perpendicular, or normal, to the plane on which the two input vectors lie. For two vectors **h** and **j**, the cross product **h × j** is a vector:

$$h \times j = \begin{bmatrix} h_y j_z - h_z j_y \\ -(h_x j_z - h_z j_x) \\ h_x j_y - h_y j_x \end{bmatrix}$$

If we have a triangle (or a quad) then we have two such vectors—any edges will do.

The *dot product*, another simple operation on vectors, computes the length of the projection of one vector onto another; for vector **a**, **a·a** = the length of **a**, and if **a** and **b** are perpendicular then **a·b** = **0**. The dot product is defined as follows, in n dimensions, and it looks like a simple distance calculation:

$$a \bullet b = \sum_{i=1}^n a_i b_i$$

The usual form of the equation of a plane is:

$$C_0x + C_1y + C_2z = d$$

Going back to the triangles A and B, note that each vertex of **A** (i.e. \mathbf{A}_i) has three coordinates, and is therefore a 3D vector. Thus, $\mathbf{A}_0 \times \mathbf{A}_1 = \mathbf{N}$ is a vector cross product that is a normal to the triangle (plane) **A**. The dot product between \mathbf{N} and any point in the plane of A, say any vertex, is the equation of the plane. So, $\mathbf{A}_2 \cdot \mathbf{N}$ is:

$$d = N_x A_{2x} + N_y A_{2y} + N_z A_{2z}$$

This has the same form as the equation of plane. That's because it is a plane and it is specifically the equation of the plane that the triangle **A** lies in. This is what we want. We can store the plane as **N** and **d**, which are really just the constants for the equation of the plane, and we do this for every polygon we may want to test for collision. Testing is a matter of plugging in the (**x,y,z**) values of the target (i.e. what we are testing for collision, like another polygon vertex) before and after motion, and seeing if the target is on opposite sides of the plane. If so, we need to look further, and if not we can ignore this target.

Using Enclosing Spheres

Two objects, let's say cars, may consist of thousands of polygons each. If one of them is *Enzo*'s car, and he is clever enough to pass yours in our race, it is possible that many of the polygons in his car will be candidates for collision with polygons in your car. This could cost a lot of time in detailed tests that are not needed. One way to avoid this is to process collisions at a higher level at first, looking at all of the polygons in your car as belonging to a single object that has a virtual shape, such as a sphere. So, if the sphere that encloses *Enzo*'s car never intersects with the sphere that encloses yours then they can't have collided. We could save thousands of polygon-polygon collision tests.

Better yet, the enclosing sphere can be defined along with the car, or whatever object we're looking at, once, when the game is created. All we need to do is find the center of the sphere, which would be the *centroid* of the object, and the radius, the distance to the most distant vertex. The centroid of the car will be what we actually move—the car is drawn on the screen relative to that point, and the collision tests are initially performed on the sphere centered on that point too.

The centroid, or center of mass, can be approximated by finding the mean coordinate of all polygons in each dimension. In 3D, the X coordinate of the center of mass is the sum of all polygon X coordinates for the object divided by the number of polygons, and in the same way the Y and Z coordinate of the centroid are computed. The distance to the most distant vertex can be calculated at the same time as we calculate the centroid, which is to say at some time after the object is created but before the game is distributed.

Now each object has a center of mass and a radius associated with it, in addition to a big set of polygons. The question that faces us now is, *for each pair of objects that could possibly collide, is it possible for the enclosing spheres to collide?* If not, we can ignore all of the polygons that are part of those objects, and that will save us a large amount of time. Of course, if it is possible then we have a lot more checking to do. Sometimes it is more useful to keep the object as polygons, as in the case of walls and buildings. Covering them with a sphere would be a waste of time, because it is easier to keep their relatively few planar faces as planes.

Sphere versus Plane Collision

The first step is to see if the sphere changes sides of the plane during the time interval. The situation is diagrammed in Figure 6.4: the sphere at time T is labelled S_T , and its center $\mathbf{S} = (S_x, S_y)$ is a distance D_T from the plane and the positive side. At time $T+1$ the center of the sphere is at D_{T+1} , and in the figure is on the opposite (negative) side of the plane. We must make sure that $D_T > r$, or the sphere is intersecting the plane at the beginning.

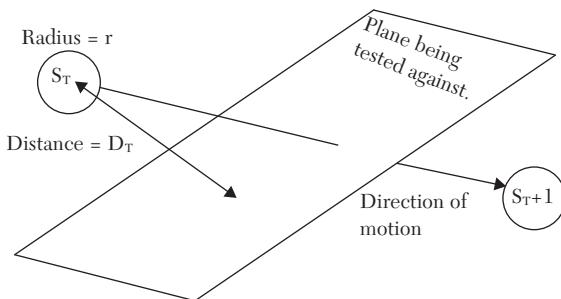


FIGURE 6.4 Given the position of the sphere at time T and $T+1$, did the sphere pass through the plane?

A sphere is not a point; it has volume. For this reason, the test concerning whether a sphere may have collided with a polygon is a little different from before: if $D_T > r$, where r is the sphere's radius, and $D_{T+1} < r$ then a collision is possible. A simple way of computing this is as follows for some

point on the plane **Z** and plane normal **N**:

$$\mathbf{d}_T = (\mathbf{S}_T - \mathbf{Z}) \cdot \mathbf{N}$$

If a collision with the plane occurs, when does it happen? We know it is sometimes between **T** and **T+1**, but *exactly* when? We can parameterize time between **T** and **T+1** to be one unit using a new variable τ , where t is between 0 and 1:

$$\tau = \frac{d_T - r}{d_T - d_{T+1}}$$

This ratio is the fraction of the distance difference between **T** and **T+1** that is represented by the sphere's radius, and is the fraction the distance travelled in that time by the sphere. So, the location of the center of the sphere at the time of the collision is as follows, where S_{xT} is the X coordinate of the center of the sphere at time **T**, and similarly for the **Y** and **Z** coordinates and time **T+1**:

$$\begin{aligned} x &= S_{xT} + \tau(S_{xT+1} - S_{xT}) \\ y &= S_{yT} + \tau(S_{yT+1} - S_{yT}) \\ z &= S_{zT} + \tau(S_{zT+1} - S_{zT}) \end{aligned}$$

where, as a reminder, S_{xT} is the X coordinate of the center of the sphere at time **T**, and similarly for the **Y** and **Z** coordinates and time **T+1**.

Sphere versus Sphere Collisions: Part 1

When checking two cars for collision, one could be enclosed by a sphere and the other could be considered to be polygons. Each polygon would be a plane for the purposes of the collision test. On the other hand, it would be better to construct a single plane between the two cars, which would both be considered spheres. A single test could then determine whether a collision was possible.

The definition of a plane requires three points, or a point and a surface normal. One point would logically be the midpoint between the two spheres, \mathbf{P}_m , found by averaging the coordinates of the sphere centers. A better point would be the midpoint \mathbf{P}_0 between the two sphere *surfaces*, but that is harder to find and may not be much better.

Still, consider that the distance between the surfaces of the spheres is $\mathbf{P}_m - \mathbf{R}_0 - \mathbf{R}_1$, with \mathbf{R}_0 being the radius of \mathbf{S}_0 , and so on. The length of \mathbf{P}_m

is $\|\mathbf{S}_0 - \mathbf{S}_1\|$, and so the fraction of this length represented by the radius of S_0 is $R_0/\|\mathbf{S}_0 - \mathbf{S}_1\|$, and similarly for S_1 . We can use this to find the points of the sphere's surfaces along the line that joins the centers. The midpoint between those two points is what we want.

The total distance between the centers in the **X** coordinate is $\mathbf{S}_{0x} - \mathbf{S}_{1x}$, so the distance along **X** to the surface of the sphere is $(R_0/\|\mathbf{S}_0 - \mathbf{S}_1\|) * (\mathbf{S}_{0x} - \mathbf{S}_{1x})$. It would be similar in the **Y** and **Z** coordinate, and so we now have the 3D coordinates of the points on the spheres nearest to each other - now simply average these.

Now we need two more points or the normal to the plane. Determining the normal is quite straightforward—it is the vector between \mathbf{P}_0 in the \mathbf{S}_1 direction, or from \mathbf{P}_0 to \mathbf{S}_0 . Let's use $\mathbf{N} = \mathbf{S}_1 - \mathbf{P}_0$ as the normal. The equation of the plane is then:

$$N_x(x - P_{0x}) + N_y(y - P_{0y}) + N_z(z - P_{0z}) = 0$$

Or, if you substitute for the normal vector coordinates:

$$(S_{1x} - P_{0x})(x - P_{0x}) + (S_{1y} - P_{0y})(y - P_{0y}) + (S_{1z} - P_{0z})(z - P_{0z}) = 0$$

Any points on this plane will be equidistant from \mathbf{S}_0 and \mathbf{S}_1 . The points \mathbf{P}_1 and \mathbf{P}_2 are two such points at random on the plane in Figure 6.5. These are points that are equidistant from the sphere centers at disparate angles. We can check this by computing the distance between $\mathbf{P}_1 - \mathbf{S}_0$ and $\mathbf{P}_1 - \mathbf{S}_1$; the two distances must be equal.

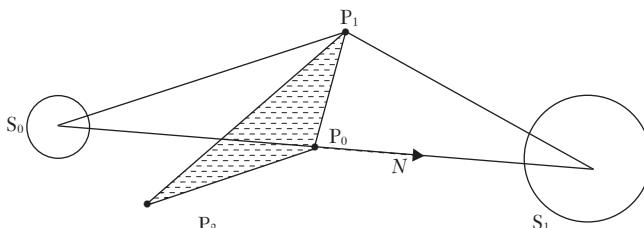


FIGURE 6.5 The geometry of two spheres separated by a plane. The point P_0 is in the middle of the spheres, and a normal N (along the $s_0 - s_1$ line) to the plane allows the plane to be defined completely.

Summary

This discussion has been a trifle complex, and sometimes it can be hard to focus on the big problem when there are lots of details to take care of. So, a summary of the use of spheres for collision detection is:

1. Calculate the center of mass \mathbf{C} for each moving object (i.e. the average coordinate in each dimension) and store it with the object. Also calculate the maximum distance from the center of mass and every polygon vertex \mathbf{V}_n . The distance between \mathbf{C} and \mathbf{V}_n is the radius of the enclosing sphere, and needs never be computed again.
2. When collision detection needs to be done, look at each pair of spheres in turn. Compute the equation of the plane that lies in between the spheres according to the method above, so that the equation

$$(S_{1x} - P_{0x})(x - P_{0x}) + (S_{1y} - P_{0y})(y - P_{0y}) + (S_{1z} - P_{0z})(z - P_{0z}) = 0$$

represents that plane.

3. Compute the distance between the sphere and the plane for both spheres, both before and after the current move. If either sphere crosses the plane (i.e. the sign on the distance changes) then a collision is possible.

This encloses complex objects by a simple sphere and reduces the number of tests that need to be done by thousands to one, in most cases. Also, the calculation to see if the collision is possible is now reduced to a few additions and multiplies for each object.

Naturally, if a collision is possible then a more time consuming test has to be performed to see if a collision actually happens.

Sphere - Sphere Collisions: Part 2

The previous discussion has been based on the test against a line/plane that we developed at the beginning. There are problems with this, especially if both spheres are moving. For example, if the spheres are moving so that one is faster than the other, and is actually moving away, then one sphere could pass through the plane without actually being able to catch up with the other and collide with it. What we have done works fine if one sphere is still, so that perhaps it represents a complex object but not another vehicle. It is quick, however.

One thing that could help is to see if the spheres are far enough apart so that they could not possibly get close enough to collide during the specified time interval. In other words, the *net* movement vector between the two spheres must be longer than the distance between them, minus the radii, of course. A movement vector is the vector between the position of the sphere

center before and after the movement, and can be easily computed from the velocity vector, current position, and time duration. If both spheres are moving, we need to compute the *relative* movement, or that movement that would be seen from one of the spheres. This amounts to subtracting one movement vector from the other. Now, if this relative movement vector is less than (shorter than) the distance between the sphere centers minus the sum of the radii, then there is no way that they could collide. This is a very simple test that rejects a collision.

There are more collision rejection tests on spheres. If the two spheres are moving away from each other then they cannot collide. How can we tell if this is so? We first create a vector that points from the center of sphere S_0 to the center of sphere S_1 ; call this vector N_0 , because it points in the same direction as the vector N in Figure 6.5. Quite specifically, let:

$$N_0 = S_1 - S_0$$

The next step is to project this vector onto the movement vector for S_0 . This is done using the vector dot product we have used before. If the movement vector is M , then $q = M \cdot N_0$ is the dot product of M and N_0 . If $q <= 0$ then the sphere S_0 is not moving toward S_1 , and there can be no collision. As before, if both spheres are moving then we use the relative movement vector.

At this point, we know that the sphere S_0 is moving far enough to collide with S_1 , and that it is moving in the right direction so that it could do so. For simplicity, assume that S_1 is not moving, realizing that we can make it so by transforming coordinates. The final question is, *does S_0 get closer to S_1 than the distance R_0+R_1 ?*

If not, then they cannot collide. Computing this last value is a little bit tricky. Figure 6.6 represents an effort to describe this situation better.

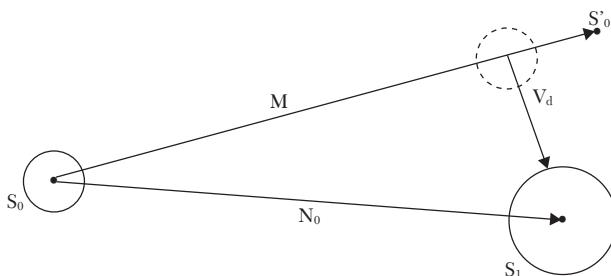


FIGURE 6.6 Geometry of moving spheres and collision determination.

In this figure, \mathbf{M} is the movement vector and \mathbf{N}_0 is the vector joining \mathbf{S}_0 and \mathbf{S}_1 , both as defined previously. We wish to determine the point on the vector \mathbf{M} , if we treat it as a line, that is nearest to the center of \mathbf{S}_1 . The following formula is from MathWorld:

$$d^2 = \frac{((\mathbf{S}'_0 - \mathbf{S}_0) \times (\mathbf{S}_0 - \mathbf{S}_1))^2}{|\mathbf{S}'_0 - \mathbf{S}_0|^2}$$

I have substituted our variables for theirs in this equation. The \mathbf{d}^2 value is the smallest distance (squared) between the line from \mathbf{S}_0 to \mathbf{S}'_0 and the point \mathbf{S}_1 , the center of the sphere \mathbf{S}_1 . If this value is greater than $\mathbf{R}_0 + \mathbf{R}_1$ then the spheres cannot collide. Note that the \mathbf{x} in the formula is a vector cross product, not a multiplication. All variables in the above except d are vectors.

If the problem involves finding the distance between the line $\mathbf{y} = \mathbf{mx} + \mathbf{c}$ and the point (\mathbf{X}, \mathbf{Y}) then a more convenient formula is:

$$d = \sqrt{\left(\frac{x + my - mc}{m^2 + 1} - x\right)^2 + \left(m\frac{x + my - mc}{m^2 + 1} + c - y\right)^2}$$

If none of the tests above fail, then a collision between the two spheres could happen.

Using Bounding Boxes

A bounding box is a rectangle or prism that completely encloses all of the polygons of an object. The sides are planes, and the volume is relatively close to that of the object—generally a better approximation than is a sphere. An *Axis Aligned Bounding Box* (AABB) has faces that are parallel to the three coordinate axes. This has the advantage of being very simple to calculate: run a plane through the most distant point in each coordinate axis direction. That is, find the minimum and maximum X coordinate in the object; construct a plane though these points that is parallel to the YZ plane; do this for the Y coordinate (XZ plane) and the Z coordinate (XY plane) thus building a rectangular prism. This is the AABB.

There is no complex calculation needed to find the AABB. We simply scan all of the polygons for the object and note the minimum and maximum value found in each dimension—six values in all. Of course, this must be done using the polygon coordinates in the *world domain*—that is, as drawn in place in the scene. Once we have minimum and maximum values, it is a simple task to use these to detect whether a given point is inside or outside

of the box, and whether two boxes are overlapping. A point is inside the AABB if its coordinates are greater than the minimum and less than the maximum in each dimension. Two AABBs overlap if any of the vertices of either one lies inside of the other. Another, similar, test of overlap is: two AABBs overlap if their extents overlap in each of the axes. This is quite fast and simple to compute.

This is, of course, not good enough. The two boxes may pass completely through each other and not touch either at the beginning or the end.

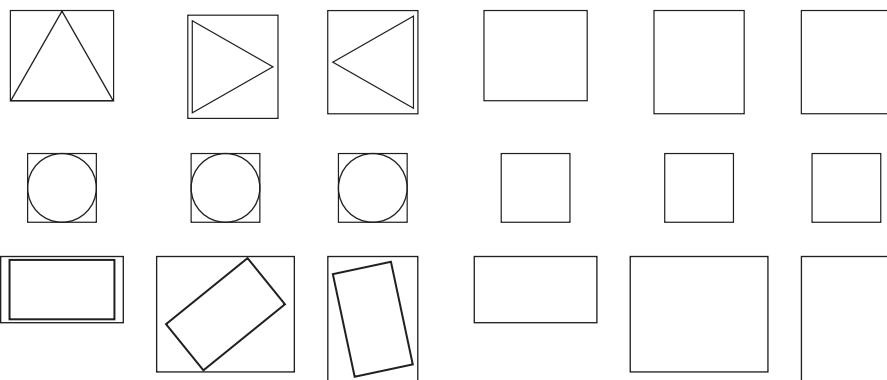


FIGURE 6.7 Axis-oriented bounding boxes for three simple shapes in three orientations. On the right you can see the boxes alone and compare their sizes.

To determine if a collision has occurred in between the start and end of the time interval we need to first consider a one-dimensional problem. A moving 1D box can be represented as two real numbers, s (start) and e (end) that describe the position of the box at two times. The set of all intervals can be represented as a list **L** of (s_i, e_i) values. It would be best to keep this list sorted in ascending order. What we need to do is find all values of (s_i, e_i) and (s_j, e_j) that overlap.

We create a fresh list, initially empty, that will contain entries for all objects currently active—call this the *active list*. As we scan the sorted list of intervals **L**, a new s_i being encountered results in the active list being output as a potential collision in interval i , and the interval i is added to the active list. When a new e_i is seen while scanning **L**, interval i is removed from the active list.

The great thing about AABBs is this: to expand this to the 3D case, we simply have lists for each dimension. If all three dimensions report an intersection between AABB_i and AABB_j , they intersect, and a collision may have occurred.

A new AABB must be determined each time the object changes direction. This is an expense not incurred by using spheres, but we shall see if there is a compensating trade-off. Another, more minor, problem is that the AABB can sometimes be a poor fit to the object. Consider a triangle, circle, and rectangle as in Figure 6.7. As the objects rotate the AABBs fit more or less well in the box. The closer the fit, the more accurately the collision between boxes will predict an actual collision.

Object-Oriented Bounding Boxes

An object-oriented bounding box (OOBB) is always aligned along the primary axes of the object. This box is defined when the object is first created, and is read in along with the polygon coordinates or computed as it is read in. The box is translated and rotated along with the object, and so it should be clear that the edges of the OOBB will not necessarily align with any axis.

These hug an object better than an AABB or a sphere, in general, as seen in Figure 6.8.

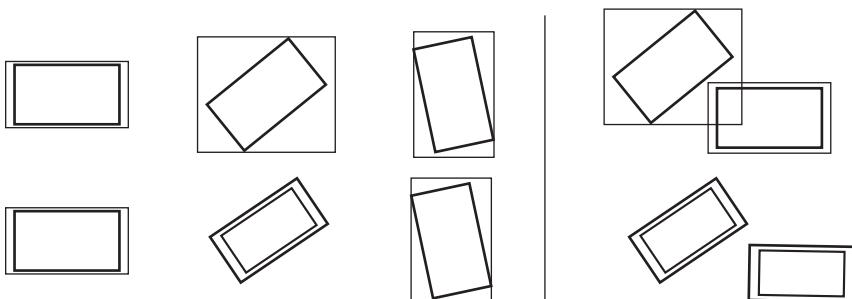


FIGURE 6.8 Object-oriented bounding boxes. (Left) As a rectangle is rotated, the bounding box has more empty space. Orienting the bounding box to the object's orientation reduces the error. (Right) An axis-oriented box indicates a collision is possible where an object-oriented box does not.

This means that a collision can be more accurately determined. On the other hand, it takes a lot more code and time to determine whether two OOBBs collide or not.

We'll not going into details on OOBB collision detection (See references 4, 5), but there is an interesting thing here. It is true that a new AABB must be computed each time the object changes direction, and this can be time consuming. However, you can construct the OOBB at the outset and rotate and translate the box with the object.

Space Subdivision

The process of looking at every pair of objects to see if a collision has happened is slow, but if the objects are spread out over a large area we can break up the entire playing volume into small pieces, each large enough so that moving objects won't pass completely through one, but small enough to contain only a few objects that could possibly collide. The basic idea is to check for collisions only among objects that are within the same piece of space.

The data structure is simple, like the concept. Space is divided into equal sized blocks, and an array can be used to represent each. A block contains a list of objects that reside in it, and at the beginning of each frame these are all cleared out. The list of objects is run through, and each object is placed into the appropriate list. The blocks that contain objects also have a count of how many objects are in that block. Finally, we look though the list of all blocks for those with more than one object, and those are tested for collisions.

If a game has a lot of 3D interactions, this method could require a lot of storage. A game mostly takes place on the ground, on a relatively flat surface, which is what we could call $2\frac{1}{2}$ dimensional. By this it is meant that we will consider squares on the terrain surface and a small volume above, and ignore the rest of the playing volume, which can't be driven on.

There are a variety of uses and implementations of the block map method, but in the context of a typical game it can be restricted in the following ways:

1. An object will be in one block, as a general rule. This will be the block in which resides the center of mass, or the center of the enclosing sphere.
2. The size of the block will be such that the object, and any other object with which it can collide, cannot pass through a whole block in one time interval. This means we only have to check the starting block and the ending block in the worst case, and in most cases the object will stay in one block.
3. We want a reasonable number of blocks, so they must be big enough that there are fewer blocks than objects.
4. There must not be more than a few (4 to 5 at most) objects in any block.

So, let's say that the playing area is 2×2 Kilometers. Let the maximum speed of any object be a highly reasonable 100 Km/hr. At 24 frames per second (FPS) a frame takes $1/24$ second, or about 42 milliseconds. 100 Km/hr is 28 meters/second, or 0.28 meters per millisecond, or 1.176 meters/frame. The rules above give us the following limits:

- 1.** The block should be at least the size of the object. A car is rarely more than 3 meters long, and a person is only 1 meter. Let's say the block must be at least 3M.
- 2.** In one frame, an object will pass through 1.176M, so 3M is large enough that it cannot pass through a block in one frame at 24 FPS. If we get to 60 FPS we'll have to make it 2.94M. 3M is still big enough.
- 3.** There will be about 250 objects in the game at most. 3M is too small—at this size there will be over 100,000 blocks. At 250 objects, the block size will be about 125M.
- 4.** At a block size of 125M, we still want there to be fewer than 4 to 5 objects per block that can be interacted with. In some games an opponent will sometimes be very close. If we are careful about how we place objects, a block size of 100M should be fine. This size gives us a little less room, and lets us put objects closer together.

At a block size of 100×100 meters, a fast object can pass though in about 4 seconds, but objects can be placed far enough apart that it takes a while to get to one. How many blocks are there then? 10 blocks per Km is 20×20 blocks, or 400 blocks all together. This is a bit large but small enough to be reasonable to search. Most, after all, will be empty.

Another advantage of using a block map is that only the moving objects need to be updated every frame. Everything else is placed in its block at the beginning of the game and it will remain there. A moving object can be quickly placed in a block by mapping the coordinates of the object center onto block map grid coordinates. For the 2×2 Km map above, we first map the object's (x , y , z) coordinates onto block indices. An easy way: X coordinate in Kilometers/100 = column index = **J** and ditto for the Y coordinate, which we will call **I**.

Each block is represented by a structure stored in a 2D array, indexed by (**I**, **J**). Each structure contains the number of objects in that block and indicators that allow access to each object. These could be pointers or

indices to an array of objects. If a complex structure is used to store them, it would be no worse than a simple linear list. Let's have a maximum of six objects per block, and have a fixed size array of indices of objects within each block structure. To speed things up even more, create a global list of all blocks containing more than one object. This, too, could be a fixed size array. Whenever an object moves, we check to see if it stays in the same block. If not, decrease the object count for the old block, increase the count for the new one. Remove the old block from the global list if it has less than two objects, and add the new one if it has more than one.

Before starting collision detection, add the blocks at which objects will end up after motion to the global list. Next, looking only at the global list, check for collisions between objects within every block in that list. Finally update the global list to remove the blocks no longer having more than one object.

Narrow Phase Collision Detection

Once we determine that a collision is possible, the next stage answers, *does a collision occur?* If one does, at what point on the surface, which polygon on the model? This question can rarely be answered as fast as we'd like, never as fast as the broad phase question. And, as always, the more accurately we need the answer, the more expensive it will be. The hope is that this detailed and expensive calculation will not have to be done very often.

There is a wide variety of narrow phase algorithms, but we will only cover the most obvious methods. When a narrow phase algorithm is invoked, it is because a collision could occur between two objects that consist of polygons. These objects were enclosed by spheres or boxes, but now we must look at the details. Let's assume that the polygons are triangles—each has three vertices—and if the object consists of 1000 triangles then there will be 300 vertices, right?

No, because in an object most of the vertices are shared. An estimate would be 1000 *distinct* vertices in this object. Each vertex will move in a known direction by a known amount, the so-called movement vector. This means that there are 1000 *rays*, or directed line segments, that we need to examine. It can be assumed that the second object is still, because if it were moving then we'd subtract its movement vector from both objects to give a net movement vector on the object being tested, as described above where we checked for collisions using spheres.

So, we have 1000 rays, and a similar number of polygons in the other

object we are testing against. This would be a million tests, each ray against each polygon. It's possible to eliminate some rays and some polygons, though. Only polygons that are on the side of the object facing the other object need to be tested, so we have perhaps 500×500 tests. This is still a significant number, but 1/4 of the previous value.

Ignoring the back facing polygons can be done using *back-face culling*. (Foley, et al. 1996) Each polygon (triangle) has a normal associated with it, or we could compute one every time we need it. Now, back-face culling is really a visibility algorithm. The question is, *can you see that polygon from where you are now?* We use the dot product between the normal to the triangle and a vector from the viewer's position to get the angle between these vectors. If it is between 90° and 270° then the polygon is facing the viewer, otherwise it is not and can be ignored. In this case, replace the viewer with the centroid of the object being tested against, and the method is the same. Of course this must be done from both objects involved in the collision, each taking turns being the viewpoint. There are tricky ways to speed this method up, too. (Kumar, et al. 1996)

Ray/Triangle Intersection

The meat of the collision test is determining which polygons intersect, where, and when. From the previous discussion, we have selected some polygons (perhaps all of them) to test against each other, we have determined what the movement vector is, and we know that object **A** is moving while **B** remains still, or at least has been made still by computing a relative movement vector. Now we will select vertices in **A** and determine their positions before and after movement, then see if that line segment or *ray* intersects a polygon in **B**. If so, it is simple to determine when and where this happens. So, step by step here is what we must do.

1. The movement vector expresses relative movement, so by adding it to a point we find where the point moves. Vertices in **A** will be named \mathbf{V}_A . They are numbered from 0 to n, so they are \mathbf{V}_{A0} to \mathbf{V}_{An} ; finally they have **x**, **y**, and **z** coordinates named \mathbf{V}_{A0x} , \mathbf{V}_{A0y} , \mathbf{V}_{A0z} , and so on. The point \mathbf{V}' is the same point as \mathbf{V} , but is a position after the motion is complete.

The ray associated with a vertex \mathbf{V}_{Ai} , given the movement vector **M** would be a vector **R** from \mathbf{V}_{Ai} to $\mathbf{V}_{Ai} + \mathbf{M}$. This is to say:

$$R_x = V_{Aix} + M_x$$

$$R_y = V_{Aiy} + M_y$$

$$R_z = V_{Aiz} + M_z$$

The line that needs to be tested runs from \mathbf{V}_{Ai} to \mathbf{R} . Call this line \mathbf{L} , and it is:

$$\begin{aligned} L_x &= V_{Aix} + (tR_x) \\ L_y &= V_{Aiy} + (tR_y) \\ L_z &= V_{Aiz} + (tR_z) \end{aligned}$$

t runs from 0 to 1 to give any point on the line segment.

2. Now compute the equation of the plane in which the selected triangle in object B resides. This equation is that of 5.8 or 5.9—you will recall that we need the normal \mathbf{N} and a point in the plane \mathbf{P} to get the plane

$$N_x(x - P_{0x}) + N_y(y - P_{0y}) + N_z(z - P_{0z}) = 0$$

3. Substitute the line \mathbf{L} into the plane equation to solve for t , the time at which the collision will occur.
4. Use the value of t found in 3 to plug into Equation 5.13 to find the point (x, y, z) at which the collision will occur.
5. Finally, determine whether the point found in 4 above resides inside the triangle. This can be done in a few simple ways—the interior angle test, the odd angle test, or the area test.

The Interior Angle Test

Compute the angle between the point and all three triangle vertices, in order. The sum of the angles should be 360° , within rounding error, if the point is inside the triangle.

The Odd Intersections Test

Draw a line from the point being tested to a faraway point. If this line intersects exactly one edge exactly once, then it is inside the triangle. If the intersection is a vertex, though some bad luck, then select a new direction and draw another line—a vertex is part of two edges, and can't be used in this test.

The Area Test

Make all triangles between the point being tested and consecutive points on the triangle—there are three. If the sum of the areas of these

triangles equals the area of the big triangle, the point is inside. This test is approximate, partly again due to vagaries of floating point arithmetic.

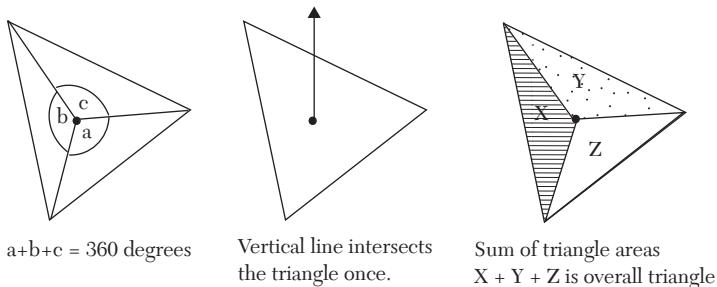


FIGURE 6.9 The three tests for determining whether a point is inside a polygon.

These three tests are illustrated in Figure 6.9.

All five steps above can be accomplished in a remarkably small time period, and is actually done in many games. You can implement any or all of the collision detection schemes that have described using what has been said here, combined with a few details found on the Web.

Or, you could use someone else's code.

Collision Detection Packages

Given the complexity of collision detection, the variety of methods possible, the relationships between the broad and narrow phase, and the difficulty of implementing some of the methods, perhaps you should use a package that is known to work correctly. There are quite a few that can be downloaded from the Internet in a few minutes, and almost any of them will save you a huge amount of time.

You should understand that, while it is possible to use this basic description of collision detection to implement your own system, you may be better off using someone else's—everyone has a favorite part of the game development process, one they enjoy more than other aspects. For you it may be graphics. For some it is collision detection, and those people have fortunately shared with you on the Internet.

Vclip

This system works for convex polyhedra, and is coded in C++ and Java, originally in 1997 (by Brian Mirtich).

The problem is that cars and trees and such are not convex. Good thing that the system includes a program for computing 3D convex hulls. However, as a result, the collision detection is approximate for non-convex objects.

<http://www.cs.ubc.ca/spider/lloyd/java/vclip.html>

ColDet

This is an interesting system that seems to work with what is called a *polygon soup*, a random appearing set of polygons, not convex, not necessarily even connected. Another good thing—it compiles on the GNU C++ compiler or Visual C++, and others. It uses a hierarchy of bounding boxes, and can achieve a very high degree of accuracy.

<http://photoneffect.com/coldet/>.

I-Collide

This claims to give exact collision results for complex polyhedra. It was first released in 1996, but the most recent version was released in 2004. However, the authors suggest that you use the code immediately below.

http://www.cs.unc.edu/~geom/I_COLLIDE/index.html.

Swift

This library is a C++ system that works on convex polyhedra or objects that are expressed as a hierarchy of convex polyhedra. It is faster than *VClip* or *I-Collide*, but is also more difficult to use.

<http://www.cs.unc.edu/~geom/SWIFT/>

Rapid

Like *ColDet*, this system works on polygon soups, the most general class of problem. It has a relatively simple interface and is coded in C++. This may be the easiest system to use with *OpenGL*, although this is clearly a matter of opinion. It uses a structure called an OBB tree (hierarchy of oriented bounding boxes) to achieve high speed yet general collision detection.

<http://www.cs.unc.edu/~geom/OBB/OBBT.html>

Summary

Collision detection is a complex activity, one that can involve a lot of mathematics. It can be broken into a broad phase, where collisions that

could happen are detected, and the possible collisions are examined more carefully in the narrow phase, ignoring objects that cannot collide.

In collision detection, the point is to determine the time and location of the collision. Collisions rarely happen exactly at a specific frame, so the trick is to determine when it happened and where the objects will be when the next frame is drawn.

In many cases, and for processing games very often, collision detection can be done using spheres that enclose the objects—if the spheres collide then the objects collide, and otherwise a collision did not occur. This is an approximation, but one that is useful in a large percentage of games.

Exercises

The problems below will apply your knowledge of the material in this chapter, and sometimes require that you do some more research before you are able to complete them. Solutions to selected exercises are available on the companion disc.

1. In sphere-sphere collision detection the distance between the two spheres is calculated and if less than the sum of the radii then a collision is in progress. Simple code would be as follows, where (x_0, y_0, z_0) and (x_1, y_1, z_1) are the coordinates of the sphere centers:

```
d = sqrt((x1-x0)*(x1-x0)+(y1-y0)*(y1-y0)+(z1-z0)*(z1-z0))
if (d < (radius1+radius)) // Collision
```

A square root calculation is expensive—what is the code that does this without a call to **sqrt**?

2. Code a sketch that has two balls (circles) bouncing in a box. Use the code in Exercise 1 to determine when the balls collide with each other, and have them react to the collision.
3. Write a sketch that will check for a collision between two moving cubes in a 3D space.
4. Use the code in Exercise 2 to create a stack of four cubes and hurl a fifth cube at the stack. The sketch should detect collisions between the cubes and have them respond. Approximate bounces are OK.
5. Some games, like snooker, are all about collisions. Write a sketch that allows a white (cue) ball to be shot in any direction on a table and collide

with one of a set of (at least) two other balls. The collisions should result in correct seeming bounces. This is a 2D problem.

IDEA *When the mouse is pressed, draw a line from the cue ball to the mouse coordinates. When released, the cue ball will follow the line at a fixed speed until it collides with a cushion or another ball.*

- Using a circle for a ball and a line for a bat, construct a bat and ball simulation. The bat will be rotating, not moving linearly, and the ball will move toward the bat. You need only determine the point of collision and time, not determine the line along which the ball will move.

NOTE *The point to line distance is key here, but the line is a segment and a bounds test is important.*

Resources

ColDet: <http://photoneffect.com/coldet/>

I-Collide: http://www.cs.unc.edu/~geom/I_COLLIDE/index.html

RAPID collision detection: <http://www.cs.unc.edu/~geom/OBB/OBBT.html>

Swift collision detection: <http://www.cs.unc.edu/~geom/SWIFT/>

Vclip: <http://www.cs.ubc.ca/spider/lloyd/java/vclip.html>

Bibliography

Basch, Julien, Jeff Erickson, Leonidas J. Guibas, John Hershberger, and Li Zhang. “Kinetic collision detection between two simple polygons.” *Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*. Baltimore, MD: ACM-SIAM, 1999. 102–111.

Cohen, J. D., M. C. Lin, D. Manocha, and M. K. Ponamgi. “I-COLLIDE: an interactive and exact collision detection system for large-scale environments.” *Proceedings of the 1995 Symposium on Interactive 3D Graphics*. Monterey, CA: ACM, 1995. 189–196.

Erickson, Jeff, Leonidas J. Guibas, Jorge Stolfi, and Li Zhang. “Separation-sensitive collision detection for convex objects.” *Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*. Baltimore, MD: ACM-SIAM, 1999. 327–336.

- Foley, van Dam, Feiner, and Hughes. *Computer Graphics*. Addison-Wesley, 1996.
- Gilbert, E.G., D.W. Johnson, and S.S. Keerthi. “A fast procedure for computing the distance between complex objects in three dimensional space.” *IEEE Journal of Robotics and Automation* 4 (1988): 193–203.
- Gottschalk, S., M. Lin, and D. Manocha. “OBB-Tree: A Hierarchical Structure for Rapid Interference Detection.” *Proceedings of ACM Siggraph '96*. ACM, 1996.
- Gregory, A., M. Lin, S. Gottschalk, and R. Taylor. “H-Collide: A Framework for Fast and Accurate Collision Detection for Haptic Interaction.” *Proceedings of IEEE Virtual Reality Conference*. Houston, TX: IEEE, 1999. 38–45.
- Hudson, T., M. Lin, J. Cohen, S. Gottschalk, and D. Manocha. “V-COLLIDE: Accelerated Collision Detection for VRML.” *Proceedings of VRML'97*. 1997.
- Kumar, Subodh, Dinesh Manocha, Bill Garrett, and Ming Lin. “Hierarchical Back-Face Culling.” *7th Eurographics Workshop on Rendering*. Porto, Portugal: Maverick, 1996. 231–240.
- Lin, M., and J. Canny. “Efficient collision detection for animation.” *Third Eurographics Workshop on Animation and Simulation*. Cambridge, UK, 1992.
- Mirtich, Brian. “V-Clip: Fast and Robust Polyhedral Collision Detection.” *ACM Transactions on Graphics*, July 1998: 177–208.
- van den Bergen, Gino. *Collision Detection in Interactive 3D Environments*. Cambridge, MA: Morgan-Kaufman, Elsevier, 2004.
- van den Bergen, Gino. “A Fast and Robust GJK Implementation for Collision Detection of Convex Objects.” *Journal of Graphics Tools*, 4(2): 7-25 (1999), PostScript (255KB), PDF (79Kb).
- van den Heuvel, Joe, and Miles Jackson. *Pool Hall Lessons: fast, Accurate Collisions Between Circles or Spheres*. January 18, 2002. http://www.gamasutra.com/view/feature/3015/pool_hall_lessons_fast_accurate_.php (accessed 2014).
- Watt, A., and F. Policarpo. *3D Games: Real-time Rendering and Software Technology*. Addison-Wesley, 2001.
- Weisstein, Eric W. *Point-Line Distance--3-Dimensional*. n.d. <http://mathworld.wolfram.com/Point-LineDistance3-Dimensional.html> (accessed 2014).

CHAPTER

7

NAVIGATION AND CONTROL

In This Chapter

- Basic autonomous control
- How to use finite state machines
- An overview of pathfinding

For the purposes of this discussion, navigation will be defined as the process of getting from one location in a game to another. There are many differences between navigating the real world and in a virtual world, but a basic rule of game AI programming is *cheat!* Remember that it is only important that the simulated characters (*non-player characters*, or NPCs) appear to behave more or less like a real person. The program that controls the opponents does not have to navigate using vision, and has advance knowledge of everything that is in the game. In addition, there are markers can be placed all through the terrain that the AI can use to guide the NPCs. These markers are not visible to the players because they are not rendered as objects—they are merely points in 3D space that are used to create a path.

The opponents will, in fact, either take the same path at the same speed each time through a course, or will at most have a finite number of variations, each one taken with a particular probability when encountered. We

can even change those probabilities each time a choice is made, but that too is programmed and well defined. Most opponents do not act in an intelligent way, but are guided by quite simple and quick algorithms.

Basic Autonomous Control

The word *autonomous* implies independent or alone. Autonomous control of opponent characters in a game is essential for making the game fun and exciting, partly because it gives a sense of competition, partly because other characters or objects are obstacles and have to be avoided, thus creating a more complex problem to solve. If the opponents are autonomous, then they are controlled by software, and so we treat the problem as one of software design. The first question is, *what is the problem?*

Another way of asking this is, *what is the goal?* The goal of a player in a game is to have fun, but this does not necessarily mean winning. The entertainment value is in the game play, the puzzle, or the contest. Allowing the players to beat a game too easily can be detrimental to the commercial success of a game. Word will get out, and the game will not sell.

On the other hand, if a game is impossible to beat then it is just as bad. Players simply give up, and again word gets out that the game is impossible. We start to find them in garage sales.

Thus, we will state a carefully worded high-level goal for the opponent in a game:

Goal 1 - Since a main goal of a game is to provide entertainment and engagement, the opponents should provide a challenge to the player without being impossible to defeat.

This goal is vague enough to be a guide for any game, but too vague to implement as-is. However, it leads to a logical set of sub-goals that *can* be implemented:

Goal 1.a - If at all possible, the human player should beat some opponents if there are many. The player should not be humiliated by a computer if they are still actually playing.

Goal 1.b - Some opponents should decrease their skill level as they get a distance ahead of the player, allowing the player a chance to catch up.

Goal 1.c - The goal of an opponent is not to win, but to provide entertaining competition. If the game involves objects, like weapons, the opponent must be able to use them. It should play the game much as a human would.

Goal 1.d - When there are many opponents, they should not all be the same. They should have variable skill levels, and should in some sense respond to the displayed skill of the player.

Goal 1.e - The player should be offered a choice of difficulty when starting the game, so that easy opponents are available as well as hard ones. They can find their skill level and strive to improve it.

Most of these rules or goals are probably not a big surprise. You may not know that the AI system actually *dumbs down* to let you have a better chance, but you probably suspected as much. We will add to this set of goals, but for now we have enough to get started. Since the precise nature of the game we are discussing is unknown, we'll keep things at a high level for the moment.

How to Control a Car

Let's use the example of a car. There are similar movement controls and navigation features for cars and other NPCs, but vehicles have navigation as one of the principal issues. The user controls their car using the keyboard or mouse or game pad. Basically, there is something that can be treated as character input and that can be interpreted by the game as a command—for example, the left arrow means turn left, and the forward arrow means go forward faster. We could control an opponent in the same basic way, except that we don't need input. So, we could have a function that would turn the opponent represented by `x` left by the standard turn angle, in the same way that the player's car would turn by angular increments:

```
void turnLeft (CHARACTER x);
```

Or we could enhance the opponents by allowing arbitrary relative angles:

```
void turnRelative (CHARACTER x, float delta);
```

Or we could do this by allowing absolute angles:

```
void turnAbsolute (CHARACTER x, float angle);
```

The idea is that a car is controlled using a set of very obvious primitive operations that can be combined into higher and higher level operations.

We also need accessor functions that return key values to the AI system like a car's current speed, position, and direction so that a high-level goal can be specified in terms of low level operations and current parameters.

The high-level goals will be expressed in terms of minor goals, which may in turn be expressed in terms of local goals, and so on until at some level the goal is *go left*, which can be done with a primitive. The design would be from the high-level downwards, keeping in mind that the lowest level is pretty much defined at the outset. Let's take a detailed look at one of the possible intermediate goals: something called *cruising behavior*.

Cruising Behavior

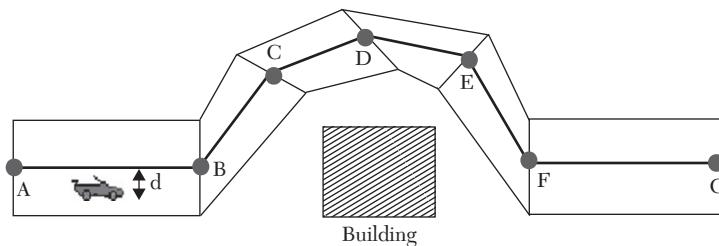


FIGURE 7.1 The simple set track and a set of line segments that allow driving behaviors to be defined.

The goal of this behavior is to maintain a set speed, more or less, while following a set track toward a geometric goal. It may also be important to avoid collisions. The *set track* is a piecewise linear path drawn along the game's terrain, perhaps along the middle of the road or race track, as shown in Figure 7.1.

The track can be identified as a connected sequence of straight-line segments, AB, BC, CD, DE, EF and FG. Points A and B are points in 3D space ($\mathbf{Ax}, \mathbf{Ay}, \mathbf{Az}$) and ($\mathbf{Bx}, \mathbf{By}, \mathbf{Bz}$) that define the ends of a line segment. The vehicle being controlled is at a known position $P=(x,y,z)$. It is moving toward the point B, at a distance \mathbf{d} from the track, and has a known speed S and desired speed S_{AB} . How do we keep the car on track?

So long as the vehicle is moving in a straight line, things are relatively simple. The program must try to keep the car as near to the line as possible, and will attempt to keep the speed as near to S_{AB} as possible.

```
s = getSpeed (THISCHARACTER);
if (s < Sab)                                /* vehicle going too slow */
    a = fmin(amax, k*(Sab-S));
```

```

else if (s > Sab)           /* vehicle going too fast */
    a = fmax (amin, k*(Sab-S));
else a = 0.0F;
s = a*dt + s;

d = linePointDist (A, B, P);
if (d < RIGHTTHRESHOLD) turnLeft (THISCAR);
else if (d > LEFTTHRESHOLD) turnRight (THISCAR);

```

There is a maximum and minimum acceleration, and if we are going too slow we increase the acceleration a little, up to the max; if we are going too fast, we decrease the acceleration (increase the deceleration) with the limit being the minimum. We then compute a new velocity based on the calculate acceleration and the time since we last did this. The constant k is used to apportion acceleration between time frames, and should be determined by experiment.

Then we pay attention to the steering. If we are right of the center line by a large enough distance, the car is turned by one unit to the left. If we are left of the center line by a large enough distance, we turn to the right by a unit. The system will straighten the steering angle automatically over the next few frames, but there is a risk of oversteering. We could fix that by only adjusting the steering angle every few frames.

Avoidance Behavior

While cruising it is possible to encounter an obstacle. In a race, it would not usually be a wall or tree or the like, because the set track would not be placed where there were natural hazards like this, but in an urban driving game or when using characters who are walking it would be common. An obstacle on the track will usually be another vehicle on the track ahead, presumably not moving as quickly. *Avoidance behavior* is what the AI vehicle does when it comes upon this situation.

The first thing to note is that other AI vehicles will be following the track, more or less. That's a reason that there's one in your way. So the first solution is to create another set track to be followed in order to pass a car on the existing track—let's call the original set track the *A-track*, and the new one the *B-track*. Some game developers would call the A-track the *driving line* and the B-track the *overtaking line*. Figure 7.2 shows this arrangement.

As a vehicle V_1 approaches another vehicle V_0 from behind, it detects the potential collision, not by traditional collision detection but by noting another vehicle ahead on the driving line. V_1 switches to the overtaking line and steers toward that line, thus avoiding the vehicle V_0 . If another AI vehicle is already on the B-track, then we simply slow down until it is gone. In a game like *Mario Kart*, we could also simply speed up and hit the other car, letting the collision sort things out. Unless V_0 is the player's car, of course.

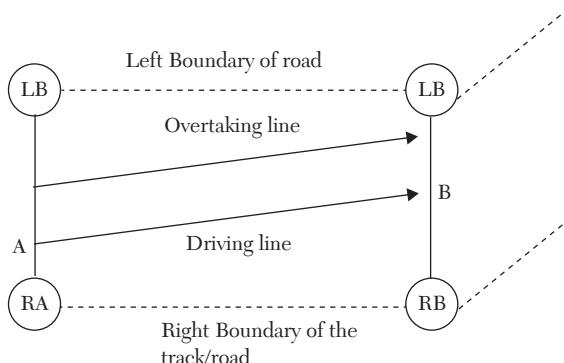


FIGURE 7.2 Section AB of the road defined in Figure 7-1, showing the driving line and the overtaking line.

If V_0 is the player's car, then its behavior is not predictable. If V_1 changes to the overtaking line, the player may just move over to block, but it could speed up, slow down, or hit something. Rather than having a fixed overtaking line in this case, we could create a new line by placing a target point in the middle of the largest gap, either left or right, between the player's car and the boundaries of the road. This point will move from frame to frame, but does present a target to steer at until V_1 gets very close.

The speed of V_1 needs to be controlled too. In principle, V_1 must slow down a bit until a gap opens up that is big enough to take advantage of. The AI could compute the trajectory of the player based on the current parameters and figure out where it will be in three to five frames. If the gap will be big enough at that time, V_1 could speed up to fill that gap and force the player to decide whether to collide with it or to avoid it. This is partly illustrated in Figure 7.3.

The use of the nodes or points that connect to create a path is generally referred to as *waypoint pathfinding*. The waypoints can be saved as coordinates in a special structure, in which there is a next and a previous

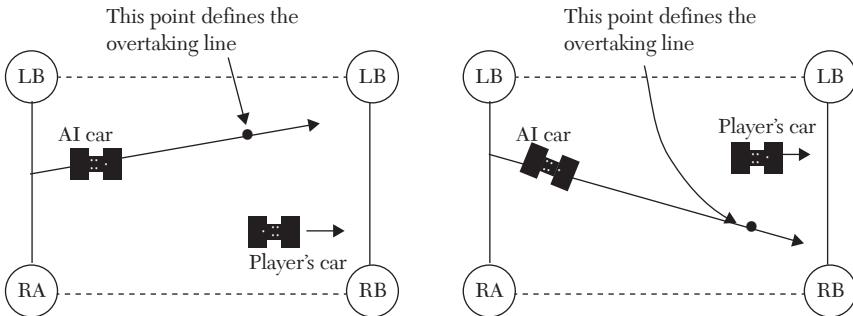


FIGURE 7.3 How to react if you come up on the player's car from behind. (Left) The new overtaking line uses the largest gap between the player and the side of the road. (Right) If the player moves over to block, it merely changes where the overtaking line is.

waypoint. Every vehicle saves the current waypoint that it is using, the one immediately ahead, in its own structure so that we don't have to search for the point closest to it. When the vehicle passes that waypoint, the next one becomes current. The use of waypoints eliminates the need for path finding algorithms in general, and simplifies the task of keeping the AI cars on the road and moving in the right direction.

In fact, there are a few ways to determine the path that an autonomous vehicle will use to traverse the race course. One is to create a driving and an overtaking line as we have described. The other is to create a different line for each AI car that can be on the track at the same time. Each car then has a relatively simple task—to keep as near to its driving line as possible. If another car is in its way, they simply collide, and the collision resolves the problem. The creation of many driving lines requires some effort up front, but simplifies the game as it plays—the cars just don't have to be as smart as the designers have done the work. Most game players don't recognize that there is only one line per car, especially if the lines are assigned at random at the beginning of the race.

The driving lines can also be associated with other information, like speed at each point. As a result, the line that is assigned to a car determines how well the car will do in the race. This practically eliminates the need for advanced computations while the game is going on.

Waypoint Representation and Implementation

The first thing to remember about waypoints is that they are, basically, points in 3D space. So the first thing we need to keep track of is their X,

Y, and Z coordinates. We also need a previous and next point, which can be stored as pointers. We may also have multiple previous and next points. Let's assume that we will have at most two of each. It has been pointed out that we may want to specify a speed. This will be the desired speed at that specific waypoint. If the point is approaching a turn, it will be in a decreasing sequence, and will increase on straight sections. It should be mentioned that the actual AI vehicle may not travel at that exact speed when passing through that waypoint. The specified speed is a goal.

A *Processing* class structure that could hold this information is:

```
class waypoint
{
    float x, y, z;
    float speed;
    struct waypoint_struct *next[2];
    struct waypoint_struct *prev[2];
    float Dnext[2];
};
```

The simplest way to use waypoints is to direct the vehicles toward straight lines that run through them. If we do, then the cars will always pass through the waypoints, and will turn sharply whenever each one is encountered.

A different way to manage waypoint traffic is to approximate a path between them, and to look ahead more than one point.

Finite State Machines

The idea that an NPC can be cruising, chasing, or avoiding is not especially profound, and clearly different behavior can be assigned to each mode or *state*. It is also convenient from the perspective of design to be able to break up the different behaviors into distinct parts, which can then be implemented independently. The use of the traditional computer science tool, the finite state machine, is a natural way to deal with this kind of situation. Finite state machines (of *Finite State Automata*, FSAs) are used in programming languages, computability, control systems, and artificial intelligence—and because they have been widely used their properties are well known, and efficient implementations abound.

We have seen the basic idea of an FSA when implementing the game states in Hockey Pong, for example. The basic idea of an FSA is a col-

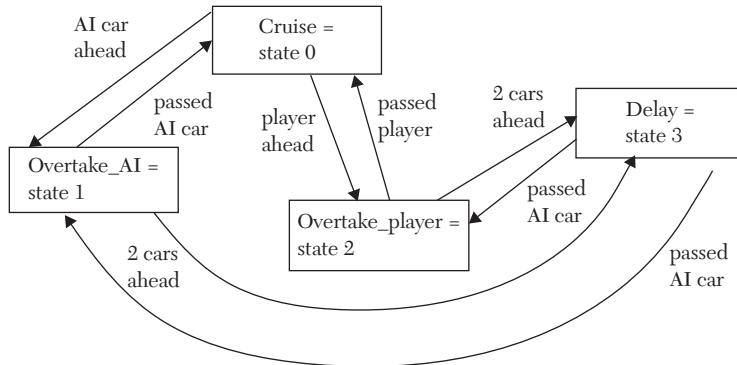
lection of states and of transitions between these states upon some input or calculation. The states have numbers, used in the implementation, and names, used by the designers and programmers as meanings of the states. In the situation described in the previous section, the AI vehicle starts out in the *cruising* state. If it encounters another AI car on the road ahead, it enters the *overtake_AI* state, and if it encounters the player's car it enters the *overtake_player* state. The behavior of the AI is quite different in each state, its goals and methods of achieving them are distinct. Figure 7.4 shows a representation of an FSA, specifically one for the three states above. It is essential to have a clear mechanism for moving between states, and a clear plan for what to do while in each state.

Mathematically, an FSA is simulated machine or mathematical construction consisting of a set of states, which are usually integers, a special state called the *start* state, a collection of input symbols or events, and a transition function that takes an input symbol and the current state and decides what the *next* state will be. The FSA begins a computation in the start state, and enters states based on input symbols/events and the transition function. There can be a special state called the *accept* state that can be used to decide when the calculation is complete.

So, if we are in the *cruising* state (state 0) and an AI vehicle appears in front of us, we enter the *overtake_AI* state (1); if we are in the *cruising* state and the player's car appears in front of us we enter the *overtake_player* state (2). These are the only state transitions out of state 0 in Figure 7.4.

While in the *overtake_AI* state, there are a couple of events that could take place. We could pass the AI car, or we could be blocked further. If we pass the car, we can go back to the *cruising* state again. If we are blocked—well, perhaps we need another state called *delay* in which we slow down and look for a change in the situation. The delay state will be state 3.

The delay state can mean different things to different vehicles, if we choose. Some cars will slow down and look for a gap through which they can sneak. Other cars might aggressively try to push their way through, colliding with their opponents if they refused to move. Still others might leave the road, if that were allowed, to try to find a way around. Any of these options could be associated with the same state, depending on the actual vehicle.

**FIGURE 7.4** A finite state machine for AI vehicles.

FSA in Practice

Implementing a finite state machine is a simple matter, so here are some good ideas about style and convention. Figure 7.4 will be used as an example, as it is simple and on topic.

The first thing to note is that the states are integers, from zero to some maximum. They also have meanings, and so can be given names. Thus, one generally define states as constants, in C for example:

```

final static int STATE_CRUISE          0
final static int STATE_OVERTAKE_AI      1
final static int STATE_OVERTAKE_PLAYER  2
final static int STATE_DELAY            3
  
```

Now we can define a state transition function. This function takes two parameters: the current state, and a state transition event. It results in the current state changing as defined by this particular FSA. This normally means that state transition events, however complex detecting one might be, need to be assigned integer labels and names, just like states:

```

final static int      TE_AI_CAR_AHEAD   0
final static int      TE_PASSED_AI_CAR   1
final static int      TE_2_CARS_AHEAD    2
final static int      TE_PASSED_PLAYER   3
final static int      TE_PLAYER_AHEAD    4
final static int      TE_ERROR_XXX       9
  
```

The error state `TE_ERROR_XXX` is representative of many possible error states, for example `TE_ERROR_103`, which means that some transitions are

actually illegal and result in some remedial action of the part of the program. Also notice that the transitions are context sensitive; the event `TE_PASSED_AI_CAR` does different things depending on what state you were in.

The actual machine can be implemented in a number of ways. A particularly good way, from the point of view of efficiency, modularity, and portability is to use a table. Transitions are integers, and these can be used to index into an array. States are integers too, and can also be used as indices. So, a state transition table for the FSA in Figure 7.4 could be:

	State			
	0	1	2	3
0	1			
1		0		2
Transition	2	3	3	
Event	3		0	1
	4	2		

This table contains state numbers, and is indexed by both current state and a transition event. So if we are in state 1 (`OVERTAKE_AI`) and we pass the AI car (`event 1=PASS_AI_CAR`) then we enter state 0 (`Cruise`); that is, an assignment of the form:

```
new_state = transition_table[TE_PASSED_AI_CAR][STATE_OVERTAKE_AI];
```

The missing entries in the table would be filled with either error states or a null transition meaning, *don't change the state*.

This is an effective implementation of an FSA, but relies on a correct initialization of the table. If the table is read in from a file it consists of integers that have no symbolic form, and this is somewhat error prone. If the table is initialized from a declaration, it is less simple to modify but we can now use the declared state names. Either way we do it, the code is less clear than some options and needs good documentation.

Another way to implement an FSA is to do so in discrete code. The usual situation is to just use `if` and `switch` statements. The first two columns of the transition table above could be implemented in the following way:

```
switch (state)
{
case STATE_CRUISE:
    if (transition_event == TE_AI_CAR_AHEAD)
        new_state = TE_PASSED_AI_CAR;
    else if (transition_event == TE_PLAYER_AHEAD)
```

```

        new_state = STATE_OVERTAKE_PLAYER;
case STATE_OVERTAKE_AI:
    if (transition_event == TE_PASSED_AI_CAR)
        new_state = STATE_CRUISE;
    else if (transition_event == TE_2_CARS_AHEAD)
        new_state = STATE_DELAY;
case STATE_OVERTAKE_PLAYER:
    if (transition_event == TE_2_CARS_AHEAD)
        new_state = STATE_DELAY;
    else if (transition_event == TE_PASSED_PLAYER)
        new_state = STATE_CRUISE;
case STATE_DELAY :
    if (transition_event == TE_PASSED_AI_CAR)
        new_state = STATE_OVERTAKE_PLAYER;
    else if (transition_event == TE_PASSED_PLAYER)
        new_state = TE_PASSED_AI_CAR;
default:
/*      Error code           */
}

```

In this case, there are no anonymous integers being used. All names are symbolic, and it is a simple matter to read through the code to see what the transitions are. This improves the maintainability of the code, and allows it to be more easily checked for correctness on a casual basis.

Both of the implementations above could be encapsulated within a simple function, which would return the next (new) state given the current state and the nature of the last event that occurred:

```
int transition (int state, int event);
```

The implicit assumption is that two events cannot occur within the relatively small time interval between two consecutive frames. This is pretty standard, and what happens in practice is that we sometimes get two state transitions in quick succession if two events happen more or less at the same time.

State and the ‘What Do We Do Now’ Problem

We now know how to move from one state to another, how to implement this, and what the states mean. What we do when in a particular state is not a matter for the FSA to deal with. What needs to be done is to determine what kinds of activities are associated with each state, and then to

execute code that performs those activities when in the correct state. We also need to execute code that determines whether any of the transition events has occurred.

Here is a general sketch of how the FSA based AI would function:

```
switch (state)
{
    case STATE_CRUISE:
        cruise (); break;
    case STATE_OVERTAKE_AI
        overtake_AI (); break;
    case STATE_OVERTAKE_PLAYER
        overtake_player (); break;
    case STATE_DELAY
        delay (); break;
}
event = test_all_transition_events(state);
state = transition (state, event);
```

This program causes the game to change between the feasible states as controlled by the events that have been defined by the designers and tested for in the function `test_all_transition_events`. This function can be quite complex, and it would probably be a good idea to test only for those events that are significant from the current state. This is why the state is a parameter.

Other Useful States

It is impossible to describe the states that a driving game can be in without knowing the detailed context of the game discussed. However, there are certain common options that can be seen to be commonly useful. This includes the following states:

Start

In driving games, it is common to have a race begin with all of the cars in predetermined start positions. The cars are not moving, and in fact may not move until the starter fires a gun or waves a flag. They then accelerate to the desired speed and select a driving track. This describes a state we could call the `Start` state.

In some games there is an actual countdown to the start, and if the player starts within a specified time of the actual start time he gets a speed

boost for a few moments. This can be done for AI cars as well, but because the AI system knows exactly when the start will take place it could easily cheat—actually, it's hard not to. So a random time is generated at the start, and any AI vehicle with a start time below the threshold would be given a boost.

Normally the `start` state changes to `cruise` when a certain speed is achieved or a specific time interval has expired, and the car has been assigned a track.

Air

A car that hits a big bump or crests a hill at a high speed may actually leave the ground for a few seconds. This has a few consequences: the engine usually revs up to a high value, causing the engine sound to change. The accelerator pedal has no practical effect—the car cannot accelerate or brake. The car cannot change direction, as the wheels are not in contact with the ground and it cannot be steered. This could be described as the `Air` state.

There should be a specific sound that is played when leaving the `Air` state, that of the wheels hitting the pavement while spinning—a combination bump and screech. Then we enter the state we were in before entering the `Air` state.

Damaged

Vehicles can become damaged in many ways. The simplest way is to collide with another car or with a stationary object, but some games involve weapons that can inflict damage, or processes that can cause the vehicle to deteriorate. Damage can result in an inability to perform normal tasks, like steering or braking. It can reduce the top speed or the ability to switch tracks. So there may be many damaged states, perhaps even one for every other undamaged state. That is, there should, in some cases, be a `cruise` state and a `cruise_damaged` state, an `Air` and an `Air_damaged` state and so on. In the `cruise_damaged` state, the car may not be able to reach the prescribed speed, but should still behave in the same basic way as in the `cruise` state.

Being damaged may also restrict the states that the vehicle can change into. For example, from `cruise_damaged` it may not be possible to move into the `overtake_ai` or `overtake_player` states, or their damaged equiva-

lents. Perhaps a damaged car should not try to pass another vehicle in the race, or at least one that is not also damaged.

From a `damaged` state, the car should change into the equivalent non-damaged state when it is repaired—for example, from `cruise_damaged` to `cruise`.

Attacking

In combat or combat driving games, an attack can take a number of forms, from simply firing missiles to an intentional collision or an attempt to push another car off of the road. The attacking state corresponds to the AI's effort to damage another car, perhaps another AI car or the player. The difference in behavior between `attacking` and `cruise` can be profound, since the attacking car has a quite specific goal—to destroy an opponent.

The attacking state may require that a vehicle actually *chase* a car, be it the player or another AI car. This is quite a distinct change from the usual `cruise` or `Overtake_AI` state in which the goals are simply to make geometric progress.

Defending

If an AI realizes that it is being attacked or chased, and there must be a carefully defined set of circumstances that determine when that is, then its car can adopt a strategy of avoidance, hiding, and perhaps high speed escape. These actions characterize `defending` mode, in those games where such conflicts are possible. The goal is obviously to hide from or destroy the attacker, and the previous goal indicated by the previous state is temporarily forgotten.

When the conflict is resolved, the vehicle should return to its previous state. Unfortunately the chase could result in the vehicle being quite a large distance from where the chase began, moving in the wrong direction for the original goal. So it may be best to move from `defending` to `cruise` and then have the system move between states based on the new local conditions.

Searching

Some games have objects that must be retrieved during the course of the game, and in other cases the opponents are moving about and you must find them. `Searching` behavior is like that of the `cruise` state, but the goals are a bit different. There is often no geographic goal in the `searching` state, only an objective one—to locate something. Thus the driving behavior

would result in large areas being covered and a minimum of backtracking or revisiting.

It would be reasonable to define searching tracks as a design feature of the game. Like a driving track, these would be defined by waypoints and would have the goals of the search built into the layout of those waypoints. The AI would then have less thinking to do, needing only to follow the waypoints blindly.

Patrolling

Patrolling behavior is very much like searching—patrolling can be called *searching for trouble*. Think of a police car on watch, driving the city at night. This is patrolling, an organized random route through an area. It should be random in practice so that bad guys cannot predict where you will be, of course. Whether it is truly random in the game is up to you, the game creator.

The nature of what is being sought is also a bit different from that seen in searching behavior. It is possible that a patrol is seeking a particular person, in which case it may be the same a searching. It may also be that the patrol is seeking a set of behaviors that indicate a crime in progress or some form of enemy activity. This is harder to identify, and requires some careful definition of the goals in advance. For example, speeding is simple to spot since the AI always knows how fast all objects are moving. An illegal lane change by the player may be more difficult to spot.

Skidding

When a car tries to turn a corner too fast, the wheels slide on the road. They try to keep moving in their original direction, as indicated my Newton's law. We call this a skid, and it may be useful to define a skidding state. This state is characterized by a lack of control, so steering will work differently than in other states. Turning into the skid may tend to align the axis of the car with the direction of motion, but the car continues in much the same direction as before. Turning away from the direction of the skid will tend to give the car a rotational velocity about its center, again without changing the direction of motion of the car very much. Braking may make the skid worse, but slowing down would permit the wheels to grab the pavement and give control back. It is a complex situation, but anyone who has been in a skid knows one thing—the original driving plan, be it going to the

store or getting to work, goes out the window in favor of just staying out of the ditch and getting control back.

Stopping

The AI might need to stop the car from time to time, to pick up passengers perhaps or to collect an object. A car cannot stop instantly, and it may be necessary to pull over to the side of the road to avoid being hit. This stopping behavior is certainly needed in some games. It is necessary to enter the stopping state well ahead of the point where the car wishes to stop. The goal is to stop at a particular place to conduct an activity, and so that place must be identified in advance and the car needs time and space in which to slow down. Again, we'll need rules to dictate how far away to change states given the current velocity and direction.

Pathfinding

Until this point, the assumption has been that an NPC exhibits a specific behavior based on what is happening in the game, but sometimes an NPC has a specific destination, and sometimes the NPC behavior is expressed in terms of a destination. Patrolling behavior, for example, may well consist of a set of waypoints to be visited in sequence. In static situations this can be handled by fixed waypoints, but if obstacles can be placed in the way then the situation becomes more complex.

A full AI solution would be to discover a path to the next waypoint on the NPC's path. There are many such path finding algorithms, including the A* algorithm. If the NPC is blocked by a moveable object but is still relatively near its path, it might be possible to simply move to a nearby location and then back to the path. A* attempts to find the best route according to some heuristic, usually based on the shortest distance. As a result, A* can be more time consuming than we might like, given that a game executes in real time and has rather a lot to do. What would be acceptable is a less than optimal but still feasible route that takes less time.

One practical idea is to use a predefined grid of directions, indexed by using the vehicle's current position. This grid could be relatively coarse, containing perhaps a few thousand entries, and it should map onto the terrain of the game. Entries in this grid, easily implemented as a two dimensional array, would be directions: either vectors or simply compass angles. The game designer would have to fill in the values at each location in the

grid with the direction to steer to get back to the road or path. This is the usual trade off—to make the machine seem clever, a person has to do a lot of work in advance, just as we did with waypoints. The other traditional trade-off in computer work is that of space VS time, and that can be seen here too. In order to speed up the path finding in this situation, a lot of extra storage space is used (the grid).

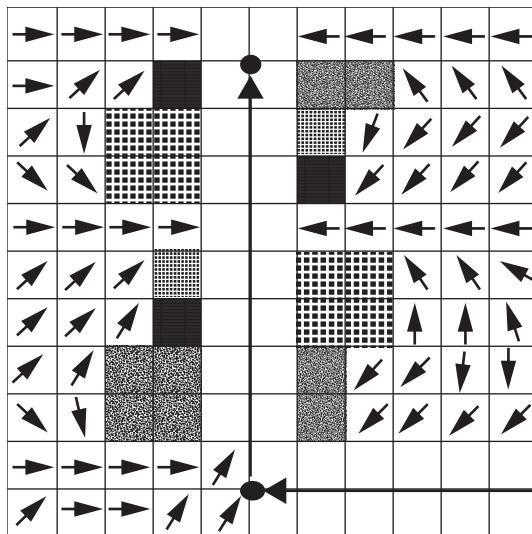


FIGURE 7.5 Use of a directional grid to find a route back to a road.

Squares with arrows show the direction that the car should move to get back to the path. The path is the route between the black circles, marked with the long thick arrow striped lines. If the car is not pointing in the right direction then it must at least steer that way.

If this method is used, the sequence of steps is:

1. Find the grid element that corresponds to the current location of the vehicle. For example, if the playing area is $1,000 \times 1,000$ yards, we could break up this area into 25×25 grid elements, each being 40 yards square. Locating the grid is a matter of dividing the (x,z) coordinates by 40 and truncating.
2. Steer in the direction saved in the grid entry. This could be a byte value to save space, and could be in fairly crude terms, since we simply have to get back to the path, not find the best route.
3. Use a low speed, since we're out of the race for the moment anyhow. The only obstacles that are a problem are moving ones, since the grid will be designed to avoid stationary objects. We could store a suggested speed along with the direction, again crudely quantized.

4. Grid elements that are near a path could contain a special value to indicate to the AI that the car should now be allowed to continue in its usual mode.

Figure 7.5 shows such a grid in a small example, and this example has obstacles so that it is easy to see how the grid is built—directions are chosen to steer the car toward the path, not always directly at it, but sometimes around static objects. As the car moves from one grid to another, it adjusts its steering direction to the new grid direction.

Thus, anywhere that the vehicle ends up after a collision will have a grid entry that directs the AI how to control the car.

A* Search

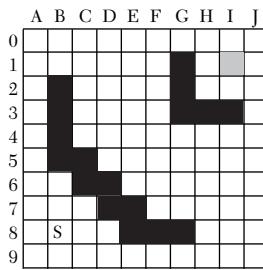
The A* algorithm is a method for searching through a set of states for a good one, one that should lead to the solution of a problem. This is a pretty vague statement, but A* can be used for quite a few distinct kinds of problem in AI, so it makes sense to be vague. In terms of finding a path, a *state* will be a situation that has a position identified that is unique and is associated with a positional goal, a target position we are trying to get to. We also need a way to determine a *cost* associated with positions; in terms of paths, a cost may be how long it will take to get to the goal, or how much fuel it will take. The cost of moving from one position to another may be connected to the terrain. Mud will cost more, and so will steep inclines, while paved roads will probably cost the least. The idea behind A* is to create a method for determining which route costs least without exploring all of the possibilities, which could be quite expensive.

It is important to realize that in order to use the A* algorithm the playing area must be divided into a grid, like we did before when using the directional grid. Each grid element corresponds to a discrete state, and has a value that is related to the start and goal states. Each of these grid elements is called a *node* in A* terms. Each node has a *cost* associated with it, which is related to how far it is from the goal or how expensive the route is from that point.

There are a couple of obvious things to notice before we get too far into the description of the method. The first thing is that it is logical to reduce the amount of computation that is done by remembering the cost associated with each node, and not re-computing it. Next, we want to keep a collection of nodes that are candidates for the next one in a path. A good

way to do this is to have a set of nodes that are possible next ones—this is the open list or open set. We will also have a list of nodes that do not need to be considered, possibly because they have already been examined—this is the closed set.

The A* algorithm is important enough in games and AI to spend a few pages on, and a picture can be very valuable in explaining how things work. So, let's walk through an example that illustrates the method. Here is the grid that gives the situation:



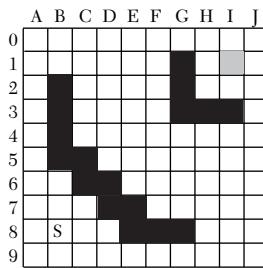
The starting point is marked 'S' and is at the node indexed as B8. The goal is a shaded node/square at I1.

Open list Closed list
Empty Empty

FIGURE 7.6 Initial situation in the A* example.

The first thing to do is to add the node **S** to the open list, since we need to consider it as the 0th step in the path to the goal. The open list should be sorted so that the node with the smallest value of the total cost function (which we will call **F**) is first. The function **F** is a score traditionally composed of the sum of the function **G**, which is how much it cost to get to this node, and **H**, the estimated cost for the remaining nodes between here and the goal. So **F = G+H**, and it seems as if **H** is impossible to calculate.

G is easy—each time we move horizontally or vertically to get to a node, we add 1 to the value of **H** for that route, and we add the square root of 2 for diagonal steps if they are allowed. To make the calculations a bit faster, we multiply by 10 and convert to integers, since integer math is much



The distance between S and the goal at I1 is $10*(8-1) + |I-B| = 70 + 70 = 140$.

Open list Closed list
B8 F=0+140 Empty

FIGURE 7.7 Initialization for step 1 of A*

faster than floating point math - so horizontal or vertical steps cost 10 and diagonal steps cost 14.

A common way to determine \mathbf{H} is to use the *4-distance* or *Manhattan* distance between that node and the goal. This is simply the number of rows between the nodes added to the number of columns between them.

After \mathbf{S} has been added to the open list and \mathbf{F} is computed, here is the result:

Next, we take one of the nodes from the *open* list—the one with the smallest \mathbf{F} value. Right now, there's only one node in the open list, \mathbf{S} . Now add all of the nodes that neighbor \mathbf{S} to the open list and move \mathbf{S} to the closed list. Compute \mathbf{F} for all of the new open list entries.

Remember, left-right and up-down neighbors are a distance of 10 from \mathbf{S} , and diagonal neighbors are a distance of 14. A sample calculation of \mathbf{F} for the node at B7 is:

$$\mathbf{H} = \text{distance to goal} = 10*(6 + 7) = 130$$

$$\mathbf{G} = \text{accumulated distance from } \mathbf{S} = 10$$

$$\mathbf{F} = \mathbf{G} + \mathbf{H} = 130 + 10 = 140$$

We do this for all eight neighbors in the open list to arrive at the results in Figure 7.8.

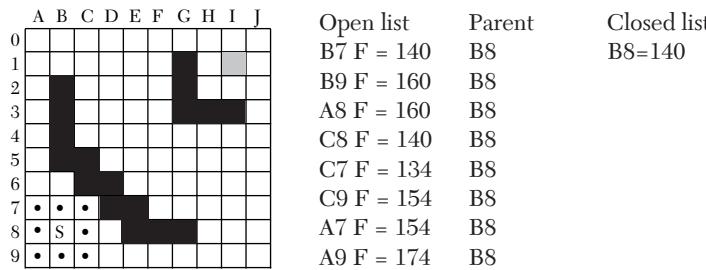


FIGURE 7.8 The results of step 1 of A*

Whenever a node is added to the open list, we make a note of how we got there—it is the neighbor of a node that was on the path, and that node is the parent. We always remember the parent of a node, because that's how we trace the route back to the start when the method is done.

Let's do the next step. We pick the node in the *open* list that has the smallest value of \mathbf{F} - in this case the C7 node - and put it into the closed list.

Then we start examining its neighbors. We must ignore squares that can't be travelled on, so the black ones that represent an obstacle are ignored. Also ignore nodes in the closed list. Clearly there are just four nodes that are legal neighbors of C7: C8, B7, B6, and D8. Add these to the open list if they are not already there. C8 and B7 are already there, so we don't add them, but we do check to see if the value of **F** for these nodes is smaller than it was before—that is, is the path that goes through the node C7 better than the one that has been computed already? If so, change their parent to C7 and their **F** value to the new one, otherwise do nothing. For the new nodes B6 and D8, add them to the open list and compute **F** values.

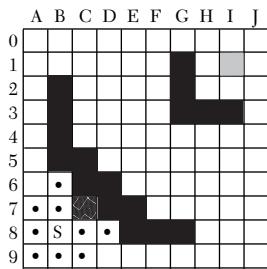


FIGURE 7.9 Step 2 of A*

	A	B	C	D	E	F	G	H	I	J	
0											
1											
2											
3											
4											
5											
6											
7											
8											
9											

	Open list	Parent	Closed list
			B8=140 -
	B7 F = 140	B8	
	B9 F = 160	B8	C7 134 B8
	A8 F = 160	B8	
	C8 F = 140	B8	
	B6 F = 148	C7	
	C9 F = 154	B8	
	A7 F = 154	B8	
	D8 F = 148	C7	

FIGURE 7.9 Step 2 of A*

Now do it again. The node in the open list with the smallest **F** is B7. Move it to the closed list and place its eligible neighbors into the open list. There are only two nodes of interest here—node A6 is new, and is added to the open list. The node at B6 is one that is already in the open list, but the exciting thing about it is that the value of **F** computed through the new parent is smaller than the old value. Therefore we change parents to B7 and adjust its **F** value to the new one, 140. The new situation is shown in Figure 7.10.

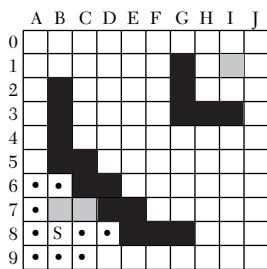


FIGURE 7.10 Step 3 of A*

	A	B	C	D	E	F	G	H	I	J	
0											
1											
2											
3											
4											
5											
6											
7											
8											
9											

	Open list	Parent	Closed list
			B8=140 -
	C8 F = 140	B8	
	B9 F = 160	B8	C7 134 B8
	A8 F = 160	B8	B7 140 B8
	B6 F = 140	B7	
	C9 F = 154	B8	
	A7 F = 154	B8	
	D8 F = 148	C7	
	A6 F = 140	B7	

And so we continue, pulling out the open node with the smallest **F**, put-

ting into the *closed* list, and putting its neighbors into the open list.

When do we stop? First, when the open list is empty. This means that the goal cannot be reached. The other termination condition is that we add the goal node to the *open* list. We trace the path of parents back from the goal node to read off the sequence of nodes in the optimal route.

The algorithm, in summary, is:

- 1.** Create **start** and **goal** nodes
- 2.** Place the **start** node into the *open* list
- 3.** Repeat while there are nodes in the *open* list
- 4.** Select the node **P** from the *open* list with smallest **F** value
- 5.** If **P=goal** then we quit with the solution
- 6.** For each neighbor **Ni** of **P**
- 7.** If **Ni** is unusable or in the closed list then continue from 6
- 8.** Let the cost of **Ni** = **H(Ni)**+ distance to **P**
- 9.** If **Ni** is not in the open list then add it
- 10.** Else if **Ni** is on the open list and the path has a lower **F**
- 11.** Then change **F** to the new value, change the parent of **Ni** to **P**
- 12.** End of FOR
- 13.** End of repeat
- 14.** If the open list is empty, there is no path to the goal.

Now we have a path from the AI vehicle that was knocked far off of its path by a collision to a waypoint that is on the original path the car was following. In other words, we have a way to get back to the normal situation after being knocked off the path.

Stochastic Navigation

The word *stochastic* means having a random component or element, and that's really what is wanted from ambient traffic. If you look at traffic from the top of a building, they individual vehicles behave both predictably

and randomly—they predictably obey traffic rules, but follow what looks like a random route. That's because we don't know where the cars are going. They all have a destination, but without knowing what it is we don't really know what a car will do at the next intersection, and especially at the intersection three blocks down. We want the traffic to look natural, and do not want all cars to turn left at 5th street, or to have the same cars go around the same block for the whole game.

So, each car should have a plan for at least the next choice. If a car is going to turn left, it makes sense that it should get into the left lane before the intersection. Each vehicle in traffic should have a short-term plan, which is updated every time it executes a planned move, like a turn. The plan is random, so it is based on the drawing of random numbers. The most likely event is to drive straight through an intersection, but left or right turns have a finite non-zero probability. For example, we could have:

Straight through	80% chance
Left turn	9 % chance
Right Turn	9 % chance
Right next alley	1 % chance
Turn into next access	1 % chance

Now draw a random number x between 0.0 and 1.0. The code for the above is:

```
if ( $x < 0.8$ ) plan = GO_STRAIGHT;
else if ( $x < 0.89$ ) plan = TURN_LEFT;
else if ( $x < 0.98$ ) plan = TURN_LEFT;
else if ( $x < 0.99$ ) plan = NEXT_ALLEY;
else plan = NEXT_ACCESS;
```

If the car enters a parking lot, it should park. This activity is likely initiated by a finite state machine state change.

It is important to realize that the traffic needs only to behave properly so long as the player is watching. Indeed, it takes time for the AI to move the vehicles sensibly, and if we can avoid taking this time it would be good. Should we create traffic when it becomes visible? That is, when the player's car turns a corner, do we need to invent some cars and plans for them?

That is certainly an option, but it would cause a problem in cases where the player chooses to explore the environment, especially if he does so by following ambient traffic. Imagine turning a corner to find that the cars you just saw have vanished. No, it is probably better to have more of the traffic

be inactive (not moving) until it is within a specific radius of the player. Naturally, if the player stays in one place too long, the traffic in the neighborhood could vanish—as it leaves the active radius it stops, and nothing can start up until the player moves closer.

Things are getting complicated. So, what may work is to give some CPU time to moving ambient traffic once in a while, each few frames. If the player is idle, there is going to be a lot of free time to give to this task. So, if traffic is within a radius of, say, 5 blocks of the player's car, it will get a *turn* (a few cycles) each frame for movement control. Otherwise, it will get a turn based on its position in a queue and the number of free cycles. As the frame rate increases, we have extra time to give to the traffic, unless the player is engaged in something such as combat. So, the distant vehicles are placed in a queue, and the front few are given movement control each frame, and are then placed at the end of the queue. This is fair, and will automatically give as much spare time as possible to traffic motion.

Summary

Navigation is the process of getting from one location to another. Sometimes there is a final destination which is arrived at in stages, and sometimes there are predefined routes to known destinations that are defined by the game developer. A *waypoint* is an intermediate destination along a route, and is defined by a set of 3D coordinates and actions to take upon arriving. A character moves from one waypoint to another on its way to its final destination. For characters that don't have predefined destinations we use *pathfinding* methods like the A* algorithm.

Characters may have a set of states that control their movement and navigational behavior as a function of their current state. It is also common to have characters simply moving around to create some form of traffic, essentially providing ambiance.

Exercises

The exercises below will apply your knowledge of the material in this chapter, and sometimes require that you do some more research before you are able to complete them. Solutions to selected exercises are available on the companion disc.



1. Create a simple elliptical track on an image 800×600 and use it to implement a basic driving simulation. One car should drive around the track completing at least 5 laps before stopping. Use no fewer than 8 waypoints to guide the vehicle.

For the next exercises, use Figure 7.11, which shows an 800×600 image that is to be used in pathfinding. Presume that a character's avatar, represented by a small blue circle, is to move from the small house in the upper right of the image to the larger house in the lower left. Presume also that this dot may not pass through the river or through the brick barriers.

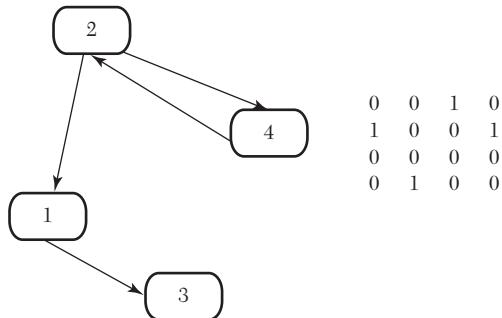
2. Mark passible and impassible squares on the image, creating what we call a *mask*. Have the avatar move toward the destination when possible and back up and try a new path when not. Turn at random when a decision has to be made, toward the destination when possible. Does the avatar find a path? If so, how long does it take?
3. Find a set of paths to the destination manually and mark them either in a same-size image or as arrays of individual pixel x and y movements. Create at least four paths that branch from the starting point and are selected at random. Does the behavior look realistic?

IDEA *A program has been written that allows the developer to use the WASD keys to move from the source to destination, and record the path. The path was written to a file and read in and used by the problem solution. You could write such a program too, or use the one provided. (**pathmaker.pde**)*

4. Repeat Exercise 3 using waypoints. In what ways is this solution better and in what ways is it not better than the ones previously tried?
5. Add a new NPC avatar to the solution of Exercise 4 above, a red circle, which will try to prevent the player's blue avatar from reaching the destination. It will start in the upper left and move toward the blue avatar when it *sees* it—that is, when there is a clear line of sight between the red and blue circles that does not pass through a brick wall. If the red avatar gets to within 12 pixels of the blue one then it succeeds and red wins. If the blue avatar gets to the destination, then blue wins. Who wins this game and under what circumstances?
6. Navigation is not only connected with existing paths and physical obstacles, but sometimes by more abstract things like traffic rules. Describe the rules for behavior when a car arrives at a stop sign on a typical city

street. Sketch a plan for the behavior of a vehicle from the approach to a stop sign until the moment that it is decided the vehicle can proceed.

7. Streetwise navigation can be complicated by road closures, one-way streets, and other features of modern life. In computer science terms, street intersections can be considered to be nodes on a *graph*. An entity called an *adjacency matrix* is used to represent which nodes are connected to which others. An element j in row i of the matrix is a 1 if there is a way to get to node j from i in one step. Here is an example:



There is an algorithm to determine if a path exists between any two nodes and how long that path is—it is referred to variously as Warshall's algorithm, the Floyd-Warshall algorithm, or the WFI algorithm. Look up this algorithm and describe it. Then discuss its usefulness in finding routes in urban contexts. How is a one-way street represented?

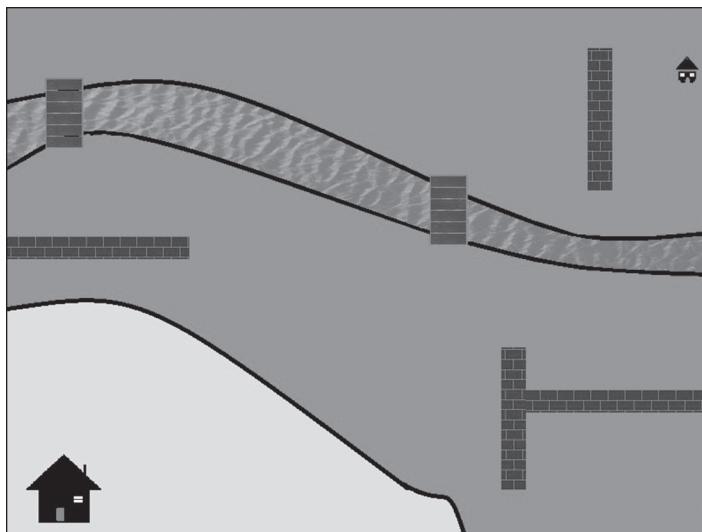


FIGURE 7.11 Sample 2D path finding image for the problems.

Resources

Pathfinding tutorial: http://wiki.gamegardens.com/Path_Finding_Tutorial

Gamasutra tutorial on Realistic Pathfinding: http://www.gamasutra.com/view/feature/3096/toward_more_realistic_pathfinding.php

Video demonstration of Dijkstra's algorithm:<http://www.youtube.com/watch?v=8Ls1RqHCOPw>

Pathfinding concept, the basics (Michael Grenier):

<http://mgrenier.me/2011/06/pathfinding-concept-the-basics/>

Bibliography

Gilbert, E.G., D.W. Johnson, and S.S. Keerthi. "A fast procedure for computing the distance between complex objects in three dimensional space." *IEEE Journal of Robotics and Automation* 4 (1988): 193-203.

Higgins, Dan. *Generic A* Pathfinding, AI Game Programming Wisdom*. Boston, MA: Charles River Media, 2002.

Matthews, James. *Basic A* Pathfinding Made Simple, AI Game Programming*

CHAPTER 8

A 3D GAME EXAMPLE

In This Chapter

- Creating a basic 3D Game
- Comparing 3D and 2D rendering

We now know enough to think about building a game that takes place in three dimensions. Many video games are 3D, although of course all of them project the game universe onto a 2D screen, and almost no games actually immerse us in graphics and sound. The same will be true of what we are going to create here. By necessity, it will be small, but that doesn't mean it can't be fun.

A game design document for any reasonable 3D game would be too large to contain in a book, so we'll have to do without—which is not usually a good idea, but most aspects of this game have been carefully considered before this chapter was written. We want to demonstrate three dimensional methods, sound, and some basic AI without building something too large. As a game developer, it would be good to try something relatively original too. There are a lot of shooters and racing games out there.

Consider a game that takes place on the ocean—there aren't very many of those. Submarines have always been interesting, and even more so when you consider that in the 1940s these boats could navigate without visual

cues, GPS, or even decent navigational radar. Put yourself in the position of a sub commander: how well can you know your position without any visual cues? It must take a lot of study and practice to be good at it.

We are going to design and build a small submarine game that simplifies the situation for the commander—there is a window. It is 3D in that the player can navigate within a relatively small volume of ocean, moving left, right, forward, backward, up and down. Objects within this game are 3D as well. An important thing to remember is that this game is an illustration of 3D methods, not really a game for mass consumption. It is designed to demonstrate elements of *Processing* that are useful in 3D games, and as such is something of a component stew. It could be taken in many directions to become a better game, but as a teaching tool if can serve many purposes, and students can modify the code to many ends.

SMV Rainbow

The game *SMV (Sub Marine Vessel) Rainbow* is a game that involves collecting objects from the bottom of the sea. It is based on a previous game, *OceanQuest*, which can be found in the game's literature. The sub itself is small and has a window that permits you to see what's going on outside the boat. You are the only person on board. A cargo vessel carrying dangerous material has sunk and has scattered cargo containers across the sea floor. Some of these containers have radioactive isotopes inside and must be found and neutralized. This is the player's task in the game. If the player finds and neutralizes all of the radioactive material then they win; if they run out of time or energy, they lose.

This back story solves a couple of problems. First, character animation is complex, time consuming, and difficult to get right. Since there are no humans on the ship or elsewhere in the game, character animation is not needed. Second, the game does not involve the destruction of property by the player, but on the contrary is a rescue mission. Ethical issues common to many shooting games are not issues here.

Let's run through a quick design process. There are two regions that must be visualized by this game—the control room and the underwater landscape. The control room can be limited to a view of the controls and sensors and a large window through which the seascape can be seen. Some of the instruments are more important than others for educational purposes—many games have a heads-up display showing game status, and many games

involving vehicles have a mini-map that shows the entire region and the player's location in it. These things will be included.

The player will be able to control the ship's position and depth, and navigate within a small region of the unnamed ocean. The player can identify undersea objects and should be able to detect the radioactive targets being searched for. When all of the targets have been found and neutralized, the game is over. Play value is limited, but remember that this is an *example* of a 3D game. Many features common to 3D games are implemented here, just not as many features as would appear in a game intended for distribution.

Screens

All games have a splash screen, and this will direct the player to a set of choices. We should have a screen for options, and in a game more complex than *Hockey Pong* it is appropriate to have a screen that explains the rules. There will be a screen that contains the game itself, and one that the game switches to when play is over. From this latter screen, the player can go back to the splash screen and play again, or exit to a screen with credits. This is a pretty standard layout, and is shown in Figure 8.1.

From this description and the diagram, it can be noted that the following buttons are needed:

Play	Go to play screen and start the game.
Options	Go to options screen, set volume, difficulty, etc.
Instructions	Tell the player the rules
Quit	Exit the game
Back	Return to the previous window

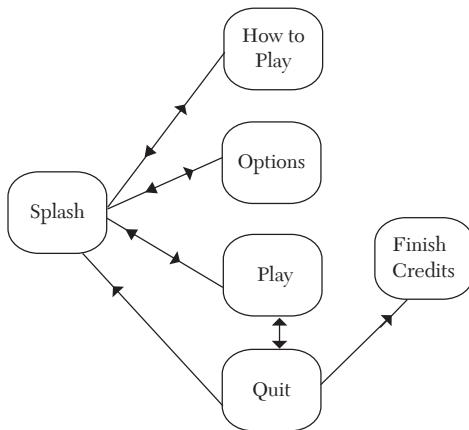


FIGURE 8.1 Game screens and the possible transitions.

There will be more buttons and controls on the options screen once it is decided what the options will be. The buttons initiate transitions between the screens, and so even if the screen art and game play are not finalized, the first prototype should draw the buttons and enable the correct transitions between screens. If any concept art is available, it should be included on the splash screen at least.

Game Environment

This game takes place at sea. A cargo vessel has sunk and will be on the sea floor somewhere. Cargo containers have been scattered about, and some of these are targets while others are not. The player must investigate them individually by navigating toward them using sensors. The player is to neutralize dangerous material. To make things simple, let's decide that there will be four containers to be located, out of a total of twelve.

The sub has a sonar system, a window, and a mapping data system that allow the player to observe the region. Sonar works all of the time but has a limited range. Active sonar involves sending out a sound (a *ping*) from the sub and looking at the reflections that result, whereas passive sonar simply listens for the sound of engines and propellers of other ships. Passive sonar can't be used in this game very effectively, so we'll not implement it.

This discussion reveals much about the implementation of the game. First it describes graphical and sound assets that will be needed. It describes the nature of the play volume and specifies a rough size; we need enough space to place a dozen cargo containers with space in between and a sunken cargo ship. This design provides a scenario that allows the establishment of initial positions for objects too, because the distance the sub can travel is relatively small, and we want the game to be fun rather than consisting of lots of tedious random motion.

The geographic location of the scenario will be ambiguous, but the layout is as in Figure 8.2. Objects will be placed in this region so that the submarine can detect them. They will be stationary and visible through the window from nearby, and visible on the sonar mini-map from a much greater distance.

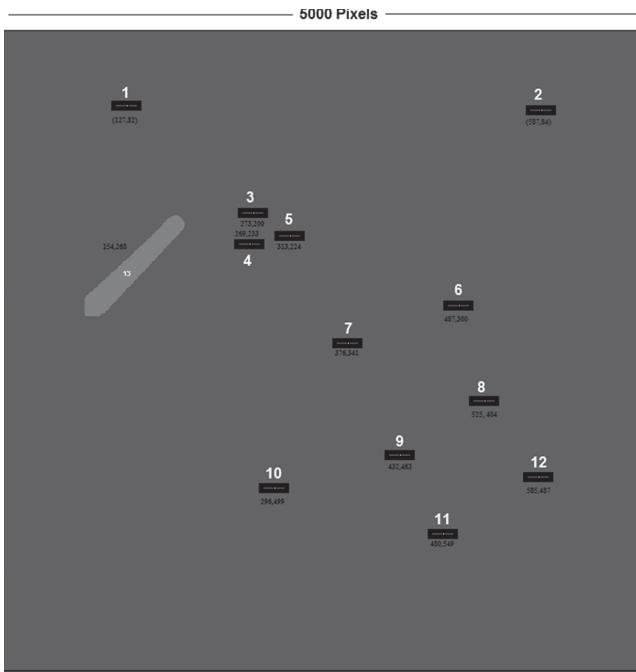


FIGURE 8.2 The geographic region within which the game takes place.

Three Dimensional Assets

Without a design document, we do not have a complete list of assets. In this case we have a general idea of what is needed, but will likely miss a few things and have to come back and create them. That's okay in this instance, as the purpose of this game is to teach you how to build a 3D game. In addition to learning about building a 3D game, we'll also learn that planning ahead is a good idea through actual experience.

The process we are about to embark on is called *Rapid Application Development* (RAD). It abandons significant advanced planning and begins projects by building rough prototypes, then refining them by interleaving stages of design and prototyping. The final prototype ends up being the product. This may be fine for software but is not generally a good choice for games being developed by a team. A game design document is a key tool for creating a schedule and ensuring that everyone is building the same game. In fact RAD does not work well for large and very complex software systems.

either, mainly because it can't lead to sensible costing or timetabling of the project. It will work here because the nature of the game matters less than what it teaches about putting it together, in terms of the technical aspects of three dimensional games. Lessons about planning, budgeting, and other crucial portions of a large game project will have to wait for another day.

The first prototype (number 0 in the computer way of counting) merely allows transitions between windows and places with buttons on those windows, and even that took about 300 lines of code. Prototype 1 will involve the creation and display of the essential objects used in the game, these include:

Cargo containers.

Cargo vessel.

The bridge and the active objects on the bridge.

Each of these items has particular design issues. Until models can be imported and used more effectively than at present, it will be best to construct these objects from basic *Processing* operations rather than by reading them in as sets of polygons. Thus, simplicity is going to be important. Let's examine these items in the order listed, which is simplest to most complex.

Cargo Containers

It will be important to have some variety in the appearance of the cargo containers. These are quite simple to make; they are simple prisms that are textured with a set of boxcar-like images. They are precisely like the buildings that were rendered in Chapter 5.

It would be possible to photograph some boxcars or containers, or to download images from a website. It's also possible to create non-photorealistic container textures using the basic Paint program. We need a side and an end for each container, and they can be used twice on each object. A top will not be used, but would be a variation on the side. Figure 8.3 shows an example of the textures created in *Paint*. They look a bit too pristine. A container travels through grubby ports and is driven thousands of miles along highways. It should show some wear. Of course, other game developers have had similar observations and have come up with a solution—a *grunge map*.

A grunge map is just an image. It's an image of grime and dirt, intended to be used as a texture. One takes an image, like the one of a cargo container, and a grunge map, and overlays the map over the image. If it's done

properly, the grunge can be seen through and the object beneath looks, well, grungy. You can apply a grunge map using *Photoshop*, but it's also possible to do it using *Processing*. Rather than applying the grunge map during game play, which will take CPU cycles, we should create texture images with the grunge map already applied (*pre-grunged*, you might say). I have provided you with a tool written in *Processing* that does this—it reads a texture and a grunge map and allows the user to specify how transparent the grunge should be. By typing W and S, the image displayed on the screen gets more or less grungy, and when it looks good simply type ESC and the image is saved in a file. This tool uses the *Processing tint()* function to make the texture more or less transparent. The basic code is:

```
image (g, 0, 0);
tint (255, alpha);
image (t, 0, 0);
```

where *g* is the grunge image and *t* is the texture image. Figure 8.3 shows the container textures created using *Paint*, a grunge map, and the result of applying the map to the image. It looks much more realistic after the grunge map has been applied. The rendering of the container, which is just the textures applied to the sides of a rectangular prism, can be seen in Figure 8.4.



FIGURE 8.3 (Left) A texture created using *Paint*. (Center) A grunge map. (Right) The grunge map applied to the boxcar texture using the grunge tool provided.

Creating other containers is a simple matter of making an end texture and a side texture and mapping these onto a new prism. The *Processing* procedure that draws these objects takes the position of the container and the two textures as parameters. The stack of containers seen on the right of Figure 8.4 was created with four calls to that procedure, passing different positional coordinates and textures.

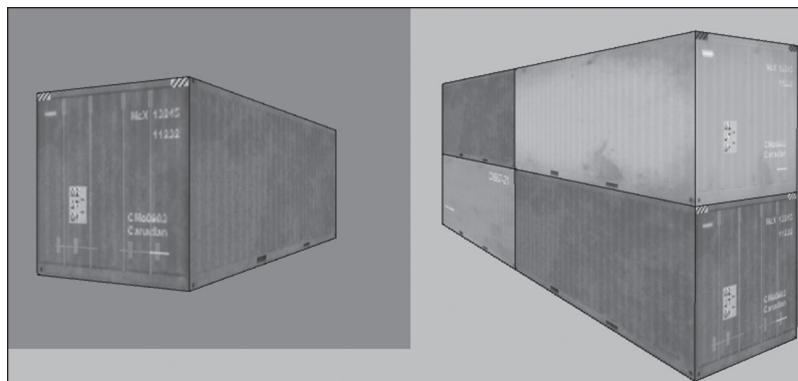


FIGURE 8.4 Cargo containers rendered as prisms and textured with both a boxcar texture and a grunge map.

A Cargo Vessel

The problem to be solved by the player in this game has been created by the sinking of a cargo ship. Such a vessel should be at the sea bottom for the scenario to make sense. It is a 3D object too, but is not a prism and needs some more detailed design to see how it can be built from simple quadrilaterals

Cargo ships are simple objects, but are certainly more complex than a basic prism. Also, they will nearly always be seen from a distance, so some details are less important than overall impression. Normally a set of 3D models would be used, but it is relatively simple to construct something that looks a lot like a ship out of basic quads. Figure 8.5 shows an example of a typical cargo vessel, and a set of coordinates that could be used to model one. This was designed using graph paper to ensure that the coordinates are consistent. The coordinates are those in any Y plane, and this design has the Y values range from +10 to -10, with the water at Y=0. There is a top consisting of multiple points connected into a polygon with a simple texture. There is no bottom; it will never be seen.

The prism at the aft end represents the structure usually found at the aft of such ships, a building for housing the crew and control room sometimes called the *superstructure*. This is a prism again, and will be textured to look like the real thing. This model has the basic shape of a cargo ship, and can be textured just as the containers and buildings have been.

Small variations can be added to make ships look different. The antenna can be removed and different textures can be used. Small shapes added

to the fore and aft can make significant differences. There will be only one of these ships on the screen at a time, and so some variation only important if multiple levels of the game are planned. Obviously it makes no difference to the game software, as all instances of a container or ship are instances of essentially the same object.

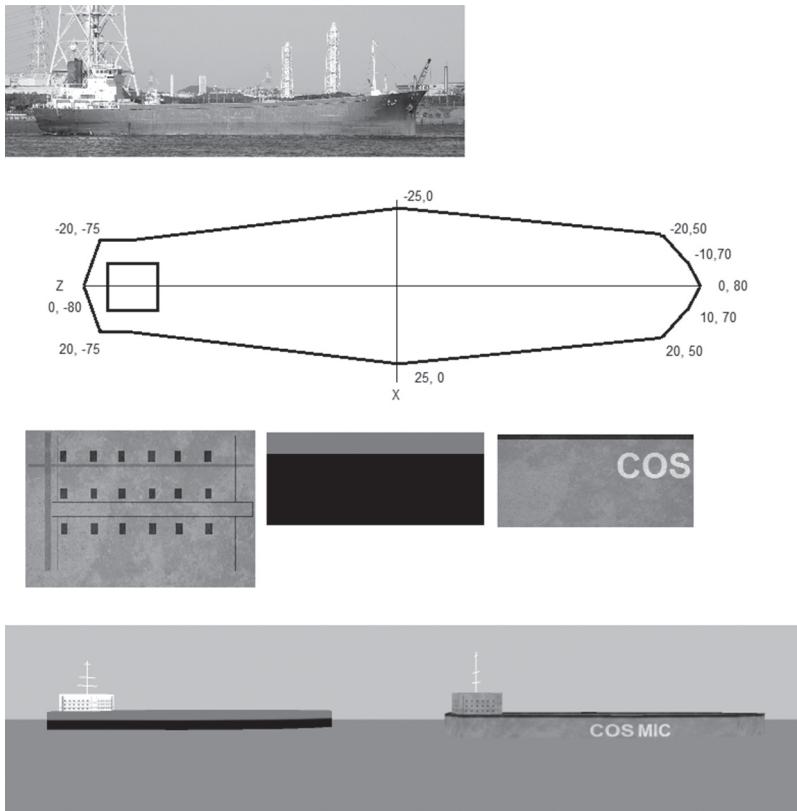


FIGURE 8.5 Modeling a cargo vessel. (a) A typical ocean going cargo vessel. (b) A sketch of a cross-section through the model, showing coordinates of the rectangular panels. (c) Some of the texture images used, some with grunge applied. (d) An example of how this model would appear from a distance on the game screen.

The Bridge

When game play begins, the players find themselves on the bridge of a submarine. It is in this small room that much of the activity takes place. The player sees the bridge in a first person view, seeing the controls, displays, and the window. As with all 3D objects being designed, the first thing

needed is a plan showing what will be included and coordinates. Figure 8.6 shows that plan. The bridge is a two dimensional stencil placed over a 3D view of the underwater scene.

This diagram serves a number of purposes. Coordinates of parts of the bridge could be found so that displays can be placed in their correct location, and sizes of screens and text areas can be measured.

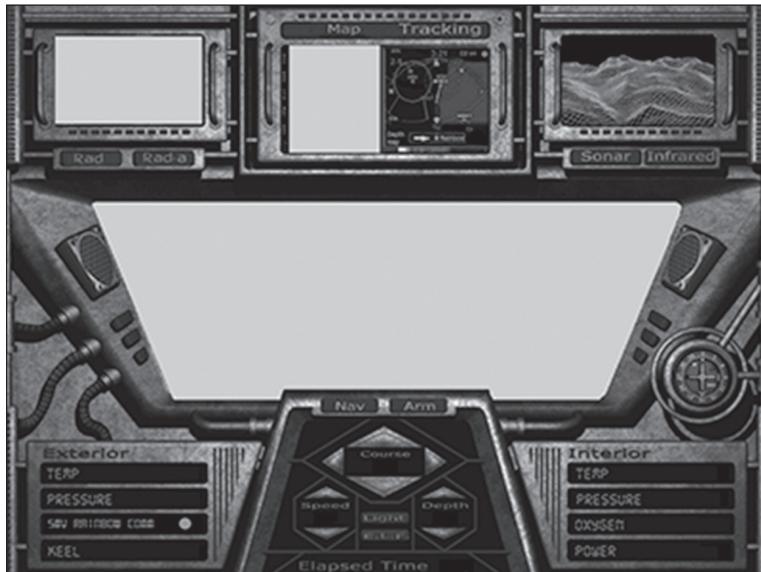


FIGURE 8.6 The bridge of the submarine, showing all displays and indicators.

The sonar display is not truly interactive because it is one way—a passive display taking no input from the player. One of the things that makes this game interesting is that the player has to do most of the work themselves, using feedback from the systems on the boat but not depending upon them to do the work.

The displays are really much like the *green screens* used in motion pictures. Static images can be displayed there, as can animations (which we will cover in Chapter 10). The *heads-up map display* and the *radiation sensor* are essential to game play.

Heads-up Map Display

The submarine has an (X,Y) coordinate where it is presumed to reside. This will be within the map of Figure 8.2, and so both values are in the

range 0-5000. These actual coordinates can be converted into values that can place the sub on a smaller map. All of the other objects can be plotted on this map as well, as they also have coordinates that were used to draw them in the scene viewed by the game. Each container and the vessel have a location in the 3D scene that can be converted into a location on a 2D map. The Y coordinate references the up and down direction in the game, so we simply use the X, Z values of the object to place it on a 2D map. Also, the heads-up display is small: 147×147 pixels, so the positions have to be scaled appropriately. Finally, the vertical coordinate (the Z coordinate here) has to be inverted to represent a map as opposed to a computer image.

So, assume that the sub is positioned at game space coordinate (1000, 30, 557). Scaling this to a 147×147 grid makes it at small map coordinate $(1000 / 5000) * 147, (557 / 5000) * 147$, or pixel position (29, 16). All scene coordinates can be mapped onto map coordinate by using this transformation:

```
mapx = (sceneX/maxSceneX) * maxMapX
mapy = (sceneY/maxSceneY) * maxMapY
```

Then the map view is inverted by setting the vertical coordinate to $147 - \text{mapy}$. This is a typical transformation for scene to small map coordinates in games of all types, for driving games to first person shooters.

Figure 8.7 shows a close-up of this display. The right side is static, merely an ornament. On the right there is a green screen with a set of rectangles (cargo containers) and the outline of the sunken vessel. The position of the *Rainbow* is the white circle, and it moves about the map as the sub moves about the game environment.

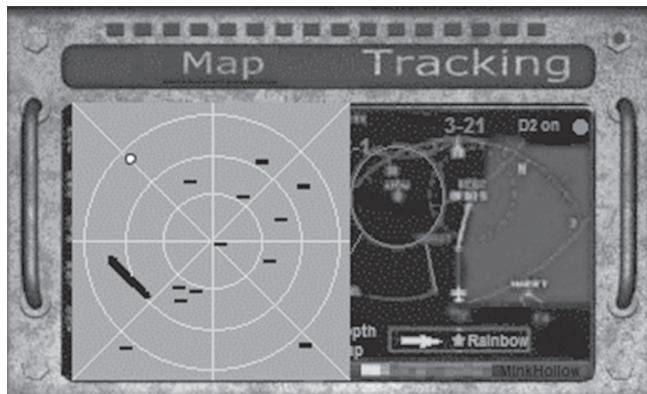


FIGURE 8.7 The heads-up map display. The white circle represents the submarine's current position.

Radiation Sensor

This sensor is the main way that the player detects the target containers. There are two radiation sensors on the ship, on the left and right sides 2 meters from the cabin. Radiation is much like light and sound—it expands from the source in a sphere, and so gets weaker as a function of distance. In fact, it decreases in intensity as the reciprocal of the distance squared. That is, if r is the distance from the source then the intensity is $1/r^2$ of that measured right at the source.

There are two sensors, and sometimes the difference between them indicates the direction to a radioactive container. At all times, the reading on these sensors is the sum of the readings from all sources. There is a numeric value and a colored bar indicating the measurement from each sensor and for the sum of the two sensors, as shown in Figure 8.8.

When the distance between the sub and a radioactive object is less than 200 pixels then decontamination can take place. This is accomplished by typing the X key, at which time the decontamination device will engage and neutralize the radioactive source. This device is pure fiction, and removing radioactivity from anything is actually impossible at this point in history. Many games have fictional components and stretch narrative to be more entertaining. If this game were intended to teach about radioactive contamination then the player would have to be informed of these facts.

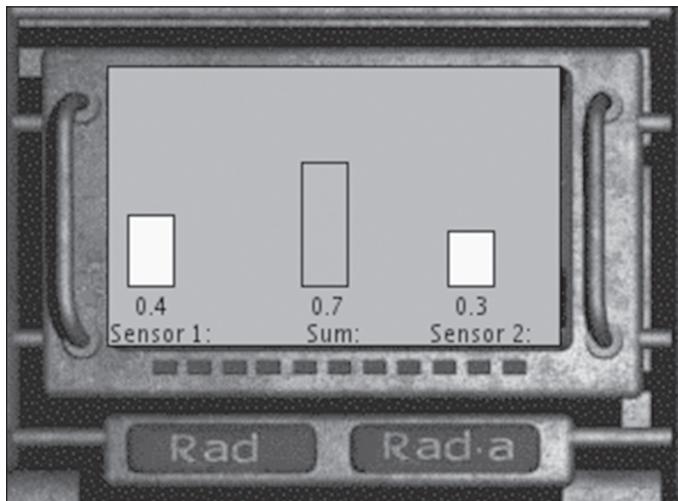


FIGURE 8.8 The radiation monitor. The center bar is the sum of the two left and right sensors, and they change color from blue-yellow-orange-red as the sub nears a radiation source.

The Indicators

The console has a number of numerical displays, or *indicators*, in addition to the graphical ones already described. Simple numerical values appear in these displays at all times, and these values are generally correct ones for the *Rainbow*'s current status. The interior sensors exist simply for show, and change slightly but are not really relevant. The others are calculated from real formulas. They are:

Temp – This is the temperature outside the sub. As the submarine descends the water gets colder according to a known relationship, in general. A rough graph of this relationship is given in Figure 8.9. There is a sharp drop in temperature, followed by a much gentler drop, followed by a steeper one again. The black curve in the figure has been approximate by three straight lines that will be used in the game as an approximation. The code in the game that computes this is in a function named `temperature`, and is (for a depth value `d`):

```
if (d<250) t = 22.0;
else if (d<=750) t = ((750-d)*0.03 + 7);
else t = ((1000-d)*0.012 + 4);
```

The three line segment approximations can be seen in this calculation, and they were extracted from the real graph.

Pressure – As depth increases the pressure of the water above increases too, and quite dramatically. The formula that expresses this is a simple one: every 33 feet (10.06 meters) of depth increases pressure by 14.5 psi (which is 1 *bar*). In the game this is calculated as `pressure = (d/10.06) * 14.5` where depth `d` is in meters.

Keel – This is the distance between the sub and the ocean floor, and is simply `1000-d`, since the ocean floor is very flat in the game and the water is 1000 meters deep.

Course – As discussed in detail in the next section, this is the direction of motion where North is 0 degrees and East is 90 degrees.

Depth – Depth is measured in meters below the surface—the maximum depth is 1,000 meters.

Elapsed Time – The amount of time since the game play was started. This is just the number of frames divided by the frame rate.

Power – In the game this starts out at 100%. Each frame drops the power

by 0.01, and use of the neutralizer costs 15%. When power = 0 the game is over, and the player loses.

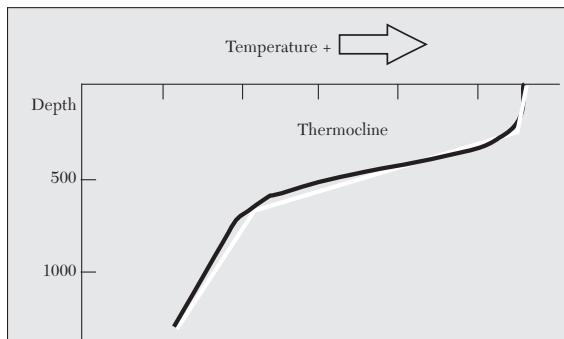


FIGURE 8.9 How ocean temperature changes as a function of depth.

In all cases of displaying a floating point numerical value, the number of digits displayed is an important factor. Displaying too many digits takes a lot of space and the lower digits tend to change wildly. An answer is to convert all displayed values to single decimal place precision. This is done by multiplying the number by 10 and rounding it, then dividing by 10 again. In code:

```
float frac1 (float t)
{
    int it;
    float f;
    it = round(t*10.0);
    f = (float)it/10.0;
    return (float)f;
}
```

All numerical values displayed have been put through this function.

Controlling the Submarine

The keyboard is used to control the motion of the submarine. The arrow keys are used to move forward (up arrow) backward (down arrow) and to turn left (left arrow) and right (right arrow). The W key allows the sub to ascend and the S key makes it descend. As already mentioned, the X key starts decontamination. Typing the up arrow key starts the sub moving forward; it keeps moving until the down arrow key is pressed. The same situation exists when the down arrow key is pressed to move backward, where the sub continues to move until the up arrow key is pressed.

This is a first person game, and as these sorts of games are quite popular it is useful to know how these controls are implemented. Recall that the viewpoint is defined by a call to the *Processing* function **camera**:

```
camera(eyex, eyey, eyez, cx, cy, cz, 0, -1, 0);
```

The position of the player (the submarine) is (**eyex**, **eyey**, **eyez**). So, when the player moves these values will change appropriately. However, it's not as simple as it seems at first because pressing the up arrow key starts the sub moving in the forward direction, whatever that is, and not the X or Z direction. So, what direction is forward?

The sub's motion is specified by an angle between 0° and 360° . The angle 0 is the positive X direction in the game (which is *not* the same as its course). Typing the left arrow changes the angle by 1 degree counterclockwise (add 1), and correspondingly typing the right arrow key changes it by 1 degree clockwise (subtract 1).

Motion is accomplished by adding an appropriate value to the X and Z position during each frame. It's important to note that the Y direction is *up*, and so the sub moves normally in the X-Z plane. Also, for simplicity, the sub moves at only one speed when it moves at all. So, motion is accomplished by adding a value `dx` to the X coordinate of the sub and a value `dz` to the its Z coordinate, where these are determined using simple trigonometry. The values are found as:

```
if (angle<0) angle = angle + 360.0;
else if (angle > 360.0) angle = angle - 360.0;
dx = bigcos(angle);
dz = bigsin(angle);
```

The functions `bigcos` and `bigsin` are just the sine and cosine functions extended to allow for correct values for all 360 degree angles, rather than over the usual range 0 to 90 degrees. Now the position of the sub is updated as follows for forward motion:

```
x = x + 5*dx;
z = z + 5*dz;
```

The values of `cx`, `cy`, and `cz` in the call to camera specify the point at which the player (camera) is looking. This should be straight ahead in this game, and so the values are:

```
cx = dx*20000.0;
cy = eyey;
cz = dz*20000.0;
```

This code would appear in the function `keyPressed`.

Normally, a ship's course is an angle between 0° and 360° , which is the angle between true North and the direction the sub is moving. This is measured in a clockwise direction, so a course of 90° means that the boat is moving East. This is not the same as the angle in the previous discussion, and is simply `90.0-angle`.

Depth is measured from the ocean surface. The *Rainbow* can ascend or descend at 10 pixels per frame, and moves upwards each time the W key is pressed, downward when the S key is pressed.

Illumination

It's dark down there. After a few hundred feet, very little light penetrates the ocean, and at 1,000 meters where *Rainbow* is working it will be absolutely dark. To emphasize this, the game has no ambient light at the sea floor. The sub is equipped with a spotlight that illuminates a small region in front of the vessel. *Processing* has just the function needed for this:

```
spotLight (255,255,255, eyex,eyey,eyez, cx,cy,cz, PI/8, 50);
```

The first three parameters express the RGB color of the light; the next three are the spotlight's position, which will be the same as that of the sub, and the next three give the direction of the light, which will be in the direction the sub is facing. The tenth parameter, which is `p/8` in the example, represents the angle of the spotlight. It will be a cone, and this angle represents how quickly the cone gets broader. The last parameter is a specification of how the light is dispersed within the illuminated cone—how concentrated or spread out it is.

There will be ambient light that increases as the sub rises to the surface. Processing also has a function that can be used for this:

```
ambientLight (eyey*0.085, eyey*0.085, eyey*0.085);
```

The variable `eyey` represents the depth in pixels, and so the light will increase by 0.085 for each pixel rise in depth.

Sound

This game has relatively little sound, and so it was decided to use *Minim* directly. The two sounds are the engines, which are a background sound that loops during game play, and the sound of the decontamination device, which is a sound effect that plays only after the X key is pressed.

The code for the sound is easy to comprehend. Setting up the sound means reading the sound files into their corresponding variables:

```
Minim minim;
AudioPlayer engine;
AudioPlayer decon;

...
engine = minim.loadFile("sound/engine.mp3", 512);
decon = minim.loadFile("sound/decon.mp3", 512);
```

When the **play** screen is entered, the engine sound is played in a loop.

```
if (sound) engine.loop();
```

The variable `sound` is Boolean, and is `true` if sound is to be played and `false` if it was turned off. This is done in the **options** screen. The engine volume is set to 0 in gain:

```
engine.setGain(0.0);
```

This plays it at a low volume. When the ship is moving the gain is increased to 10—this is done where the speed is set, in the `keyPressed` function.

When X is pressed, the decontamination sound is played, if the sound is turned on:

```
if (sound)
{
    decon.rewind();
    decon.play(); // Sound - decontamination
}
```

It is rewound first, otherwise it will only play once.

3D versus 2D Rendering

When the game is started, the player sees the splash screen, and can set parameters, read the rules, quit, or play the game. Selecting play (by clicking on the **play** button) takes the player into the play state, and the game will display the submarine's controls and the 3D scene visible through the window. The keyboard controls can move the ship, and the view of the scene through the window as it changes.

The main game loop draws all of the three dimensional objects in their relative places in the display volume. Naturally the player cannot see them

all—some are behind the sub and some are not illuminated. They come into view as the sub moves, or course. Any change in position changes the parameters to the `camera` call that displays the view from the player's perspective.

During each frame, the relative positions of all objects are computed and rendered, the radiation detected by the sensor is calculated and displayed, the position of the sub on the HUD map is updated, and all numerical values associated with the control room display (i.e. depth and temperature) are calculated and displayed in their proper positions. There remain some outstanding issues, though.

3D versus 2D Screens: Popups

The ship and containers are 3D objects, and the user can navigate through the undersea world in 3D while observing the volume from the bridge. The map is a projection of the 3D world onto 2D, and the other displays are 2D as well, being images and text. It turns out that this presents a problem. The player is on the bridge looking at the 3D view through the window, is being presented along with 2D displays. Unfortunately, a 3D renderer is always expecting 3D coordinates, and the view on the window will at best be a projection of a textured object on a plane drawn in three dimensions. This means that the coordinates of the widgets on the screen will not agree with the coordinates of the same item on the image. It's also tricky to get a good perpendicular view of the map screen image, since it must be drawn in 3D and projected so that it is perfectly rectangular and fills the correct space in the window.

In other words, drawing the map screen as an image using the usual call such as `image (map, 0, 0)` will draw an image somewhere in the volume of the bridge near the origin. Unless the player happens to be looking directly at that point it will not even be seen.

A solution that works very well but is not initially obvious is to have two renderers operating—a two dimensional renderer for the user interface and flat displays, and the three dimensional renderer for the scene outside. We learned about the `PGraphics` object in Chapter 5, and a pair of these will be used to hold the views being rendered. Create two `PGraphics` instances:

```
PGraphics g2;
PGraphics g3;
...
size (800, 600, P3D);
```

```
g3 = createGraphics (800, 600, OPENGL);
g2 = createGraphics (800, 600);
```

The undersea scene will be drawn into the `g3` object. The 2D bridge display is drawn, along with a transparent window and all screens and controls, into the `g2` object. The `g3` is drawn first, then the `g2` so that the correct portion of the undersea scene can be seen through the window.

```
void draw ()
{
    Image (g3, 0, 0);      // Render 3D scene
    image (g2, 0, 0);     // Render 2D scene
}
```

The control room display is essentially an image over which the other displays are drawn. It is a GIF, and the window area is a transparent color. This is essential—if the window is not transparent, nothing can be seen of the scene outside. The call to `camera` renders a 2D version of the 3D scene but this is displayed first, followed by a display of the controls. Figure 8.10 illustrates how the game screens are rendered, in terms of the 3D transformation and the `g2` and `g3 PGraphics` images. All of the screens, such as `rules` and `options`, are drawn into `g2` only and displayed only in 2D.

Any pop-up messages such as *YouWin* turn out to be tricky to implement in a 3D space. Text is drawn as 2D in the x-y plane. It is tricky to draw text so that it can be read from any point the user's avatar may be looking. So, an easy way to make these messages work is to draw them in 2D on the screen after the 3D `PGraphics` image has been rendered and displayed.

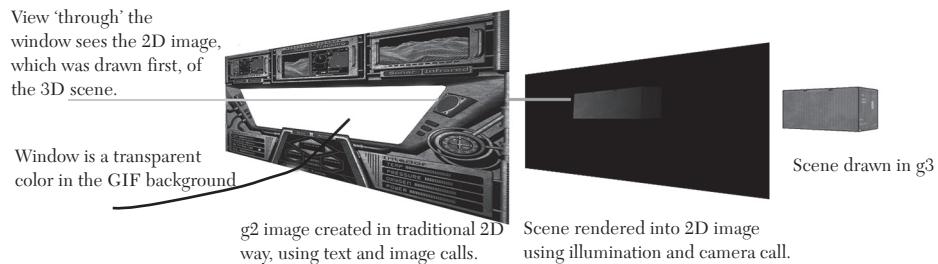


FIGURE 8.10 Rendering the two parts of the game scene, the 3D undersea environment (which is drawn ultimately to a 2D image) and the 2D control room window. G3, the 3D scene, is drawn first, then the 2D control room (g2) over that. The scene is seen through the transparent window of the control room, which is a GIF background image.

Summary

The 3D game presented here needs more work needed to make it really entertaining, but the technical aspects of it are representative of those found in many 3D games. The player must navigate in three dimensions, unlike games such as Hockey Pong, and can move around objects to see the other side. This means that objects in the game must be rendered in three dimensions most of the time.

It is important to define the area/volume within which play will take place and to define the operations, especially motion and views, which can result from game play. Once objects are well defined in 3D, many viewing operations become trivial, and can be implemented by simply positioning the viewpoint at the right place and/or rescaling the viewing transformation. Illumination can be specified as ambient or as a spotlight, in any color. Mixing 2D and 3D rendering involves creating two graphics contexts and drawing each one separately.

A game designer has to know everything. That is, the game is not just about computer graphics and sound, but also about how pressure and temperature change with depths and how submarine sonar systems work. Whenever possible, the game should portray a realistic view of the situation being modeled.

Exercises

The exercises below will apply your knowledge of the material in this chapter, and sometimes require that you do some more research before you are able to complete them. Solutions to selected exercises are available on the companion disc.

1. Discuss ways that the game could include a terrain model that would limit diving to various depths. How difficult would this be to implement?
2. The game currently does not restrict the position of the submarine to the area defined by the game. Modify the code to do so.
3. Currently there are four radioactive containers, and they are always the same ones. How could the game be modified so that a distinct set of containers would be targets each time the game was played? How could the number of containers on the sea floor be made variable as well?

4. The sea floor in this game is rather uninteresting. Game designers frequently decorate the scene with objects typically found in the environment. In this case it would be undersea plants, sponges, and the like. Create two such decorations and add them to the game.
5. Many military submarines can fire torpedoes. The *Rainbow* cannot, and it is on a peaceful mission, but describe the modifications needed to the game to allow a torpedo to be fired.
6. The *Rainbow* can reach the surface but nothing happens when it does. What *should* happen? What would be involved in making it so?

Resources

Sites for downloading inexpensive images for use as textures:

- <http://www.rgbstock.com/>
- <http://www.dreamstime.com/>
- <http://www.mayang.com/textures/Metal/html/Flat%20Metal%20Textures/>
- <http://webtreats.mysitemyway.com/>

Grunge textures: <http://www.grungetextures.com/>

Earth and Space Research – Definition of thermocline. <https://www.esr.org/outreach/glossary/thermocline.html>

Bibliography

- Dunn, F., and I. Parberry. *3D Math Primer for Graphics and Game Development*. Second edition. Boca Raton, FL: Taylor & Francis Group (A.K. Peters), 2011.
- Foley, J., A. van Dam, S. Feiner, and J. Hughes. *Introduction to Computer Graphics*. Addison-Wesley Professional, 1993.
- Gortler, Steven J. *Foundations of 3D Computer Graphics*. Cambridge, MA: MIT Press, 2012.
- National Oceanic and Atmospheric Administration. *Pressure increases with ocean depth*. n.d. <http://oceanservice.noaa.gov/facts/pressure.html> (accessed 2014).

- Parker, J.R., S. Chan, and J. Howell. "OceanQuest: A University-Based Serious Game Project." *Digital Games Research Association Conference 2005*. Vancouver, BC: DiGRA, 2005.
- Spence, R. *Information Visualization*. Boston, MA: Addison-Wesley, 2001.
- US Navy Museum. *Elements of Submarine Operation*. n.d. <http://www.history.navy.mil/branches/teach/dive/elem.htm> (accessed 2014).
- Ware, C. *Information Visualization – Perception for Design*. San Francisco, CA: Morgan-Kaufman, 2000.

CHAPTER 9

THE WEB AND HTML5 GAMES

In This Chapter

- An overview of HTML, HTML5 and JavaScript
- How to move a game to the Web

The World Wide Web is a remarkable system for delivering multimedia content into people's homes. Initially, of course, it was used for text documents, but very quickly images were allowed, then sounds and video, and finally games. Now a majority of Americans play browser-based games. The reasons are pretty clear: they are almost always free, and don't need special hardware; they usually don't need special software, and when they do it is free and downloadable (*Flash*, for instance); they run on any device with a browser, including phones and tablets; and they often allow a player to play with friends and family.

Processing has an extension, *Processing.js*, that permits most *Processing* programs (including games) to run through a browser. There are a few constraints on the developer and things that can cause problems if proper attention is not paid, but from the perspective of the player there is nothing special that needs to be done to play a *Processing* game through a browser.

We're going to learn how to make browser-based games. We'll start with a short discussion of how the Web works and how content is delivered to your computer, we'll look at what the new HTML5 standard offers, look at *Processing.js*, and then convert *Hockey Pong* into a Web-based game to show what the differences are between *Processing* and *Processing.js*.

How the Web Works

The Web is a collection of communication and presentation software layered over the Internet. The Internet connects individual computers through a hardware based network. Little packages of information are sent along wires or radio links to computers that are either their destination or which send them on to the next link in a chain of computers. All of the computers along the chain can see these packages (called *packets*) but there is an address inside the packet that identifies the actual destination, and only that computer is supposed to look at it.

Now consider a packet or a set of packets that describes a document: text, images, and perhaps sound, programs, and video. When received, this document can be displayed on a screen in the intended manner, but until interpreted correctly, it is simply more data in the network. A program that displays these data as documents can be called a *browser*. Creating and sharing the documents across the Internet as packets is done by special software/computer systems called *servers* (more properly *Web servers*) and protocols have been devised for the servers and browsers to identify any desired document page/site in the world—HTTP and URLs. This is the World Wide Web.

From the perspective of the user, the Web is a computer window that can be linked to documents anywhere on the Internet. From the perspective of the browser, the Web is a large distributed collection of data. When the user requests to see a particular *page* (document) the browser requests that a copy of that page be sent to the user's computer. This could require many packets, but the browser organizes them and then, like Word or PowerPoint, builds a graphical representation of the document and displays it.

The important issue here is that most of the work is done on the user's computer—this is called *client side* processing, and means that the files are downloaded (copied from the network to the user's computer) and then the processing is done locally. When a video is displayed—or, more relevantly,

when a game is played—it is played on the user's computer and not normally on a remote one. There are some exceptions, but this rule applies in the vast majority of situations.

What does this mean for a game developer?

Firstly, that the game will respond differently depending on who is playing it and the computer they use. Many computers on the Web are relatively old and can't handle intensive calculations or advanced graphics.

Secondly, the user chooses the browser they will use, and browsers all behave differently. They display colors differently, can view different sets of video and sound files, conduct graphics display differently, and so on. For a game to run on the Web, the differences must be known and taken into account.

Thirdly, the code, graphics and sound associated with the game have to be downloaded to the user's computer. A developer needs to be careful about the size of the asset files the game needs, and must also be cognizant of the size/quality tradeoffs of the various file formats and compression schemes. Download speeds will vary depending on the user's connection, but the game and its developer will be blamed for slow response and especially for using features that are not functional on the user's machine.

Web documents consist of text and embedded code. The basis of the Web for many years has been *HTML* files with *JavaScript* scripts and plugins for games like *Flash* and *Unity*. Let's look at the structure of Web documents before moving on to design and coding issues.

HTML and HTML5

There are now many ways to create Web pages, but still the most well-known and the most common is the *HyperText Markup Language*, or HTML. This language can describe how a document is to appear on a page, normally a Web page but it is a general document description language. A document consists of text, images, sounds, and videos organized in a specific manner. The formatting of the document is done by inserting tags into the text. For example, consider:

```
<b> Hi there, folks! </b>
```

This would result in the text being displayed in bold characters. The tag `` means *set the following text in a bold font*, and the `` tag indicates the end of the bold text. There are text formatting tags (italic, font size, color),

document formatting (line break, headings, paragraphs) and settings for the browser to understand (header, body, character set). HTML is well known, and there are many places to learn more about it. (See the **Resources** section in this chapter for some suggested tutorials.)

A *Web page* is a document that can be described using HTML in a specific way. There is a *header* that tells the browser and server about the contents of the Web page, and there is a *body* that contains the text and formatting, including images and sounds, which is the purpose of the document. The browser downloads the web page (HTML file) through an Internet connection, parses it, and displays it on the local computer. All of the work involved in document display is done on the viewer's computer; this is an important fact that defines how files and other bits of data are retrieved and displayed.

A simple Web page would have the following HTML source code:

```
<html>
<head>
    <title>
        This appears in the title bar of the browser window
    </title>
</head>
<body>
    <h1> This is a heading </h1>
        This is text that represents the content of the page.
        It is formatted by the programmer, who inserts tags
        into the text. <br>
        The <br> tag creates a new line so that a
        paragraph can begin. One can
        also insert images and sounds into the page.
</body>
</html>
```

This source code is displayed by a browser as the image seen in Figure 9.1. The browser reads the files, interprets the tags as commands, and formats the text much as the *Word* or *WordPerfect* would, creating an image that can be displayed on the screen.

Note the separation of the document into a *head* and a *body* using tags. The *head* is small in this document, but it can contain a lot of specifications concerning the document including locations of extra data (files) and scripts (programs).

HTML is far too large to describe completely here. It is important to understand how HTML works at a basic level, because web-based games are set up using an HTML document. It is possible to create and deploy *Processing* games using predefined and packaged HTML code, but of course it's always better to understand what is going on.

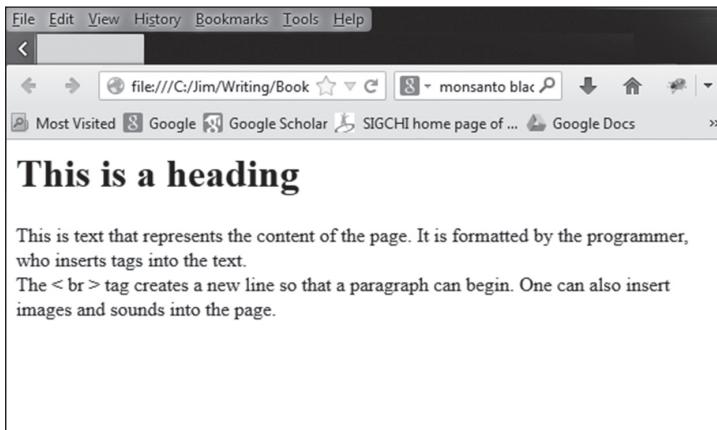


FIGURE 9.1 The example web page displayed by the *Firefox* browser.

JavaScript

JavaScript is a programming language developed to allow web pages to be more interactive than permitted by pure HTML. It is a more lightweight language than *Java* or *C++*, and is interpreted, which allows a great deal of portability. Its syntax is strongly influenced by *C*, and therefore looks like *Java* as well.

While it is certainly possible to write any computational program in *JavaScript*, its purpose is to interact with the user on the Web. So an important capability of *JavaScript* is to load a web page (new content). The code for this could be:

```
<html>
<head>
  <title> JavaScript example 1 </title>
</head>
<body>
  This is a simple web page demonstrating JavaScript.
  <script>
    window.location="http://www.google.com";
```

```
</script>
</body>
</html>
```

The script (program) itself is contained between the tags `<script> ... </script>`. Scripts can also reside in files and referenced in a web page using the file name.

Proper use of *JavaScript* requires that the programmer become familiar with a large collection of library routines and built-in variable names, just as you need to do the *Java* or *Processing*. In this case, the variable `window` represents the current browser window, and `location` is the URL of the page it is displaying. Setting `window.location` to “`http://www.google.com`” sets the browser to point at Google, and the Google home page will be displayed. Of course, the user could have done that in the first place, and this is a basic example.

Giving the user a choice is very much what *JavaScript* coding is about. A program that offers such a choice about whether to visit the Google site would might the following body:

```
<body>
This is a simple web page demonstrating JavaScript.
<script>
if (confirm("Click to go to Google"))
    location="http://www.google.com"
</script>
</body>
```

The *JavaScript* function `confirm` opens a decision box on the screen, and a yes/no decision can be made with the mouse. Figure 9.2 shows this screen. `Confirm` does all of the work involved with drawing the box as a graphic, displaying the text buttons and handling the mouse events, much as the function `rect` in *Processing* draws a rectangle for you. As you can see this is a very useful language for creating Web pages.

JavaScript is a language as powerful as *Java*, and entire books have been written to teach it. We’re not going to learn much *JavaScript* here, but it important to know a few things about it, and of course *Processing.js* will communicate with *JavaScript* code and HTML5 scripts in Web pages. *JavaScript* has functions, variables, if statements, loops, and lots of syntax. Fortunately, if we use *Processing.js* we don’t have to learn *JavaScript* as well.

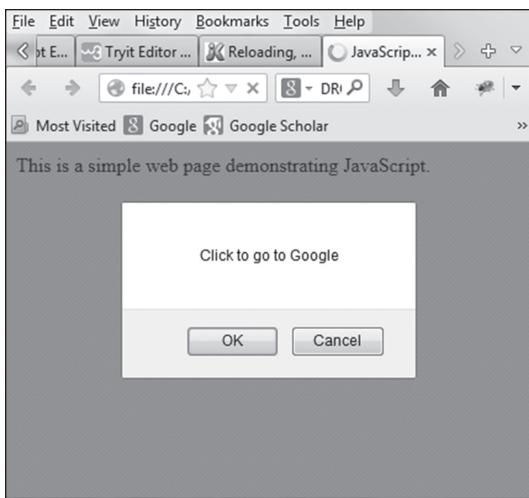


FIGURE 9.2 A decision box opened by *JavaScript* function `confirm`.

HTML5

HTML5 is the most recent HTML standard, and contains some new tags that specifically support multimedia display and interaction on the Web. An HTML5 script can use most of the old HTML tags, so it is *backwards compatible*—a terrific feature. The HTML5 standard adds about 30 new features to the standard, removes another dozen or so (including **font** and **center**), and adds a large number of new attributes. The most important new feature is the **canvas** element.

The **canvas** is used as the container or display element for live graphics, and as such is essential to computer games. Normally the canvas is drawn into by a script, usually *JavaScript* – the canvas is simply the place where things are drawn, and contains nothing by default.

A basic use of the canvas tag would look like this:

```
<canvas id="myCanvas">
```

Your browser does not support the HTML5 canvas tag.

```
</canvas>
```

The message will appear on the page if the canvas feature is not supported by the browser, meaning that it really does not support HTML5. This is rare these days. This code creates a canvas and gives it a name that can be used by *JavaScript* (or *Processing*) to draw in it. More details on the **canvas** are hard to present before showing how to draw in it, which requires a

script. However, the **canvas** will be the place where any Web-based game we develop will be drawn and played.

Other new multimedia tags in HTML5 are:

- <audio> Sound/audio content. Music, voice.
- <video> Video content. Movies, animation.
- <source> Allows sound or video content to be specified as a variety of source file types; MPEG and AVI, for example.
- <embed> Defines a container for an external application, like *Flash*.
- <track> Allows the user to specify subtitle or caption files that should be displayed when the media file plays.

Most of these tags are not relevant to game development.

Processing.js

Processing.js is an interesting piece of software. It takes a *Processing* program and converts it into pure *JavaScript*. Each *Processing* function native to the API is mapped to a corresponding and equivalent function in the final *JavaScript* code. The resulting program draws into an HTML5 canvas instead of to a sketch window, and creates the same effect as it would in a sketch window—with some constraints. These constraints are created by the fact that *JavaScript* is not *Java*, and the *Java* libraries are not directly available. Thus, objects like the *Minim* sound library are not available in *JavaScript*, and so can't be used from *Processing.js*. Also, there are differences in the way the two languages work, most importantly in the fact that *JavaScript* has no integer type.

Using *Processing.js* requires three components all working together: an HTML5 Web page, *Processing*, and *JavaScript*. The first thing that is required is the download of *Processing.js* itself. This is done from the Website <http://processingjs.org/download/>, and there are a numbers of choices—download them all. The key file for deployment is *processing-1.4.1.min.js*, and for development is *processing-1.4.1.js*. These are *JavaScript* programs that do the work of converting *Processing* to *JavaScript*.

The second component is a *Processing* program. Many have been written as examples and problems so far, and any of them (except for the audio

examples) should work. These will be converted into *JavaScript*, and so it can't depend on any external *Java* libraries.

The third part of a *Processing.js* program is a Web page. This page can be pretty simple, but has to include a link to the script file *processing-1.4.1.min.js* in a `<script>` tag, needs to create a canvas for displaying the game, and must link the *Processing* program, a *.pde* file, to the canvas as an output device.

Much of this can be done as formulas or patterns. The Web page can be very standard, with only a few substitutions. The only item that is really in the hands of the developer is the processing code.

Making a *Processing.js* Program Work

Let's make a simple example.

1. Create a directory named *example01*, and copy the downloaded file *processing-1.4.1.min.js* into it.
2. Now create an *index.html* file that contains the following:

```
<html>
<head>
<title> Example 1: Processing.js </title>
<script src="processing-1.4.1.min.js"></script>
</head>
<body>
<br><br>
<h1> First Processing.js Example </h1>
<center>
<canvas data-processing-sources="example01.pde"> </canvas>
</center>
</body>
</html>
```

The key aspects of the HTML5 page above are the ones in bold text. The `<script>` tag specifies the *processing-1.4.1.min.js* file as containing *JavaScript* code to be associated with this page. The `<canvas>` tag creates a canvas and associates the file *example01.pde* as data for it. This is an as yet uncoded *Processing* program that will be used as data for the *Processing.js* interpreter, and can be one of a great many different *Processing* programs. The rest of the HTML code sets up the Web page and displays some text.

3. The *Processing* program should be simple, since it's a first example.
Let's do a simple drawing a circle and a rectangle:

```
// Example 00 - Demonstration of Processing.js
void setup()
{
    size (200, 300);
}
void draw ()
{
    background (128);
    fill (0);
    ellipse (120, 60, 100, 100);
    rect (100, 180, 90, 110);
}
```

This sketch can be entered and executed using the standard *Processing* environment and saved. To display the result, double click on the *index.html* file, or otherwise have it displayed by a browser. Figure 9.3 shows the result displayed in a *Firefox* window.

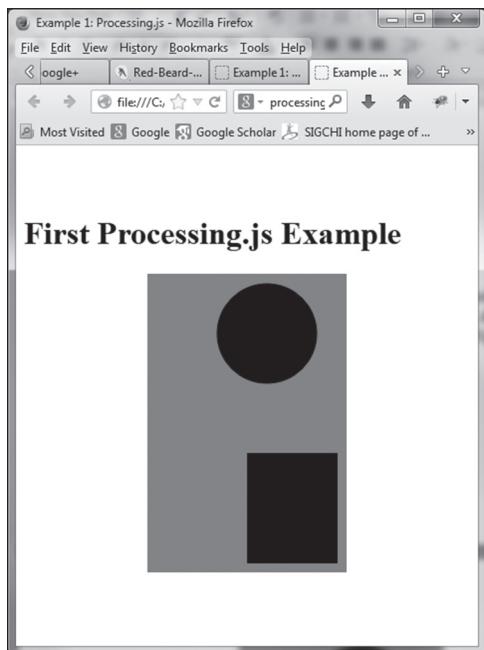


FIGURE 9.3 Our first *Processing.js* program as it runs in the Firefox browser.

To execute a different *Processing* program in a Web page, use the same HTML file, but substitute the name of the new program for the name *example00.pde* in the **canvas** tag. That would normally be the only change required unless a new *JavaScript* library was also required.

Seeing this work on such simple code only starts the process of learning how to make a Web-based game. Let's convert *Hockey Pong* into a Web game and see what kinds of problems arise.

Hockey Pong

Hockey Pong is over 1,300 lines of code, if one includes the sound library. This should be a sufficiently complex example to expose a large collection of features and problems in *Processing.js*. The first thing to do would be to try and run *Hockey Pong* in *Processing.js* just as-is and see what happens. Sometimes (frequently) an existing *Processing* program will just work correctly in *Processing.js*. So, we create a directory to hold the game—for historical reasons it will be called *proto3*—and copy *processing-1.4.1.min.js* and the source code for *Hockey Pong*, *proto3.pde*, into that directory. Now create a new *index.html* file that contains the following:

```
<html>
<head>
<script src="processing-1.4.1.min.js"> </script>
</head>
<body>
    <canvas data-processing-sources="proto3.pde">
        (Your browser doesn't support canvas)
    </canvas>
</body>
</html>
```

This HTML5 source file is nearly identical to the previous one. Running the program means double clicking on the HTML file, which invokes the browser and ... nothing happens. The browser window opens, but is blank—no game, no sound, no errors. This is quite disappointing.

A problem for developers of *Processing.js* code is that browsers are meant to be used for browsing, not so much for code development. They are the host for our game code, but do not deliver error messages or any other assistance, at least not by default. Almost anything could be wrong and the browser won't tell us. The *Processing* code works, though—you can

double click on *proto3.pde* and the game will play.

What's happening, at least the first thing that we need to recall, is that *Processing* is *Java*-based, and does not have its own sound library—it uses *Java*'s. This won't work in *Processing.js*. In fact, no additional *Java* library can be used by default with *Processing.js*. So step one in the conversion is to comment out all of the audio calls. Other external libraries may be used to display and capture video, interface with external devices, and communicate with the Internet, but for now get rid of any *Java* library calls.

Rule #1: Remove all calls to external libraries.

Now we save the *Processing* file, close the browser page, and again double click on the HTML file. Sadly, again, nothing seems to happen—except sometimes when using *Firefox*. One way to get a hint about what is happening is to use the development tools of the browser. They all have them, but are accessible using different means:

- From *Chrome* (Ver. 34.0.1847.116) click on the three short horizontal bars on the right side of the window near the web address bar; then click on **Tools** on the popup window, and then on **Developer Tools**. The bottom portion of the browser window will become a developer's console.
- From *Firefox* (Ver. 29.0.1), select from the **Tools** menu item the **Web Developer/Web Console** item.
- From *Internet Explorer* (Ver. 10.0.12), click on the gear icon next to the address bar, then click on the menu item **F12 developer tools**.

A list of error messages should appear on the developer's console, and they vary from browser to browser. From *Chrome*, for example, we get:

Uncaught Processing.js: Unable to load pjs sketch files: proto0.pde ==>
XMLHttpRequest failure, possibly due to a same-origin policy violation. ...

If we look up “same origin policy” we find out that, if two pages have the same *origin* then they have the same protocol (e.g. HTTP), port, and host. For security reasons, pages that do not have the same origin can't access certain files or run scripts. Otherwise a script could run and modify files or settings without constraints. Poorly written or malicious scripts can cause damage to a computer system. The problem is that when one double clicks on an HTML file, the browser will open and display it but the origin can't really be determined, as there is not really a port or hostname as such. That's simplistic, but is the basis for the error.

This kind of problem is common, and can be solved in a couple of ways. You could run a local web server from the PC on which you are developing. (See XAMPP Web server in the **Resources** section of this chapter.) This is a complex solution for beginners. Another choice is to upload your files to your Web server—you'll have to have one to deploy the game anyhow—and then try to access from there. When you do, you will find that the game displays on the screen and seems to work.

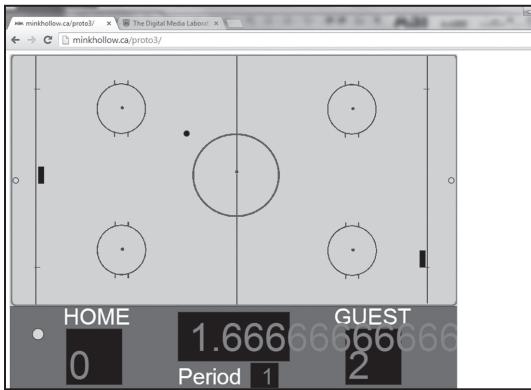


FIGURE 9.4 The timer is displaying improperly, as a real number with many digits.

There are two visible problems: the time clock displays incorrectly, and there is no sound. Figure 9.4 shows the problem with the clock. The time is displayed with a huge number of decimal places. The issue is that the time is expressed as an integer.

Integers in JavaScript

The problem with the timer is that the time is expressed as an integer number of minutes and seconds, and *Processing.js* becomes *JavaScript* which has no integer type.

Integers in *Processing* become floating point numbers in *JavaScript*. This is very important to know, because integers are used quite often in programs and the lack of integers and their associated properties can cause unexpected problems while converting code to run on the web. Integer division does not occur in *JavaScript*, so dividing 3 by 2 in *Processing* yields 1; in *JavaScript* it yields 1.5. For example, some programs use division by powers of two as a right shift. This won't work in *JavaScript*. The *mod* operation is available, so $3 \div 2$ is 1, but the result is real.

The function `frac1()` that was written in the previous chapter does not work in this environment, and so can't be used to fix the clock problem. On the other hand, the game clock is really a different kind of thing from the timers that are used to time sections of the game, and so the game was modified to use a new class named `gameClock` that counts down from an initially specified number of minutes and seconds, and can return a count of minutes and seconds remaining. This works because the only operations are addition and subtraction, which should work perfectly for floating point values of minutes and seconds. This new class has the following operations:

```
class gameClock
    gameClock (float m, float s) // Constructor
    void start (float m, float s) // Start a new countdown
    float getMinutes() // The 'minutes' portion of time
    float getSeconds() // The 'seconds' portion of time
    boolean finished () // Is the countdown complete?
    void tick() // Tell the clock that a frame has passed.
```

Draw calls `tick` every frame. The timer is set to 2 minutes 0 seconds for a period, which is complete when `finished` returns true. The values returned by `getMinutes` and `getSeconds` are fine for printing now, and the problem has been solved.

Preloading

Sometimes images that are loaded into a game and used as backgrounds or textures need to be *preloaded*. Reading an image over the Internet can require a significant amount of time to a computer, and in *Java* images are often read in parallel to the rest of the program execution. This means that the image might be referenced before it has been completely read in. To prevent this we download the images in advance so that the `loadImage` function takes much less time.

Preloading is done by placing a directive at the very beginning of the program:

```
/* @pjs preload="a.jpg,b.jpg"; */
```

This directive preloads two image files, *a.jpg* and *b.jpg*. If this is not done, then the images will often not appear in the game. Textures, for example, will not be displayed and backgrounds will be absent.

Sound in *Processing.js*

The `simpleAudio` class that was developed in Chapter 3 depends on *Minim*, and *Minim* is a *Java* library. We can't use a *Java* library easily from *Processing.js*. Fortunately, a solution to the problem is to use a package that emulates portions of *Minim* in *JavaScript*. *Processing.js* can call *JavaScript* functions directly because the code will be converted into *JavaScript* and interpreted together with the other libraries on the Web page.

Download the *Minim JavaScript* emulator from:

<https://github.com/Pomax/Pjs-2D-Game-Engine/blob/master/minim.js>

The file *minim.js* must be placed into the directory where the sketch is. Now change the header of the *index.html* file so that the new library is included, like this:

```
<head>
<script src="processing-1.4.1.min.js"> </script>
<script src="minim.js"> </script>
</head>
```

The documentation for the library is brief, but makes clear the fact that not all functions of *Minim* are implemented. It contains the following:

```
play()
loop()
pause()
rewind()
position()
cue()
mute()
unmute()
function canPlayOgg()
function canPlayMp3()
```

The `AudioPlayer` object is provided by *Minim.js* as well. Otherwise the *JavaScript Minim* library works in the same way as does the *Java* version—we declare a *Minim* variable and make an instance:

```
Minim z;
z = new Minim(this);
```

Each sound is an instance of `AudioPlayer`, and needs a variable that needs to be instantiated, for example, a referee's whistle:

```
AudioPlayer whistle;
whistle = z.loadFile("sounds/sfx-whistle.mp3");
```

Now the whistle can be played by calling `whistle.play()`.

Modification to the game involves looking for all calls to the sound library `simpleAudio` and replacing them with equivalent calls to `Minim.js`, and when no equivalent exists to remove the call altogether. When this is done, save the sketch and upload to the Website. (Don't forget to upload the new version of `index.html`.) Enter the URL of the game into the browser and ... no sound.

In fact, sometimes it will work—it depends on the browser you are using. A sad fact of the Web for developers is that each browser behaves differently in many regards, and in particular each has implemented sound players for different sound file types. *Firefox* and *Chrome*, for example, can play Ogg *Vorbis* files (.ogg), whereas *Internet Explorer* plays MP3 files. Naturally, we'd like the game to be playable on nearly any browser.

First, remember that *Processing.js* can call *JavaScript* code, and *JavaScript* can detect the kind of browser being used to display the page. This is a good example because not only can we detect the browser type, but we can call any *JavaScript* code, and there are a lot of really useful functions available. A simple way to find out the browser is to use the variable `navigator.userAgent` (look up this *JavaScript* variable on the Web). This is a string that holds the *apparent* name of the browser—*apparent* because developers can spoof *Firefox*, for example, into believing that it is *Explorer*. For the major browsers, here is the value of `navigator.userAgent`:

FireFox	“Netscape”
Chrome	“Netscape”
Internet Explorer	“Microsoft Internet Explorer”
Opera	“Opera”
Safari	“Netscape”

The value “Netscape” is returned for three browsers in the list. Some suggest that a better way to browser detect is to use `navigator.userAgent`:

Firefox: “Mozilla/5.0 (Windows NT 6.1; rv:26.0) Gecko/20100101 Firefox/26.0”

Chrome: “Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.36(KHTML, like Gecko) Chrome/31.0.1650.63 Safari/537.36”

Internet Explorer: “Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.1; Trident/6.0; SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR 3.0.30729; Media Center PC 6.0; Tablet PC 2.0; .NET4.0C; .NET4.0E; InfoPath.3)”

Opera: “Opera/9.80 (Windows NT 6.1) Presto/2.12.388 Version/12.16”

Safari: “Mozilla/5.0 (iPad; CPU OS 6_0 like Mac OS X) AppleWebKit/536.26 (KHTML, like Gecko) Version/6.0 Mobile/10A5355d Safari/8536.25”

The `userAgent` value is much more detailed, giving versions and compatibilities. It’s harder to figure out what actual browser you are using, but that does not really matter so long as we can play the sound we want, at least in this instance. Many Web developers suggest checking for specific browser capabilities rather than asking what the browser is. This is a good idea in general, but does not always give the best result in specific cases, sound being one of those. For example, the following code will print the result of calling *JavaScript* functions for determining sound player capability:

```
var audio = new Audio();
println (audio.canPlayType("audio/mpeg"));
println (audio.canPlayType("audio/ogg"));
println (audio.canPlayType("audio/wav"));
println (audio.canPlayType("audio/webm"));
The result for Firefox was:
maybe
maybe
maybe
probably
```

This is not especially useful information. Thus, the code that is used in *Hockey Pong* determines the browser first, then selects a sound file type to use.

```
if(navigator.userAgent.toLowerCase().indexOf('safari/')>-1)
    loadMP3();
else if (navigator.appName.equals("Netscape"))
    loadOgg();
else if (navigator.appName.equals("Microsoft Internet Explorer"))
    loadMP3();
else if ... // Other browsers?
    .
.
else
{
    println(""+navigator.appName);
    soundOn = 1;
}
```

So: if we're using a *Netscape* compatible browser (*Firefox*, *Chrome*, etc.), the function `loadOgg` is called; this function loads the complete set of sound files in *Ogg Vorbis* format. If, on the other hand, we're using *Internet Explorer*, then the *MP3* versions of the sound files are loaded (function `loadMP3`). This can be done for any browsers that we like, and the final code turns the sound off if the browser being used is not known. This code resides within `setup`. There is a special test for *Safari*, because it won't play `.ogg` files but `appName` is **Netscape**.

Of course, this means that all of the sound files must be available in all relevant formats. Normally we'll need copies as `.wav`, `.mp3`, and `.ogg`, and the *Audacity* program that was used in chapters 3 and 4 can create all of these formats. Keep each set of sounds within their own sub-directory of the **sounds** directory: **sounds/mp3**, **sounds/wav**, and so on. Using the code above the game will only download the sounds that it can use.

The code at this point is known to work, with sound, from *Firefox*, *Explorer*, *Chrome*, *Safari*, and *Opera*. *Safari* was especially tricky, and required extra code and testing.

Fonts

A minor error that may not be noticed is that the font chosen to be used on the scoreboard does not display properly. The text is displayed, but a digital clock font (named *Quartz*) was selected and the *Ariel* font is actually used. The problem is that *Processing.js* uses fonts a bit differently. Here's how to make a font work properly.

First, look for the True Type font file that you need. In this case it's `quartz.ttf`, and a search on the PC located it—copy the file into the sketch directory. Now modify the `loadFont` call appropriately. Instead of

```
loadFont ("Quartz-Regular-48.vlw");
```

we need to call:

```
loadFont ("quartz.ttf");
```

Finally, preload the font file just like we preload images. A comment at the beginning of the `.pde` file:

```
/* @pjs font="quartz.ttf"; */
```

Now the font will display properly. Any font for which a True Type file exists should be able to be used. Many True Type fonts can be downloaded from the Internet, and *Quartz* is one of them.

More JavaScript

There are aspects of web-based games that can profit greatly from *JavaScript's* capability for working the Internet. Many games have a set of rules and a screen that explains them. It is equally possible to use a Web page to display rules and have the game pop up a new browser window, or move from the game page to the rules window. Warning: some experience with *JavaScript* is useful here.

A call to `window.open()` will create a new browser tab, and a URL passed as a string parameter causes the browser to open that page. If you place such a call in the *Processing* function `mousePressed` then it will open a new tab every time the mouse button is pressed while the cursor is on the **canvas**:

```
void mouseClicked ()
{
    window.open("http://www.google.com");
}
```

This call could more properly be the response of a button on the splash screen. The call to `window.open` returns a handle that can be used later on. Simply declare it as:

```
var x = window.open("http://www.google.com");
```

It's now possible to test to see if the window is open, and if so close it; if not, then open it:

```
var x = window.open("http://www.google.com");
void mouseClicked ()
{
    if (x.closed)
        x = window.open("http://www.google.com");
    else
        x.close();
}
```

Sending mail from *JavaScript* directly is not possible, but one can make server requests—you can look up documentation for *Ajax* if you are interested.

Multiplayer games can be built using *JavaScript*, but again, the more complex communication protocols require some server-side intervention. Look at `node.js` and `WebSockets` for more information.

Processing can access *JavaScript* variables and functions simply by using them, because the *Processing* code becomes *JavaScript* in the end. Does it work the other way? Can *JavaScript* access *Processing* variables? Not directly. *Processing* variables are not exposed to outside access, but functions are. As a result, it is possible to code functions that modify or set *Processing* variables, and these can be called from *JavaScript*. For this to work effectively the HTML file should give an ID (name) to the canvas. So, for example:

```
<canvas id="myCanvas" data-processing-sources="code.pde">
</canvas>
```

The functions in *code.pde* can be accessed through the name `myCanvas` using a standard *JavaScript* mechanism:

```
var pcode = Processing.getInstanceById("myCanvas");
    ...
pcode.setColor();
```

The call `pcode.setColor()` calls a user written `setColor` function within the *Processing* program associated with the canvas, *code.pde*. If that function does not exist, the call will be ignored, no error message will be generated, but it probably won't behave as expected. (See: Problem 4)

Processing.js Development Tips

Developing for the Web can be quite different from developing for a PC or other computers. The Web has a number of idiosyncrasies that can cause issues when testing code. Here are some ideas that can make development easier, if not an ideal process.

- 1. Develop code using the standard Processing environment first.** Get the program working properly before submitting it to *Processing.js* though a Web server. *Processing* has better error messages, for one thing.
- 2. Save the Processing code before you execute it, every time.** Using the standard environment the newly added code is a part of the local file and executes when the run button is clicked. In *Processing.js* you have to save the file so that the most recent code is available for the Web page.
- 3. Install a local web server.** The **everything** download on <http://processingjs.org/download/> packages a Web server with it, but it needs a

language called *Python* to work. XAMPP is a basic *Apache* Web server (industry standard) which works very well on small computers and can be downloaded for free from <http://sourceforge.net/projects/xampp/>. When it has been installed according to the directions, place your development projects in the **C:\xampp\htdocs** directory. *Hockey Pong* could be copied to **C:\xampp\htdocs\proto3** for example. Then, using the browser of your choice and with XAMPP running, copy the URL **http://localhost/proto3/** into the URL box and load the page.

- 4. Develop code using the standard Processing environment before using the Web server.** The web server does not give error messages, and any problem will probably result in a blank page.
- 5. Learn to use web development tools.** For example, *Firefox* has a downloadable tool called *Firebug* that aids development, in particular by showing *JavaScript* errors and warnings.
- 6. Learn more about JavaScript.** It is important to understand your tools. It's a pretty good bet that anything that you want to do on the Web has functions available for that purpose in *JavaScript*.
- 7. Frequent the web sites and mailing lists.** Start with <http://groups.google.com/group/processingjs>.
- 8. Update and test sequentially.** If you can run with standard *Processing*, test every change there first. Save and then try with the local Web server. Finally, upload to the final Web server and try it there. Errors at later stages can be caused by missing files.
- 9. Install all major browsers.** Test every change on all browsers.
And, of course, *play a lot of Web-based games*.

Summary

The World Wide Web is a system for delivering multimedia content into people's homes, offices, and phones.

HTML is the most well-known and the most common way to create Web pages. *JavaScript* is a programming language developed to allow Web pages to be more interactive than permitted by pure HTML. HTML5 is the most recent HTML standard, and contains some new tags that specifically support multimedia display and interaction on the Web. *Processing.js*

is the version of *Processing* used to create Web-based sketches and games. It converts *Processing* into *JavaScript*, and requires a freely downloadable *JavaScript* library and a host HTML5 Web page.

Because *Processing.js* converts into *JavaScript* and *JavaScript* has no integer type, a major issue in *Processing.js* games is dealing with integer conversion. Integers in *Processing* become floating point numbers in *JavaScript* and the lack of integers and their associated properties can cause unexpected problems while converting code to run on the Web. Integer division does not occur in *JavaScript*, for example.

The other major issue is that *Java* libraries can't be used, and have to be avoided or recoded into *JavaScript*. Sound is a key element of games to which this is relevant. To play sound effects and music, use *Minim.js*, a *JavaScript* version of *Minim*.

When developing Web games, the features of the browser, and to a lesser degree of the server, are critical. Browser feature detection is important in creating games that are universally playable.

Exercises



The exercises below will apply your knowledge of the material in this chapter, and sometimes require that you do some more research before you are able to complete them. Solutions to selected exercises are available on the companion disc.

1. Write a *Processing.js* sketch that fills the canvas with a medium grey color (128,128,128) and then varies the red value up to the maximum of 255 and then down to the minimum of 0 repeatedly.
2. Write a *Processing.js* sketch that has two canvases. One is exactly the same as the one in Exercise 1, the other changes the green component of the color instead of the red component.
3. In Chapter 5, a sketch was written that drew a collection of buildings by creating textured prisms using calls to a function named `drawBuilding`. (See the sample code in Chapter 5) Convert this to work in *Processing.js*. Hint: preloading can be critical.
4. (Advanced) Create a Web page that has two canvases. The first one (called A) fills the canvas with a random grey level every second. The

other one (called *B*) checks the grey value displayed on the first and gradually changes its level to match it.

Resources

Tutorials

HTML

W3Schools Online Web Tutorials: <http://www.w3schools.com/html/>

HTMLDog HTML beginner tutorial: <http://htmldog.com/guides/html/beginner/>

Summary of HTML Tags - <http://www.december.com/html/spec/sum.html>

HTML5

W3Schools Online Web Tutorials: http://www.w3schools.com/html/html5_intro.asp

Canvas Tutorials: <http://www.html5canvastutorials.com/>

JavaScript

W3Schools Online Web Tutorials: <http://www.w3schools.com/js/>

Code Academy: <http://www.codecademy.com/tracks/javascript>

JavaScript for Beginners: <http://www.tutorialspoint.com/javascript/>

Processing.js

Processingjs.org - <http://processingjs.org/learning/>

Eight effective *Processing.js* tutorials: <http://www.designfloat.com/blog/2013/07/24/8-effective-processing-js-tutorials/>

Joy of Processing: <http://joyofprocessing.com/blog/>

Minim JavaScript Emulation

Daniel Hodgin: <https://github.com/Pomax/Pjs-2D-Game-Engine/blob/master/minim.js>

Downloads

XAMPP web server download: <http://sourceforge.net/projects/xampp/>

Node.js download: <http://nodejs.org/download/>

Quartz font for download: http://ttfonts.net/font/32075_Quartz.htm

Bibliography

Fulton, S., and J. Fulton. *HTML5 Canvas: Native Interactivity and Animation for the Web*. 2nd. Sebastopol, CA: O'Reilly Media, 2013.

Haverbeke, M. *Eloquent JavaScript - A Modern Introduction to Programming*. San Francisco, CA: No Starch Press, 2011.

Hudson, C., and T. Leadbetter. *HTML5 Developer's Cookbook*. Upper Saddle River NJ: Addison-Wesley , 2012.

Lawson, B. *Introducing HTML5*. Berkeley, CA: New Riders, 2010.

Seidelin, J. *HTML5 Games: Creating Fun Games with HTML5, CSS3, and WebGL*. Sussex, UK: John Wiley and Sons, 2012.

CHAPTER 10

ANIMATION

In This Chapter

- How to create elementary animations
- Basic animation math

Animation is a discrete art by necessity. There is no technology that permits the recording of the motion of real-world objects precisely. In between two positions of a moving object there is always another position, and recording all of them is impossible. Video recordings capture still pictures every 1/30 of a second, and when these are played back at the same speed they look good enough to seem like they are moving. It is an illusion caused by *persistence of vision*. The human eye takes some time to process an image and keeps it for a fraction of a second while processing it. Still images displayed fast enough can give the appearance of motion because we can't process the images any faster than that in real life.

Animation uses drawings, human or computer generated, to simulate a video scene. The objects in an animation don't exist except as renderings. Consecutive images in an animation, or *frames*, show motion as a change in position, size, and/or orientation of the drawn objects. With its call to draw every 1/30 second, *Processing* looks as if it were designed specifically

to display animations. Draw can simply display the next frame in sequence each time it is called, a simple program of about two dozen lines including the reading of the image files. This program is **Animation01.pde** on the accompanying disc. It reads a dozen image files showing a ball falling and then bouncing. It displays them in intervals of 1/12 of a second, and then starts over again. The essential code is:

```
void draw()
{
    image(images[frame], 0, 0);
    frame= (frame + 1)%numFrames;
}
```

In a game, animations serve many purposes, but in only one, the *cut scene*, do we display the animation as a full screen sequence of frames. In all other cases, animations form a part of the scene—perhaps a character is walking and the gait is a sequence of frames; perhaps a display on a video screen can be seen by the player; sometimes an effect, like an explosion, results from a collision. Games are a special case for an animator.

Animation forces an artist or designer to think in terms of time and motion. Game design makes a designer think in terms of story, image, and time. As mentioned in previous chapters, a game need not *be* real, but has to *look* real, so the animations that are used in a game must contribute to the look and feel of the game in that they make the game seem more real, but should not take valuable computing time away from the rendering or AI components. The motion intrinsic to the animations can make the game much more appealing and lifelike.

Very high quality games spend a huge amount of time, energy, and money on high quality animations. The characters in games like *Grand Theft Auto* are nearly perfect in their lifelike qualities at times. What will be discussed here will be just the basics, and the references are intended to lead you to more details if animation is a special interest.

Creating Elementary Animations

It is probably a good idea to do something first and then discuss the theory later. Animations consist of a sequence of drawn frames, so we'll need some drawing software. The most commonly available drawing program on a PC is *Paint*, and although it offers only elementary drawing functionality it

is perfectly useable, ubiquitous, and free. As a first project it is important to select something simple to do and yet having some complexities and growth potential. One idea is billiards.

The animation should be linear and two dimensional, and billiards fits the bill. The game has two white (cue) balls and a red one. Each of two players uses one of the cue balls, and strikes it with the cue stick hoping to hit both of the other two balls. What will be animated is one stroke. For a first draft we will need:

- a billiard table drawing, which is a pool table with no pockets
- renderings of the three balls

The *Paint* program can be used to create the images. The balls can be circles, the table is a green rectangle with markings and a wide boundary. An initial scenario needs to be set up: which ball will be struck and what direction will it move. Once that is done, the significant events in the animation need to be determined. A significant event occurs whenever something new happens: in this case when a collision occurs. All of these events are a consequence of the initial configuration, which is what makes this a *simple* animation.

Figure 10.1 shows the renderings of the table and the balls, and outlines a plan for the action in the animation. The plan is this: ball 1 is struck by the cue stick and moves along the path indicated by the line until it strikes ball 2. It will bounce off ball 2, again following the line, until it strikes ball 3. It will bounce off ball 2 into the corner and bounce out again. Ball 2 will start to move when ball 1 strikes it, moving toward the bottom cushion and bouncing off it. Ball 3 will also move when struck by ball 1, moving toward and bouncing off the left cushion.

In animation, as in life, the events occur in a particular order, and it is important to get it right. The events are collisions, and in this animation they are:

1. Ball 1 collides with ball 2. Ball 2 starts moving.
2. Ball 2 collides with the cushion, bounces.
3. Ball 1 collides with ball 3. Ball 3 starts moving.
4. Ball 3 collides with the cushion, bounces.
5. Ball 1 collides with the cushion, bounces.

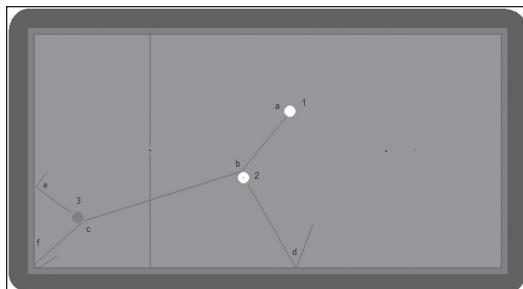


FIGURE 10.1 Initial configuration and plan for the billiards animation. The lines indicating the paths are approximate at this point.

The animation will be constructed based on these events, which are the basis for what we call *key frames*. The key frames will be drawn as the first task in the construction process. Then the animation frames that represent times in between the key frames are drawn—these are called *tweens*.

The first key frame is the initial setup and will be called key frame 0. The second, key frame 1, represents the collision of ball 1 with ball 2, and will show the contact between the two balls. From this frame can be determined the directions the balls will take for the next few frames. As shown in Figure 10.2, the bounces can be determined geometrically from the motion of ball 1 and the precise point of contact. We don't have to do any math, just make the angles look right. The rule is the struck ball will move along a line that joins the centers of the two balls. The striking ball moves away along a line that is 90° to that of the struck ball. This situation is also shown in Figure 10.2.

The second key frame will show ball 2 striking the cushion. Ball 1 will have moved toward ball 3 during this time too. Ball 2 will rebound with an outgoing angle equal to the incoming angle.

Key frame 3 will show ball 1 striking ball 3. The rules for this impact are the same as for the previous (and all) ball-ball collisions. Ball 2 will have moved farther along its track also. The remaining collisions are ball-cushion collisions, and are just like the previous one.

The next step, now that all key frames exist, is to determine the timing. First, let the overall animation take three seconds. We need to determine how much the ball slows down during each time period, and how speed is transferred during collisions. If the speed is divided equally for any ball-ball collision and none is lost on a cushion bounce, neither of which is strictly true, then the timings can be approximately determined and all key frames

can be put in their places. Time 0 is the beginning of motion for ball 0, and is the time of key frame 0.

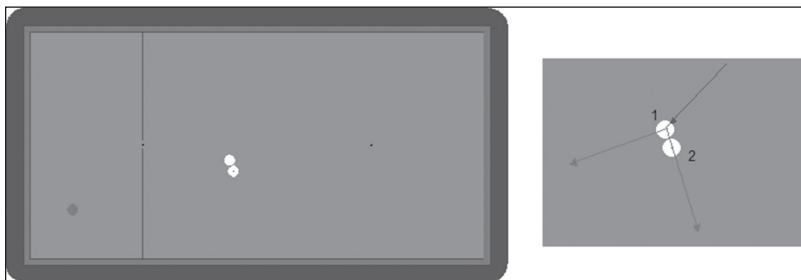


FIGURE 10.2 (Left) Key frame 1, showing the first collision. (Right) The geometry of a ball to ball collision: ball 2 moves along a line joining the ball centers and ball 1 moves away at 90 degrees to that line.

Between the points **a** and **b** ball 1 is moving at full speed (call it speed = 1). At that point ball 1 and ball 2 move at half of that speed. Ball 2 moves at that speed from then on, but ball 1 again shares its speed when it hits ball 3 (point **c**). Now balls 1 and 3 are moving at 0.25 speed and ball 2 is moving at 0.5 speed.

If we measure the distances of the paths, we'll be able to determine a time frame for the key frames. Figure 10.3(a) shows the speeds of the ball and the lengths of each path, while Figure 10.3(b) shows the time needed at the given speeds to travel the path. All distances are relative to the **a-b** distance, which will be treated as 1. It does not matter which portion we choose to be equal to 1, everything works out the same. The longest path in terms of time is **a-b-c-f** which sums to 9.4 time units. Let's make the total number of time units a nice round 10 and have all of the balls bounce a short distance off of the cushion. This means that 10 time units is 3 seconds; 3 seconds at 24 frames per second is 68 frames, or 6.8 frames per time unit. Thus the number of frames between **a** and **b** will be 6.8 (we can round to 7), between **b** and **c** will be very nearly 30, and so on. That is the last critical thing that we need to finish the animation—the number of frames between each key frame.

A table of key frame times and frames between them would look like this:

Key Frame	Time	Frame	Delta
0	0	0	0
1	1	7	7
2	3.8	26	19

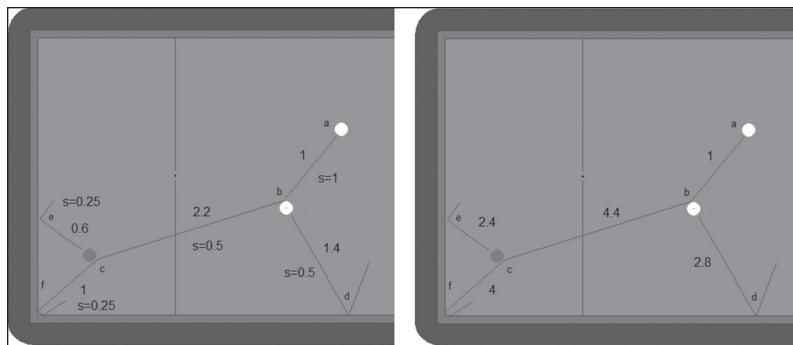


FIGURE 10.3 (a) Distances between key frame events and the speed along the path segments. (b) The relative time spent on each path segment.

Key Frame	Time	Frame	Delta
3	5.4	37	30
4	7.8	53	13
5	9.4	64	27

The column labeled *delta* tells us how many frames between key frames, and allows us to do the drawings. So between key frames 0 and 1 there are 7 frames in all.

Now we simply draw the correct number of frames showing balls on the correct background (the table) separated by the correct amount of space; that is, for key frames 0 and 1 the tweens are drawings of the cue ball moved 1/7 of the distance each time. Using *Paint*, one way to do this is to use a ruler to measure the line on the screen, divide the distance by the number of frames, and mark the positions along the paths with colored dots or lines. Move the ball to the mark, remove the rest of the marks, and save the frame; repeat this.

Numbering the marks is a good idea, because it allows recovery from program crashes, power failures, and other disasters. Figure 10.4 shows the marks used for part of the billiards animation. Note that the rule markers don't have to be precise. The tweens between any pair of key frames can be assigned to teams of artists, and because the action has been carefully scripted the result should be acceptable. A lot of famous cartoons (*Bugs Bunny*, for instance) were constructed using key frames drawn by the director, usually the best animator available, and then assigning the tweens to less senior people.

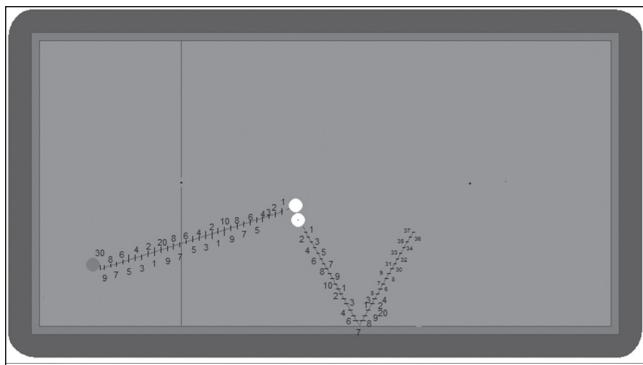


FIGURE 10.4 Marking the points on an image where the balls will be when frames are captured. Move the balls to the locations for each frame (1, 2, 3...) and then erase the markers and save the frame.

Paint has some serious limitations as an animation tool. It has no timing facilities at all, does not handle transparency, and can't even rotate objects except by 90°. *Photoshop* can deal with transparency and rotation, and has much more advanced image editing facilities—it can be expensive, though. *Flash* is designed to do animations and can do everything you might want, but is also costly and is disappearing from the Web.

The other way to conduct an animation is called the *straight ahead* method, in which the artist draws the first frame, then the next, and so on in order. Using this method is fine if all done by a single artist, and if no serious errors are made. It's quite hard to fix a single frame, and it would be likely that the animation would have to be redone from the point of the error. We'll look at this method later in the chapter.

The complete billiards animation can be found on the disc, and is available as a GIF, MP4, and AVI file. The GIF is the original, and is cleanest.

Animation Math

In order to be good at animation, an artist must have an understanding of how things move. Living things move in a different way from non-living things, and all have a *natural* motion from the perspective of a human viewer. Viewers tend to be uncritical of the math and physics and more critical of the general quality of the perceived motion, but movement that is technically incorrect is less likely to be acceptable. Let's look at the simple cases of movement of inanimate objects.

Balls and rocks and feathers move in the environment as described by Newton's Laws of Motion, of which there are three:

- 1.** Inertia. Every object that has weight will remain in its current state of motion until a force is applied to it.
- 2.** Constant acceleration. An object accelerates in the direction of the force applied to it. The greater the force, the greater the acceleration given to the object. For a given force, the greater the mass of the object the smaller will be the acceleration. The famous equation that describes this is $F = m*a$.
- 3.** *For every action there is an equal and opposite reaction.* If a force is applied to an object, the object reacts with an equal and opposite force on whatever applied the force. For instance, if you kick a ball, it pushes back on your foot.

What do all of these rules mean in the context of animation? They define what is meant by reality and reasonableness in terms of object motion. The first law has been observed by everyone mainly as *friction*. We don't expect that a box that has been pushed will move forever, because that's not what we see. We do expect it to slow and stop because of friction, and that force is present in all motion observed before the 20th century. If we'd been living in outer space our whole lives then the first law would appear to us in a more literal fashion—objects that move tend to continue to move. Friction is not a major issue in space, at least not for basic linear motion.

The second law defines how things move when they are pushed or pulled, and again our interpretation of motion that we see is defined by what we have seen before. This law is most obvious in falling objects. An object thrown into the air slows, stops, and falls back because the force of gravity acts on it. An object thrown up and horizontally moves in a parabola, the vertical motion behaving as described above and the horizontal motion behaving according to the first law. Of course, there are other equations that relate speed, position, and acceleration and those are essential to determining object positions, but are almost always related to forces applied.

The third law is more subtle, and is illustrated in day to day life as reaction to collisions. Objects that collide don't generally just stop moving, they *bounce*. An obvious example is the game of pool. The cue ball strikes another ball and imparts some of its speed to the other ball, making it move.

The other ball kicks back, and slows the cue ball and nearly always makes it change direction. Cartoon animations exaggerate this effect, and sometimes have the objects distort in shape during a collision and then return to normal form.

Motion Equations

The elementary math that describes motion is known to most people intuitively. Distance, velocity, and time are related in an obvious way. When driving a car at 60 miles per hour (MPH) for one hour, we end up driving for 60 miles. It seems simple enough, and all drivers know this. The equation is:

$$\text{distance} = \text{speed} * \text{time} \text{ or } d = v*t$$

Physicists often use the letter **v** to represent speed. It stands for *velocity*, and is technically different from speed, but for now will be treated the same. Objects in an animation must adhere to this relationship or they will look strange.

Acceleration is less intuitive. It is the change in velocity as a function of time, and we see it every day—a car starting to move when a light changes from red to green is accelerating. Elevators accelerate when the door closes and it starts to move up or down. Dropped objects accelerate downwards and then stop again when they strike the ground and stop moving. In general, an object having acceleration **a** for a time period **t** satisfies:

$$\text{speed} = \text{acceleration} * \text{time} \text{ or } v = a*t$$

If the object has a speed v_0 before it started to accelerate then that has to be considered, and the relationship becomes:

$$v = a*t + v_0$$

Finally, we can find the distance travelled by an accelerating object:

$$d = \frac{1}{2}at^2 + v_0t$$

This is very important, because it provides the way to compute the position of an accelerating object at any time. Falling and bouncing objects are accelerating and are the most common examples, so let's consider an object, a ball, which is falling; the acceleration due to gravity is $a=32 \text{ ft/sec}^2$. Assume that it is dropped, so $v_0 = 0$. At intervals of 1 second we have:

$$d = \frac{1}{2}at^2 + v_0t = 16t^2$$

time	distance (ft)
1	16
2	64
3	144
4	256
5	400
6	576

Clearly, the distance between consecutive positions of the ball is not equally spaced. An animation of the falling ball would have to be drawn with this fact in mind. Now consider the situation of a ball being thrown upwards. There is now an initial velocity, and the acceleration opposes that velocity; that is, the velocity starts as negative (meaning moving upwards), against the force of gravity. A fastball can be thrown with a speed of up to 106 MPH, but let's assume that the ball is thrown upwards at a reasonable 35 MPH, which is 103 ft/sec. This velocity opposes the acceleration given by gravity, and so will be positive while acceleration is negative. Again, at intervals of 1 second we have:

$$d = \frac{1}{2}at^2 + v_0t = 16t^2 - 103t$$

time	$\frac{1}{2}at^2$	v_0t	distance (ft)
1	-16	103	87
2	-64	206	142
3	-144	309	165
4	-256	412	156
5	-400	515	115
6	-576	618	42
7	-784	721	-63

According to this table, the ball moves upwards for a little over 3 seconds and then falls back. At time $t=7$ the ball is 63 feet below where it was originally thrown. At what time does the ball stop moving upwards? When the velocity becomes zero, and using $v = a*t + v_0$ that time is:

$$\begin{aligned} 0 &= 32*t - 103 \\ 103 &= 32*t \\ t &= 103/32 = 3.21 \text{ seconds.} \end{aligned}$$

Figure 10.5 shows this ball throwing example as points on a grid. A critical thing to notice is that when the ball is moving its fastest, the distance between consecutive drawn points is the greatest. That's because, in the fixed

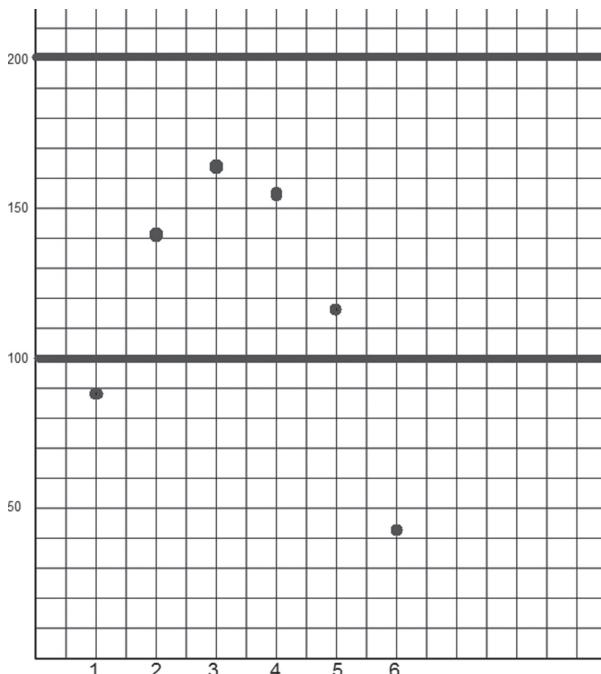


FIGURE 10.5 Ball throwing experiment. The position of the ball at selected points in time.

interval between calculation times, the ball moves farther when it moves quickly. This is quite a simple idea, but critical in an animation, where only fixed interval samples are seen.

One more bit of theory and then we can draw something else. A question of some interest is, *where is the ball in the middle of any time interval?* At time $t=1.5$ the ball will have a height of between 87 and 142 feet, but where exactly is not known from the graph. We can use the equation to figure it out, but as animators we are interested in the position relative to the other two points. Is it in the center? No.

Let's look at the dropping ball again. The equation covering this was $d = \frac{1}{2}at^2$ and we know that between the at time $t=1$ it has fallen 16 feet, and at $t=2$ it has fallen 64 feet. At $t=1.5$, half of the time between those two points, $d=8*(1.5)^2 = 36$ feet. This is 20 feet from the first point ($t=1$) and 28 feet from the second, or $20/48$ of the distance between the two points, or almost 0.4 of that distance. So, when drawing the tweens, the tween in the middle in terms of time should be drawn 40% of the distance between the two key

frames. This process can be repeated for other tweens—although it is not exact, it will be close enough. As a result, the distance between the balls in successive frames will increase, which is correct according to intuition.

In summary, basic physics can be used to calculate the positions of objects in frames. In particular, the tweens can be generated using the fundamental motion equations for objects undergoing simple motion: falling, rolling, and so on.

Reactive Animations

What will be called *reactive* animation is likely the most common sort to be found in a video game. Simply put, it is an animation that represents a reaction to an event in the game—for example, an explosion or fire after a collision, or the shattering of a brittle object that has fallen. These tend to be quite brief and not necessarily easily modeled by physics. A car crashing and exploding is an example. The animation is short, often has a random component, and can be accompanied by an external sound effect.

Many such animations are built with simple tools using a straight ahead methodology. That works pretty well because they tend to be very short, running between 1-2 seconds which is 24-60 frames. Each drawing is a variation, sometimes a slight one, of the previous frame. Using computer tools, it is possible to start drawing a particular frame using the previous one, and just moving parts of it around.

Consider an animation of a balloon popping. Start with a balloon, as shown in Figure 10.6. In each successive frame, parts of the balloon from the previous frame are erased and moved. Because the balloon is exploding, the parts should be moved away from the center. The explosion itself takes frames 15 through 23, which are the ones shown in the figure. It should be clear how successive frames have been built. The *Paint* tools *erase* and *select* are used to remove parts of the balloon and move other parts away from the center, giving an expanding volume of smaller balloon parts.

The same technique can be used to build short animations of explosions, impacts, rocket exhaust, and other event based visuals. Each frame is a random variation of the previous one, and this works well so long as the animation is short and not repeated. If played in a loop the animation loses its randomness, and the viewer can see details not intended. These are like sound effects—if they are played too often they lose their impact, so multiple short animations could be the solution.

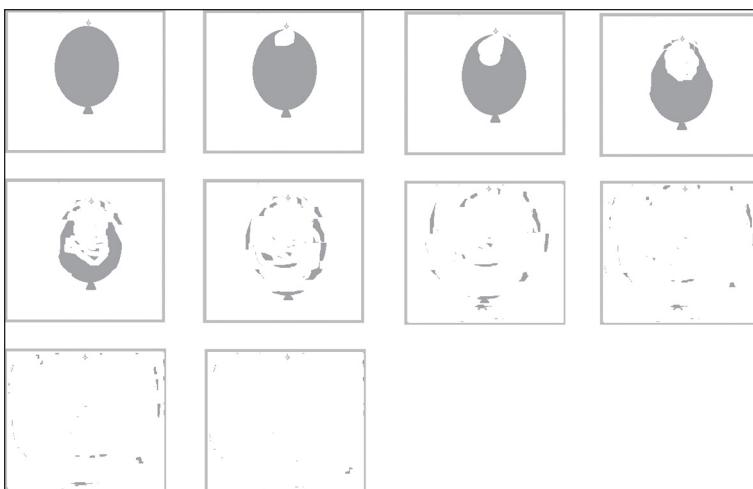


FIGURE 10.6 Drawn frames for the balloon popping sequence. Each frame is a variation of the one before, created by a random edit with the overall plan in mind.

Reactive animations will occupy a small portion of the screen for a very short time. It is important to place the frames in the correct spot in each frame. Consider an animation of a small explosion, perhaps a hand grenade. This must be seen at the location where the grenade was located just before it explodes, meaning that it could be drawn anywhere on the screen depending on the play in the game.

In terms of software, the need for animations in specific locations implies the need for software that will manage the animation, just as we had software to manage the sound clips. An animation not only consists of a set of frames, but a current frame being displayed, a location, a frame rate, and possibly an orientation, in the case of 3D games. We need a way to start and stop animations too, and some might be linked to a sound that is to be played simultaneously; sound can be a part of an animation, but for reactive animations it is never an integral part of it.

Let's design an animation manager for games, starting with what we know right now. It will possess an array of frames (`PImage` data) and a way to read them in. Included will be a way to normalize the frames; they should clearly all be the same size. We'll need a way to play and stop the display of frames, and they will be displayed at a specified point on the screen. Sound will also be managed. Here is what an outline of this, implemented as a class, would look like:

```

class animate
{
    int Nframes;           // Total number of frames
    PImage frames[];       // Actual frames
    iInt nextFrame;        // Next frame to be played;
    int xpos, ypos;        // Position for this animation
    String name;           // Root name for frame image files
    String sname;          // Name of the sound file associated
    animate ();             // Constructor
    void play();            // Play the animation
    void stop();             // Stop he animation
    void setPosition(x,y); // Set the position
    void setName(s);         // Set the image file name
    void setSName(s);        // Set audio file name
    void addFrame(p);        // Add an image to the end
}

```

To use an animation in a game, an instance of the animation class would first be declared. The frames would then be read in. It is common to have a pattern in the file names of the frames that can be recognized by a program and read in automatically. We will have a text name ending in three digits, then “.” and the suffix that defines the image file type (jpg, gif, etc). For the balloon animation the files are **balloon00.png**, **balloon01.png**, and so on in an obvious sequence. The animation class is given the name of the first file, and it reads the image files into the array `frames` until all files have been read—that is, until the next file in the sequence does not exist.

As an example of the use of this class, imagine that we have a game that uses an initial screen with a small animated feature, a jet of gas or steam. The screen will consist of a graphic, and on top of this our animation will be played. Figure 10.6 shows the screen with the animated section outlined. The frames of the animation are played sequentially, in this case as a loop, after being translated to window coordinates (28, 156), the area corresponding to the box. After initialization, each frame is displayed there in succession using the following code that uses the `animate` class:

```

animate jet;
PImage page;
void setup()                               void draw()
{                                         {
    frameRate(24);                      image (page, 0, 0);
    jet = new animate ("b", "jpg");      stroke (255, 255, 0);
    jet.setLocation (28, 156);           rect (27, 155, 50, 114);
}

```

```

page = loadImage ("002.jpg");      jet.display();
size (page.width, page.height);  }
}

```

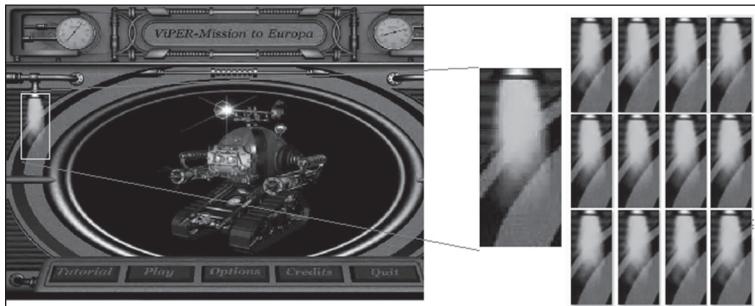


FIGURE 10.7 Animating a portion of a screen. The steam jet is translated to the correct position before display.

This animation software object seems reasonable given what we know right now. It may change a little as more requirements are seen to be needed by other types of animations. The frames of the jet animation are built starting with a small image cut from the screen. This image has the steam added to it and is saved, then the steam is varied a bit and it is saved as a successive frame, and so on until enough frames (in this case 17 in all) are created. In that way, the background remains constant and compatible with the rest of the existing background. Another way to do a similar thing is to use steam frames having a transparent background color.

Using Real Images

Until now, the animations consist of drawn images linked together in a sequence. There is no reason why a real image cannot be used at the starting point. There are two main ways to do this—to vary a real image a bit, as we've been doing with drawings, or use a short video that has been converted into still image sequence.

If a single real image is to be used as a starting point, the process is similar to the one we used before: manipulate the original image to become a second frame, then the third, and so on. It's a bit trickier to edit captured images and keep them looking real. Part of the problem is light and shading, which in real images is continuous, and part is boundaries between objects in the scene, which in captured images are not precise.

Consider the example of a candle. Some images of a burning candle could be taken using any camera. Taking enough to be used as consecu-

tive frames in an animation is possible in this specific case because candles don't change much between two images a few seconds apart. The flame may not behave as we wish in these frames, and so editing one or two into a sequence of 24 to 48 is probably a more practical idea. These images can be used as the basis for a set of animation frames: each one can be edited, shearing the flame, changing its shape and color, and so on.

Using real video data is another viable alternative. The first step is to extract the still frames from a video image, and this requires special software tools. An excellent video creation/extraction tool is *VideoMach*, which has a free downloadable version. Extracting frames is exceptionally simple: load the video and **save as** (for instance) *jpg*. The result is a collection of JPEG files in the save directory that are consecutive frames from the video, and these are named '00.jpg', '01.jpg', and so on. These files can be played as if they were an animation, and it will look just like a video so long as the frame rate is the same as in the original video, usually 30 frames per second.

VideoMach can also take a set of frames and create a video in one of a dozen formats, including *gif* and *avi*, and this is a valuable facility for previewing the animations before inserting them into the game. Simply create and play an *AVI* file, or use the built-in preview facility, so see how smooth the frame transitions are, whether the lighting is good, and if there are ar-

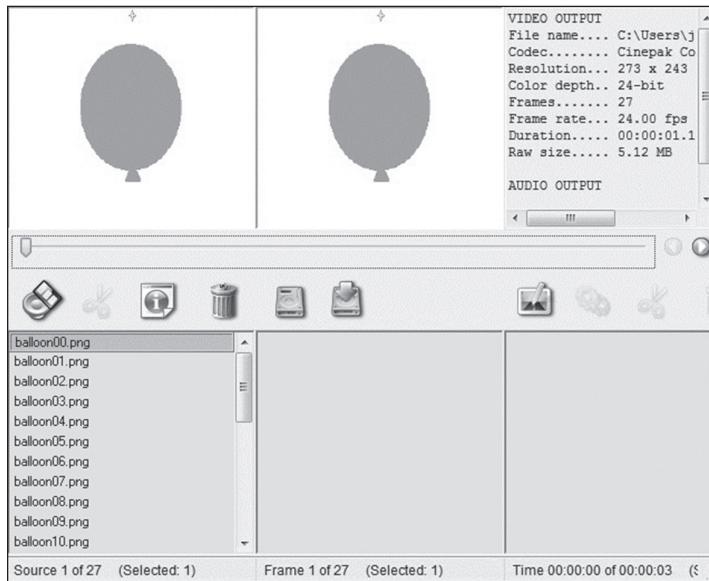


FIGURE 10.8 Using *VideoMach* to create an *AVI* video file from individual frames.

tifacts. Figure 10.7 shows *VideoMach* being used to make an *avi* version of the balloon popping animation.

Microsoft's *MovieMaker* tool can similarly create a video from still frames, but requires the user to specify the duration of the frames manually, so is somewhat less convenient, and can't save in the same variety of formats. Another tool named *UnFREEz* can make animated gifs quite easily, and is a very small and fast program.

Mac's *iMovie* can export still frames from a video, as can the free *GOM Media Player* on the PC.

Ambient Animations

Ambient animations are used to provide interesting background activity. An animation of a computer screen or data display that appears on a control panel in the background would be an example. So would smoke or sparks from damaged equipment. These very frequently have to loop, and so cannot represent an obvious pattern if that can be avoided. They also are required to be playable in many locations simultaneously. A control console can use the same animation in many positions as digital readouts so long as they are not identical simultaneously; they may need to be played out of sync. They also may need to be rescaled, skewed, or rotated too.

Making such animations is similar to making a reactive animation. It's the playback that can be distinctive. The needs of such animations will be important in continuing the design of the `animate` class. Consider, as a practical example, the creation of a control room for a spacecraft launch. In practice, there will be scores of active screens visible, but let's limit it to five. The image that will be used for the background is shown in Figure 10.8, and is in fact a still photo of the Russian control room for the International Space Station. The three large screens at the front and a couple of smaller ones will have animated displays. The screens will be the location of the animated displays, and are colored green in the original image. The green simply marks the locations, and is not needed for the display—this is not *green screen* technology.

The two small screens in the lower part of the image are essentially rectangular and can be overwritten with an animation as we've done before. Since the screens look like computer terminals we can make a set of frames that show text, images, and windows scrolling past and popping up. Making

a large set of such animations, one for each display in the image, would be time consuming and not worth the effort just for an effect. What can be done instead is to display the same sequence of frames in each location, but starting from a different point, and maybe at different speeds. This idea requires modifications to the `animate` class.

The ability to play the same animation in more than one place implies the need for an operation that makes a copy of an animation. Each copy could be placed at a distinct location in the scene. The `animate` class does this through a function called `clone` that returns a reference to a new `animate` object having the same properties as the one cloned. So, the assignment:

```
an2 = an1.clone();
```

creates a copy of `an1` that can be located elsewhere and displayed separately. The frames are the same. The code simply creates a new `animate` instance and initializes all of its local variables to those of the instance being cloned.



FIGURE 10.9 The control room background image showing green screen areas where animations will play. (Background image from NASA)

That solves part of the problem. Next, we need the ability to slow down the display of frames for an animation, and to begin playing the sequence of frames anywhere we choose. For sequencing, a new procedure named `setNext()` has been created which simply sets the value of the next frame to be played. Play normally starts at frame 0, but a call to `setNext(12)` will begin play at frame 12. Now, two adjacent screens can play the same animation and seem as if they are distinct.

The procedure `setRate()` slows down play of the animation; `setRate(1)` is the default, which plays frames at a rate of 1 per call to `display()`. A call to `setRate(2)` means that two calls to `display()` will be needed to change the frame displayed, and `setRate(4)` means that four calls to `display()` will result in a change to the frame. The larger the parameter to `setRate()`, the slower will be the rate at which frames will be displayed.

Finally, a procedure `setRandom()` allows the `animate` class to display frames in a random order. For some items such as computer screens and some sorts of motion, this gives a more realistic appearance than does a loop, where the repetition can be seen after a few cycles. People are good at recognizing patterns, and repeated cycles of fewer than about 60 frames can be identified.

The three large screens in the scene present a new problem—they are not actually rectangular. They are in real life, naturally, but they are being observed from above and to the side, so perspective has given them irregular shapes. How can we play a rectangular image frame in such a space? By using texture mapping.

We discussed texture mapping in detail in Chapters 7 and 8, where we used it to place textures on graphical shapes to make them look more real. We always mapped textures on to rectangular regions, but *Processing* does not limit texture mapping to rectangles. All that's needed is to specify a texture and four points that form a quadrilateral. The texture will be the frame that is to be displayed. The quadrilateral will be defined by the user in a call to a new procedure in the `animate` class:

```
setMasked (x1, y1, x2, y2, x3, y3, x4, y4)
```

When called, this function establishes that the animation is to be played in an irregular region defined by the four (x,y) coordinates passed as parameters. These coordinates should be passed in clockwise order starting at the upper left. Any call to `setLocation` will be ignored, as these coordinates define the location completely.

It's worth looking at how this procedure works. The frames of the animation are saved in an array named `frames`, and the next one to be played is specified by an index variable `playFrame`; specifically, the image `frames[playFrame]` will be displayed next. In the class `animate` the function `display()` is what does the heavy lifting, and it's not all that heavy. There are two ways to display a frame: the standard, as a small rectangular

image, and the masked way, using the frame as a texture. Here's the annotated code:

```
// A prior call to setMasked established the values of
// the vertex coordinates and set masked to true.
void display ()
{
    if (masked) // If the frame is non-rectangular
    {
        beginShape(); // Make a shape
        texture (frames[playFrame]); // Texture with the frame
        vertex (x0, y0, 0, 0); // Upper left
        vertex (x1, y1, 1, 0); // Upper right
        vertex (x2, y2, 1, 1); // Lower right
        vertex (x3, y3, 0, 1); // Lower left
        endShape(); // Close the polygon
    } else
    { // Display the frame as a simple rectangular image
        image (frames[playFrame], locX, locY);
    }

    count = (count + 1)%playRate; // Next frame
    if (count == 0) // Increment or not? This code implements
    { // the frame rate control.
        playFrame++;
        if (playFrame >= nextFrame) playFrame = 0;
    }
}
```

The `draw()` function that displays five smaller animations within the scene of Figure 10.8 looks like this:

```
void draw()
{
    image (page, 0, 0); // Display the background image
    main.display(); // Center main image, mapped
    main1.display(); // Left main, mapped
    main2.display(); // Right main, mapped
    ll.display(); // small lower left, rectangular
    lr.display(); // small lower right, rectangular
}
```

This program works on most computers, but it might be necessary to allocate more memory to *Processing*. The number and size of the frames dictates whether this will be required. To increase the memory allocation

click on the **File** menu on the *Processing* sketch window and select **Pref-
erences**. The window that pops up has a way to select more memory (“In-
crease maximum available memory to ___ MB”). If more than 512 MB are
needed, it may be a good idea to find ways to decrease the sizes of the im-
ages. Rescaling and new formats are the first things to try. The program that
displays the control room animations can be found on the companion disc.



Character Animation

Creating animations of living creatures is among the hardest tasks in video games, and animating humans is the most difficult. It’s because living things move in very complex ways, and viewers are very familiar with how that motion should look, and are therefore quite critical of flaws. Character animation is a specialized subject, and we will look at the most commonly needed type here: gait.

If your game has a human character, it will need at the very least to walk around the game space. The legs and arms must behave as we human observers expect them to. We will need to build a short animation of the character taking a single complete step, and then this can be played whenever the character moves. Of course, the avatar may need to jump, shoot, reach, crouch, or do many other tasks, but character animation is a complex skill that depends to a great extent on the artist’s abilities.

Animating gait is a matter of creating a sequence of drawings that shows the arms and legs of the character walking in a normal way. From the illustration in Figure 10.10 a set of workable steps is shown. First (top) is a set of key frames, in this case showing the right arm and leg of a stick figure during a single full step. Next the left arm and leg are added and tweens are created (middle). Finally, the character can be fleshed out over the skeleton and colors or textures added. The final sequence here has just enough detail to be useful both as an example and as an avatar. This avatar is carrying a pack of some kind on his back.

When the avatar is moved—for example by pressing the right arrow or the D key—the frames are played in sequence and the position of the avatar is moved to the left, giving the desired effect. The frames can be redrawn or simply flipped horizontally to allow the avatar to move to the left.

This discussion of character animation is intentionally trivial. Although the actual display of frames is simple and uses methods that have been discussed in detail, the creation of the frames can be profoundly complex.

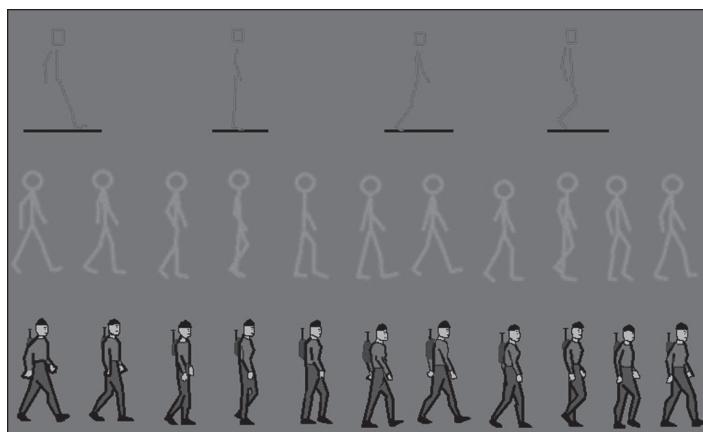


FIGURE 10.10 (top) Key frames for a gait, right side only. (Center) Tweens for the gait. (Bottom) A set of frames for a single stride.

If you have the ability to create realistic motions in a frame sequence then your games will profit from this skill. If not, it can be learned or (perhaps better) people with that skill can be engaged to work for you. The cost is small and the improvement in the quality of the result can be priceless.

Cut Scenes

A *cut scene* or *full motion video* (FMV) is an animated narrative that explains a part of the game's story or background. It is played on the full screen like a movie. There is often one at the beginning of the game, and often one that serves as a transition between levels. They are made obvious by a change in aspect ratio to a more cinematic 1.85:1 from the standard computer screen value of 1.6:1. No interaction with the game is possible during a cut scene.

These can be made in any way one chooses, even with live action video. The issue again is how to play it. Cut scenes can have a much higher image quality than the game proper, and to read and display single frames would take far too much time and memory to be practical. What you must do is to create an animation, save it as a video file, and play that file at the proper place in the game. This section of the book could be called, *How to use Java based video classes and functions*, not unlike the way we played sound files using the *Minim* class. To play videos we use the *Movie* class.

The use of this class is simple enough, although a detailed understanding of it is harder to come by. First we need to import the class. Then

declare a variable of type *Movie*; a video is assigned to it through the constructor and passing the name of the movie file. The video is started by calling either the `play()` or the `loop()` procedure of the *Movie* object and is rendered on the screen just as an image would be, by using a call to `image()`, because a variable of type *Movie* can be used as an image. We also need to code a procedure named `movieEvent()` that will handle, well, movie events. The only such event of importance is the one that says a new frame image is ready, decoded from the movie file, in which case we call the `read()` procedure for that *Movie*.

What appears to be happening is that the video file is opened by the *Movie* constructor and input is started. A video file is not just a sequence of frames, but is compressed both spatially and temporally, and the *Movie* class seems to collect frames until an internal buffer is filled. When a frame is available the user provided procedure `movieEvent` is called by the *Movie* class asynchronously, and this allows the user to call the `read` procedure to make that frame the current one. The *Movie* variable is useable as an image for the display function `image()`, so this function can be used to put the next frame on the screen.

For the *Processing* video library to work, you need to have *QuickTime* available on your computer. It can be downloaded for free. *QT Lite* will not work, and at this time *QuickTime* is not available on *Linux* machines. There is an alternative video library named *GSvideo* which might help in these cases.

An example of a program that can display the video file of the steam jet (**steam.avi**) is:

```
import processing.video.*;      // Import the class
Movie clip;                    // Declare a variable
void setup()
{
    size(149, 212);
    clip = new Movie(this, "steam.avi"); // Read the video file
    clip.loop();                      // Start playing
}
void draw()
{
    background(0);
    image(clip, 50, 50);      // Each iteration display a frame
}
void movieEvent(Movie m) // When a new frame is available
```

```
{
    m.read(); // Read a new frame
}
```

So, to play cut scenes in a game, one needs to add special states to the game play FSA to enable the `Movie` files to be connected to the proper variables, and to display them at the proper moment.

Summary

In *key frame* animation, the frames that define the motion are created first, and then the frame in between those (*tweens*) are drawn. This lends itself to a production line system that uses multiple animators. In addition we can more easily synchronize key moments in the action with specific frames. However, the use of many artists can lead to inconsistent drawings. In *straight ahead* animation we create the first frame, then the second, then the third. This leads to artistically exciting images, and is good when there is a lot of action. On the other hand, there is a lot of pressure on the animator as this method requires a great deal of concentration. It's also hard to correct, and some kinds of specific timing is hard to do (e.g. lip synch).

Basic physics can be used to calculate the positions of objects in frames. In particular, the tweens can be generated using the fundamental motion equations for objects undergoing simple motion: falling, rolling, and so on.

A *reactive* animation is likely the most common sort to be found in a video game, and represents a reaction to an event in the game—an explosion after a collision, for example. *Ambient* animations are used to provide interesting background activity. An animation of a computer screen or data display that appears on a control panel in the background would be an example. A *cut scene* is an animated narrative that explains a part of the game's story or background. It is played on the full screen like a movie.

Creating animations of living creatures is among the hardest tasks in video games, and should be avoided if possible or hired out to a professional if not. Avatar motions depend on character animation and need to be good.

Exercises

The exercises below will apply your knowledge of the material in this chapter, and sometimes require that you do some more research before you

are able to complete them. Solutions to selected exercises are available on the companion disc.



- 1.** Make a short (<10 second) animation of a ball bouncing. A basic solution could be a loop, but really it should bounce to a lower height each time, and finally end up resting on the ground.
- 2.** Make a short animation of a ball bouncing down some stairs. You may use the animation from Exercise 1 as a start.
- 3.** Create a video file from one of the animations in Exercise 1 or 2. Use any tool you like, but document the process you used in a brief (< 5 page) document.
- 4.** Take some images of water flowing from a faucet. Create a short animation of this flow and display the frames as an animation. Using any tool you like create a video file of this animation.
- 5.** Make a video of water flowing from a faucet and extract the frames. Select 24 or so and play them in random order in a small window as an animation. Describe how it looks different from a video of the water flowing.
- 6.** Record a video of you (or a friend) banging your fists on what could be an invisible glass box enclosing you. Select a second or so of this video and extract the frames. Texture map them onto a rotating cube so that it appears as if you are trying to escape from it.
- 7.** Observe the gait of any animal. Try to create an animated gait for this creature. Play it back and try to find flaws—keep a written description of the process and what you find.
- 8.** An entertaining way to make an animation is the *stop motion* or *stop frame* technique, in which actual objects in an actual scene are moved, photographed, and moved again. Miniatures are often used. Make a stop frame animation using any material at your disposal and turn it into a video file. Plan the animation using key frames.
- 9.** A common tool used in the creation of animations and films is the *storyboard*. This is a sequence of drawings, usually including sketches of the key frames, that tells the story of the animation. It helps with the design, and is used to present the idea to the production group. Make a storyboard for any 30 seconds of animation you choose—a cartoon, perhaps.

Part of the value of a storyboard is in its presentation so, if you can, present your storyboard to a small group.

Resources

Wideo: An online animation creation tool: <http://wideo.co/>

Dreamstime, stock photos for use as textures: <http://www.dreamstime.com/>

VideoMach video creation tool: <http://gromada.com/videomach/>

UnFREEz tool for making animated GIFs: <http://www.whitsoftdev.com/unfreez/>

Macintosh *iMovie*: <http://www.macworld.com/product/412943/imovie-09.html>

Movie ToImage, for getting stills from an iPhone: <http://www.macworld.com/product/599549/movietoimage.html>

GOM Media Player can save stills from a video: <http://player.gomlab.com/eng/download/>

Apple's *QuickTime*: <http://support.apple.com/kb/dl837>

GSvideo: <http://gsvideo.sourceforge.net/>

Storyboard That, an online storyboard creation tool: <http://www.storyboardthat.com/>

Bibliography

Faigin, Garry. *The Artist's Complete Guide to Facial Expression*. Toronto, ON: Watson-Guptil Publications, 1990.

Furniss, Maureen. *The Animation Bible*. New York, NY: Abrams, 2008.

Lemay, Brian. *Layout and Design Made Amazingly Simple*. Oakville, ON: Animated Cartoon Factory, 2006.

Pocock, Lynn, and Judson Rosebush. *The Computer Animator's Technical Handbook*. San Francisco, CA: Morgan Kaufmann, 2002.

Webster, Chris. *Animation: The Mechanics of Motion*. Burlington, MA: Focal Press, 2005.

White, Tony. *Animation from Pencils to Pixels*. Burlington, MA: Focal Press, 2006.

—. *The Animator's Workbook*. New York, NY: Watson-Guptill Publications, 1986.

Williams, Richard. *The Animator's Survival Kit: A Manual of Methods, Principles, and Formulas*. London, UK: Faber and Faber, 2001.

CHAPTER 11

ANDROID HANDHELD DEVICES

In This Chapter

- How cell phones work
- Targeting the Android operating system from Processing
- Troubleshooting installation

The best estimate is that there are 6.8 billion cellular phones in service as of 2013. In the second quarter of 2013, the *Android* operating system possessed a nearly 80% share of the smartphone market. This figure varies with the source of the number, but *Android* phones are clearly the market leader. Making games and other applications for *Android* could be a profitable enterprise, and will reach a lot of people. Most of us have a cellular phone, but what are they really? What is *Android*, and when you *run an app*, what actually happens?

And, of course, how can we create a game (*app*) for an *Android* phone?

There are many tools available for developing apps, and many of those can be downloaded for free from the Internet. A great thing about *Processing* is that it can make *Android* apps from sketches with relatively little effort. Of course, it is not easy to access the full *Android* system, and so there will be some things that your app will have trouble with, but in principle, getting started is simple.

A warning: getting an *Android* app to work properly using *Processing* may require some patience. The software is complex and installation is a bit touchy. There are many versions of *Android* in common use and they all have slightly different characteristics. The development system connects *Processing* to many *Android* libraries, a downloader, and an emulator; this arrangement is sensitive to the executing environment on the PC—and if you think about it, no two PCs on the planet are really identical after they've been used for a short while. There are many ways for the system to fail. This chapter represents an attempt to clarify the most common issues so that *Android* programs can be developed successfully. The situation will change with time, for the better one hopes, but knowing these pitfalls will have to be useful, and if the system works perfectly for you the first time then so much the better.

How Cell Phones Work

Some people think of cell phones as being like the old *walkie-talkies*. They are, after all, radio transmitters and receivers, and have a small antenna and therefore a limited range. However, a big difference is that they do not receive transmissions directly from another phone. All signals from cellphones are intended to be received by a large fixed antenna, which is linked to a sensitive receiver. This connects your call to the telephone system, sometimes by wire, or optical fiber, or microwave, and hence to a computer switching system and then to its destination. If that destination is another cell phone, the antenna nearest the destination is selected, the call is sent to it, and it is transmitted to that cellphone with an indication of who it is for. All of this happens even if the caller is standing right next to the person being called.

That last part is essential, the part about calls being for a specific phone, and knowing about it opens up a whole realm of technical complexities. There is a way that your phone knows what calls are for it and what calls are not, and your phone ignores the calls that are for other people. The implication is that your phone could actually listen to any call, calls that were not destined for you, which is the problem with radio—anyone can listen. And indeed, we don't even need a phone to listen to cellphone conversations. All we need is a radio receiver tuned to the right frequencies.

When a cellphone is turned on, it listens for (tries to receive) an SID code (System ID) from the company providing phone service. If one can

be found a connection is made, and the company knows that your phone is ready to get a call. You will communicate with the nearest cellphone tower (antenna), and these are placed in a hexagonal grid across wide areas of the world so that you will always be near to one. When you make a call your phone places a request with the service provider, who assigns two frequencies to you: one to receive information and one to send. This means that you can talk and listen at the same time.

Most systems send information in digital form. That means that it may not be possible to simply listen to your call using a radio receiver, because the conversation will be sent in digital form, such as an *mp3* file. However, the conventions for sending these files are known, and it is a simple matter to connect a computer to a radio receiver and download and decode the sound files. In other words, cellphones are not a secure form of communication. However, since the phone is sending digital information anyhow, it can also download computer programs to run on its internal processor—these are *apps*. The computer on a cellphone is quite complex by historical standards and has a lot of memory, and apps can be very involved programs—for example, a GPS navigation system is quite a sophisticated piece of software. A typical cellphone contains a general-purpose computer that can run downloaded programs, provide a user interface, and, of course, handle telephone calls.

A computer typically runs an *operating system* that allocates resources and deals with the interface and user requests. *Android* is one of a small number of cellphone operating systems, other main ones are *iOS* (Apple), *Symbian* (Nokia), *Windows Phone* (Microsoft), and *Blackberry OS*. A cellular telephone is what used to be called an *embedded system*, a computer that was lodged inside of another device. There are a lot of those now: cars, refrigerators, DVD players. But a cellular phone is a portable device, one that people carry with them all day and has positional and orientation issues. This makes it different from other computers, and the operating systems are also different from those that run on laptops and PCs.

Android

Android is an operating system based on the *Linux* system that has been available on PCs and other computers since about 1991. The internal operation of *Android* is of relatively little interest to a game developer, but its limitations are important and those will be discussed as they become an issue for a game programming. It is an open-source system, meaning that

anyone can examine the source code and see how it works. It is sufficiently complex that it might not help to do so. It is the most used cell phone operating system in use today.

Android (2013) uses a standard ARM, MIPS, or x86 32 bit processor, and needs 512 Megabytes of memory. It implements the graphics standard OpenGL ES. It uses a so-called *touch screen* interface, where the user's finger touches and swipes the screen in place of the mouse found on standard PCs. There are many optional devices that are supported, such as GPS, accelerometers, and other position and touch related items. *Processing* does not make all of these easily available yet.

User applications (apps) are also known as an *Android* application packages, and are simply programs compiled in a certain way, signed, and archived. The file suffix **.apk** is indicative of an app. Most apps are coded in *Java*, so *Processing* is a natural option.

Targeting Android from Processing

In spite of the fact that *Android* apps are usually built using *Java*, and that *Processing* is really an extension of *Java*, the systems are both complex and they were not designed to work together, so problems can appear.

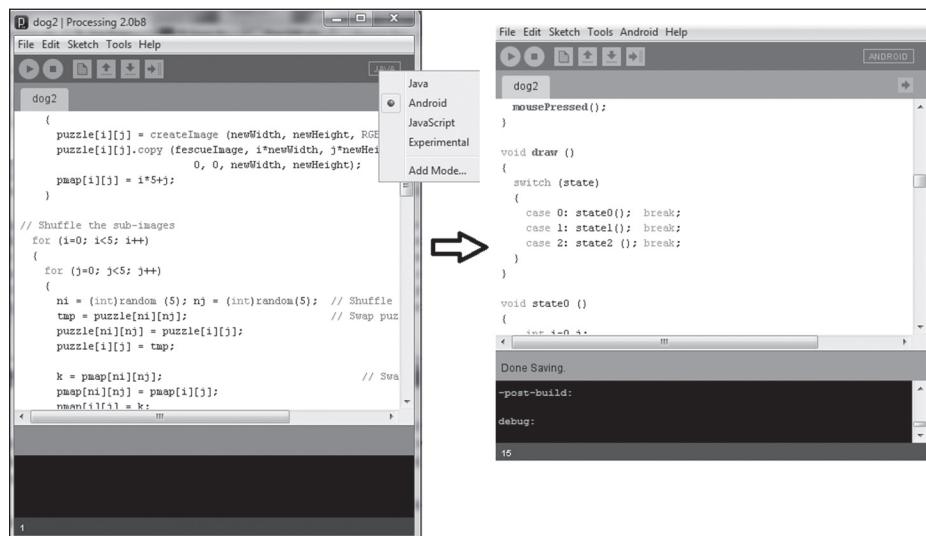


FIGURE 11.1 A standard *Processing* sketch window (left) that is Java/PC based; the *Android* window (right) targeting the portable system.

Recent versions of *Processing* all possess an *Android* option as indicated on the sketch window. Figure 11.1 shows a typical sketch window and what happens when the Java box is clicked on with the mouse. A menu pops up allowing the selection of *Android* (or *JavaScript* or other things). The window changes color when *Android* is selected, and the word *Android* appears in the selection box instead of *Java*. This step is only the first is a sequence of events that allows the development of *Java* code targeted for the *Android* system. Next we have to set up the *Android* development system and emulator on our computer.

Getting Set Up

The *Android* SDK (Software Development Kit) needs to be installed on your computer before *Processing* can create *Android* apps. It contains the libraries needed to interface to the system and developer tools that make development a more straightforward process. Download it from the Website <http://developer.android.com/sdk/index.html>. Be certain to click on the text box **Use an Existing IDE**. The other box, labeled **Download the SDK** will give you the Eclipse version, and that's not what we want. The file downloaded should be an executable named **installer_r22.3-windows.exe** or something similar—double click on it or to run it.

Remember where it installs (the directory is listed at the top of the installation screen). An image of the SDK manager window that will be on the screen is shown in Figure 11.2. There are check boxes that you can select to download more tools. At the least you will need to check the following:

Under the **Tools** item check the box for **Android SDK Platform-tools**.

Under the **Android 2.3.3 (API 10)** check the box for **SDK Platform**.

Under the **Extras** item check the box for **Google USB Driver** (Windows only).

The **Install n Packages** box will be colored now, and will have a small number where *n* appears. Click on this box and install these new items. It will take a few minutes. Sometimes a few of the items don't install the first time, in which case the window will indicate which ones have been downloaded using a check box, and the **Install n Packages** will display a smaller number of *n* than it did previously. Keep installing until all have been successful. Figure 11.2 shows an example of the download manager for the SDK.

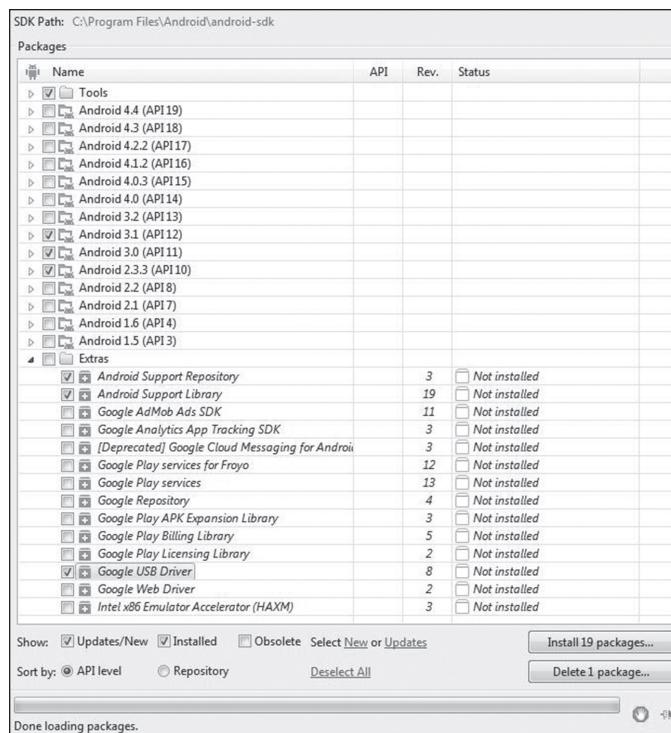


FIGURE 11.2 The Android SDK Manager window showing the location where the SDK is installed and listing the packages that can be downloaded.

The *Processing Android* wiki has more detailed instructions for all of this, but may not always be available. It can be found, as of March 2015, at <http://wiki.processing.org/w/Android>. There are many packages that can be downloaded. Some of these are for specific *Android* releases, some are specific tools that might be of interest to certain developers.

When you have installed all of the *Android* SDK successfully, reboot your computer. Then you can try to compile and execute your first program. Here's a simple program to try:

```
int x=30, y=30;
int dx=10, dy=10;

void setup ()
{
}

void draw ()
```

```
{
    background(255, 255, 255);
    fill(255,255,0);
    ellipse (x, y, 40, 40);
    x = x + dx;
    y = y + dy;
    if ( (x<0) || (x>width) )
        dx = -dx;
    if ((y<0) || (y>height))
        dy = -dy;
}
```

This just moves a yellow circle about the display area, bouncing off of the boundaries. Enter this code, select *Android* as the target, but do not click on the start icon, the triangle in the upper left of the window. Instead, click on the menu item named **Sketch**. A menu will drop down as shown in Figure 11.3. Select the **Run in Emulator** option and stand back.

What is supposed to happen is that the *Android* emulator will begin to run, a program that behaves like the cellular phone running *Android* that is the target. The sketch will be sent to it, and will be executed there. This can take up to a minute, and windows will appear and change. Ultimately, the program will be displayed in a graphic simulation of a cell phone drawn on the screen.

It won't work, very probably, but don't give up too easily. There are a few things that it will help to know.

Getting Through the Installation Issues

Common Problem #1

One thing that it is very important to realize is that errors in the compilation of your program will appear as usual in the black region at the bottom of the sketch window, but they may be hidden by messages from *Processing* and *Android*. When you select **Run in Emulator**, your program is first compiled and then sent to the emulator. Compilation errors will be at the top of the message area, and you'll have to scroll up to see them. Errors from *Android* can be caused by a failed compilation.

One good idea is to always compile the code in *Java* mode first, and only try to load to *Android* when it works the way you want.

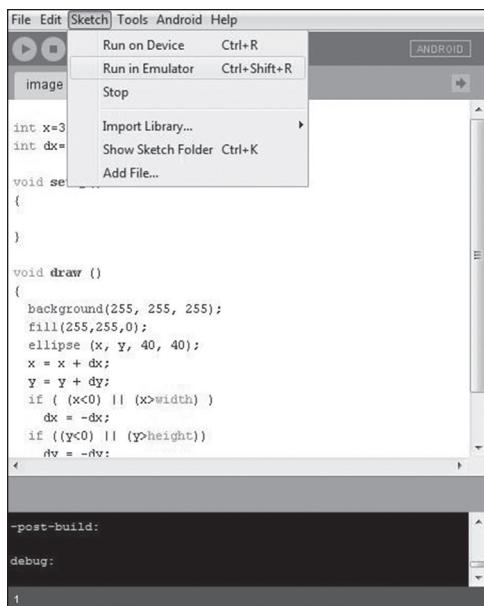


FIGURE 11.3 Running a *Processing* program on the *Android* emulator.

Common Problem #2

Sometimes the emulator does not execute automatically. If so, you will get a message that says “Waiting for device to become available” for a long time. Finally, it changes to “Lost connection with device while launching. Try again.” Did you click on the start triangle icon by mistake? If so, the system is attempting to find a real *Android* phone connected to your computer and is failing.

If you did try to start the sketch using the emulator and got this message, perhaps the emulator is not being started automatically. Click **Android** on the sketch menu, and a new menu will drop down. One of the items there should be **Android AVD Manager**. Select it and a new window, the **Android Virtual Device Manager** window, will open. Figure 11.4 shows this. Note that in the Figure there is one item listed: something called *Processing-Android-7* appears with a check mark next to it and some

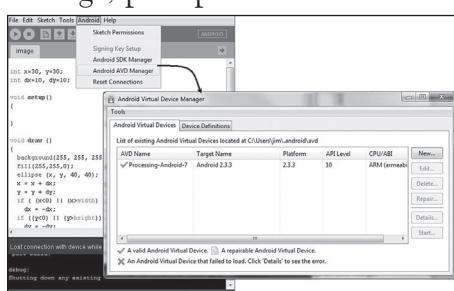


FIGURE 11.4 Starting the *Android* Device Manager where the emulator is.

other information. Click on this. The **Start** box on that window should be activated, and if you click on that the emulator will start. You will have to do this each time you start *Processing*.

The most complex situation occurs when there is no *Processing-Android-7* emulator in the list. In that case you need to create one. This situation is illustrated in Figure 11.5. The **AVD Manager** window shows no emulator, so click on the **New** box. Fill in the options as suggested in the figure and click on the **OK** button. Now the AVD Manager window should show the device you created. Select it and click on **Start**.

When the emulator begins, it will open a window on the screen. This window will start out as a simple graphic and change gradually into the appearance of an *Android* cellphone screen. Now try to run your program again.

If that fails try yet again. Failing that, stop the emulator and restart it. Be patient.

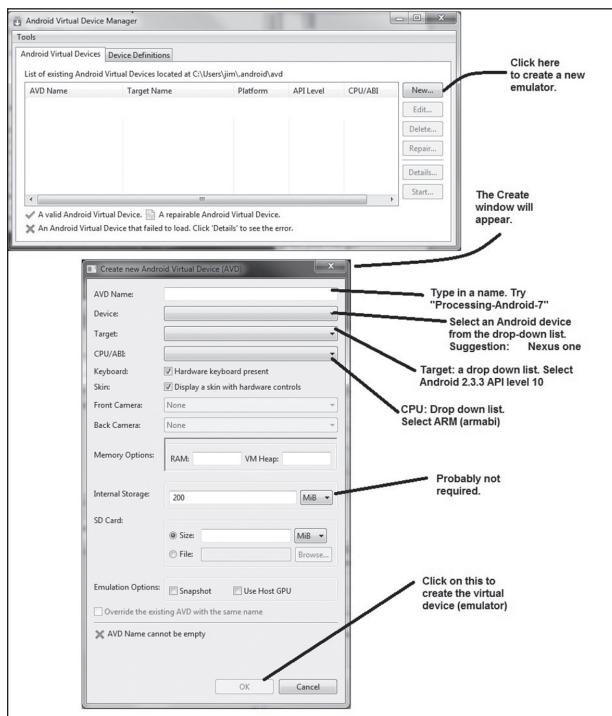


FIGURE 11.5 Creating your own version of the emulator if one does not exist.

Common Problem #3

Sometimes a version of the SDK is released that presents problems for some of the users. A symptom of this is that even very simple programs fail to execute properly, giving errors such as the following:

```
UNEXPECTED TOP-LEVEL EXCEPTION:
[dx]  java.nio.BufferOverflowException
[dx]  at java.nio.Buffer.nextPutIndex(Buffer.java:499)
...
[dx]  at com.android.dx.command.Main.main(Main.java:103)
BUILD FAILED
C:\Users\Jaap\android-sdks\tools\ant\build.xml:892: The following
error occurred while executing this line:
C:\Users\Jaap\android-sdks\tools\ant\build.xml:894: The following
error occurred while executing this line:
C:\Users\Jaap\android-sdks\tools\ant\build.xml:906: The following
error occurred while executing this line:
C:\Users\Jaap\android-sdks\tools\ant\build.xml:284: null returned: 2
```

In this specific case, start the *Android* SDK Manager (from the **Android** menu on the sketch). Expand the **Tools** item and see what has been installed. If revision 19 of the **Android SDK Build-tools** and **Android SDK Platform-tools** have been installed, then select them with the mouse and click the **Delete Packages** button. This may fail; if after a few tries you cannot delete these, then remove the entire SDK and re-install it without these specific packages. Revision 18 is known to work.

As time passes these instructions may become less relevant, but something similar is bound to be needed. It's a good idea to explore the SDK and become familiar with the online user groups who have experience with the system.

Common Problem #4

Your sketch can't display any images. You get null pointers from `loadImage()`.

Images must be *copied* into the sketch. We've not had to do this before, but we *must* do it for *Android* code that uses images. Doing this is easy—select the **Sketch** menu item, and then select the **Add File** item at the bottom of that menu. A file selection window will open, and you need to navigate to the location of your image file and select it by double clicking on it. Now your image will be known to *Android*.

These are the common serious problems encountered when trying to get a first *Android* program to compile and run. There may be others. The ability to build *Android* programs that are essentially PC applications too, and that can run on a Web page, is remarkable, and one should not expect it to be without problems especially in the early stages.

Your First *Android* Program

When any installation problems have been solved, you will be able to execute your code on the emulator. Let's try with the program written above, the one with the bouncing circle. Run *Processing* and type in the program or copy from the enclosed disk. Now start the emulator. The emulator runs in its own window, and there are four stages in its initialization, each indicated by a different graphical appearance. These are illustrated in Figure 11.6. If the initialization gets stuck you should notice, and it can be restarted. State 3 requires some intervention—this represents a locked phone, and to run a sketch you must click the mouse on the lock icon and drag right—on a real phone we would touch the icon and drag right. When successful, the final screen will become visible, and the emulator is ready to run your code.

Now select **Sketch** from the *Processing* window menus and within that select *Run in Emulator*. There are normally four stages in the progression of a compile/load sequence: the build, waiting for the device (the emulator or phone) to respond, installing the sketch, and launching the sketch. *Processing* gives a message in the sketch window announcing each of these, as shown in Figure 11.7, which gives the usual progression of these steps. If it fails, it's possible to try again, hoping that the emulator simply missed a message. Persistent failures mean that the emulator or perhaps the system has to be restarted.

When the emulator runs the program successfully it will be obvious: the yellow ball will bounce across the surface of the emulator screen.

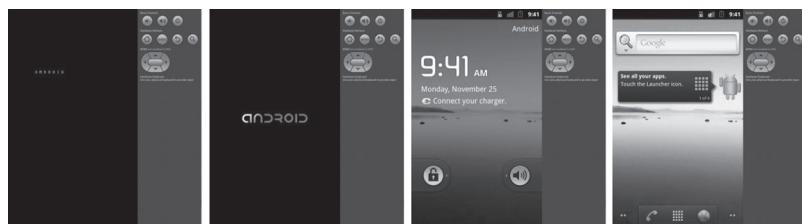


FIGURE 11.6 The start-up screen sequence for the *Android* emulator.

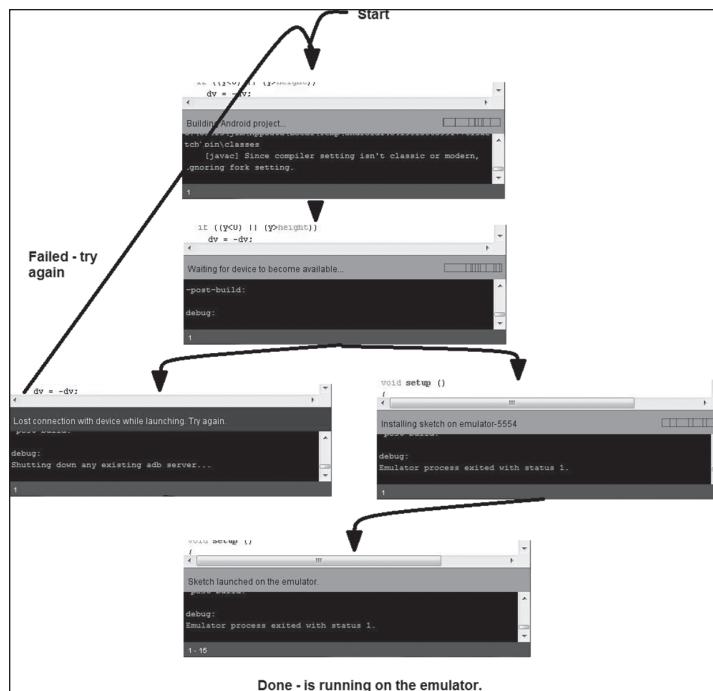


FIGURE 11.7 Build sequence as illustrated by *Processing* progress messages. You can try again after a “Lost Connection” message.

An Android Puzzle: Touch Events

Let’s now create a simple game. It will be a puzzle in which an image, in this case of a dog, is cut into 25 square pieces and then shuffled. The pieces are all right side up but scrambled. Touching one of these squares on the screen makes it change color to indicate that it has been selected; touching another exchanges the two squares. The goal is to put the image back into its original form by interchanging pairs of sub-images. This is a pretty easy game to build, but the goal is to get something non-trivial executing on an *Android* device, not to change the world of game design.

The game will do the following things: it will read in the image and break it into 25 (5 by 5) equal sized portions. The image will be 400×400 pixels in size, so the portions or sub-images will be 80×80 pixels. Now these portions will be placed in random order into a 5×5 array of `PImage`s—this will represent the playing surface. At the same time, the number of each sub-image will be stored in a 5×5 integer array, where upper left is num-

bered 0, the one in the next column is 1, and so on. This array is shuffled along with the images, and will be exchanged with them too, so that when it is in order again the puzzle is solved.

The display will show the shuffled portions arranged as a 5×5 grid so that it is the same size as the original image, and will be displayed in the top half of the screen. The original image will be displayed in the bottom half of the screen so that the player can see the target at all times. When the player touches the screen, the sub-image touched (if any) will be outlined in light blue and will have a partially transparent square drawn over it so that the player can see the selection. This means that the touch has to be detected by the program and the location determined so the square can be drawn.

A second touch over another sub-image will result in an exchange of images, which is really just a matter of copying one element of the array into another and vice-versa. The outline and transparent square will go away. If the same cell is touched twice in a row then no change occurs. To be nice to the player, when a sub-image is in the wrong place a white spot (circle) is drawn in the center, which goes away when it is correctly positioned.

This is a straightforward program, and it should be coded using the standard *Processing* environment and made to work there before moving to *Android*. Here are the key things that change when *Android* is targeted:

1. The dog image, named **dog400.jpg**, must be copied to the sketch. Select the *Sketch* menu item, and then select the *Add File* item at the bottom of that menu. A file selection window will open, and you need to navigate to the location of your image file and select it by double clicking it. Now your image will be known to your *Android* sketch.
2. Screen touches are mouse events. You must use the procedure `mousePressed()` to determine when and where the screen has been touched. The function `mouseClicked()` does not work, probably because *Android* has no concept of a mouse button.
3. The procedure `size()` is not required, because each *Android* device has a fixed and known size. If you specify a size then the behavior of the screen is unpleasant—it appears to center smaller screen sizes in the available space, and large screens are cropped.

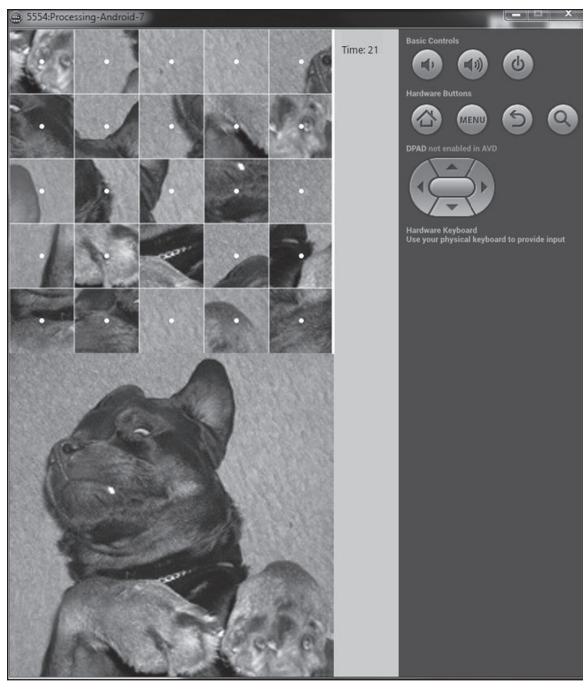


FIGURE 11.8 The emulator window showing the dog puzzle game executing.

A screenshot of this game being played in the emulator appears in Figure 11.8. A timer has been placed on the screen, and this can be used by the player as a kind-of score. At the end of the game, the closing screen displays the final time needed and the number of moves taken.

Playing a Game on a Real Device

So far, the development that has taken place has all ultimately been running on a PC. Yes, the *Android* system is the target, but the code is really executing on an emulator which in turn runs on a PC. Can these games we're planning to develop using *Processing* run on a real *Android* device? Yes, of course. To do so one needs an *Android* phone or tablet and, initially, a USB cable to connect the phone to the PC.

Turn on the target device (phone or tablet) and go to the **Settings** application. Navigate to **Settings/applications/development** and look for the item **USB Debugging**—make certain that the box for this is checked (active). Now return to the main screen and plug your phone into a USB

port on your computer. In the *Processing* sketch window, click on the usual start button (the triangle icon), type control-r, or select **Run on Device** from the **Sketch** menu. The code will be rebuilt and should load on the device. It is supposed to be that simple. However, sometimes you will get the following message:

Failure [INSTALL_FAILED_OLEADER_SDK]

In this case your phone may have an old version of *Android*. You could update and try again, or you could edit the file **AndroidManifest.xml** in the directory having the source code. Look for the line that is something like:

```
<uses-sdk android:minSdkVersion="15" />
```

and change the 15 to something smaller, say 10. This may fix the problem, but you could go to lower versions if necessary—at some point it's likely to fail due to a version that's too low.

Another problem is that the emulator is not guaranteed to work properly with all phones—it is set up to be a particular one. We set up the emulator to be a *Nexus One*, and this has a specific screen size. It is often disturbing to see your game displayed on a screen of a different size and/or resolution. When this is run on a *Galaxy-S*, for example, the result is less than ideal. It should be set up to deal with various screen sizes.

Fixing the dog puzzle was easy, and while it could have been presented in its final form in the first place, it is educational to see how things are learned as development progresses. The screen size of the display device is given by *Processing* in the variables `displayWidth` and `displayHeight`, so rather than relying on the screen being that of a *Galaxy-S* it is possible to make the program more general. Read the image, get the display size, and do this:

```
if (displayWidth < displayHeight/2) i = displayWidth;
else i = displayHeight/2;
dogImage.resize (i, i);
newWidth = newHeight = i/5;
```

This determines the proper square size of the image so as to fit the screen, resizes the image appropriately, and tells the program what the size of each sub-image is. The original code inappropriately assumed that the image would be 400×400, and so the sub-images would be 80×80. The new code uses a sub-image width equal to the image width divided by five.

On a portable device, the screen is hand-held and its orientation can change. This is true of the puzzle game as well, but it works automatically in both orientations. Figure 11.9 shows the dog puzzle running on a phone in both horizontal (landscape) and vertical (portrait) orientations.



FIGURE 11.9 The dog puzzle game executing on a *Samsung-S* phone in both orientations.

Exporting an Android Game

A portable game is of little use unless people can get at it. Anyone with an *Android* phone knows about the online stores from which apps can be downloaded. Not everyone knows how to get software to that store, nor are they aware of the process involved in preparing software to be shared. We have a small game, the dog puzzle, and now will make it available for the world to use. The process is not a simple one: a digital signature, essentially an encryption key, must be created and applied to the *Java* archive, the archive must be built in the correct form, and it needs to be optimized and verified.

We are going to need some more software, much of which is already on your computer:

Keytool can be found in your *Java* SDK bin directory—for example, C:\Program Files\Java\jdk1.6.0_14\bin

This is the program that creates and manages the public encryption keys. RSA keys will be used for our *Android* applications.

Jarsigner can be found in your *Java* SDK bin directory.

This applies the encryption keys to a *Java* archive, a process called *signing*, and checks the validity of the result.

Zipalign can be found in your *Android* SDK tools directory—for example, C:\Program Files\Android\android-sdk\tools

This is a tool for optimizing the way *Android* application packages (APKs) are structured. It exists to increase efficiency and standardize the way the packages are organized.

We also need an *Android* build utility. Some people use *Eclipse*, but since we are compiling with *Processing* that's overkill. Ant builds *Java* applications and is a lot smaller than *Eclipse*. You can download Ant from <http://ant.apache.org/bindownload.cgi>

Before proceeding any further, it is important to remember that *Processing* and *Android* are based on *Java*, and that for the steps below to work properly the *Java* system must be installed properly. It is crucial that the paths to the tools are known to the system. On *Windows* the *Java* bin directory must be a part of the standard search path (named **PATH**) and it helps if the *Android* SDK directory wherein lies **zipalign** is there too. Also, make certain that **JAVA_HOME** is the path to the current *Java JRE* directory.

With these tools at your fingertips, here is the annotated process for creating a signed package, which is what is required for distribution, from the dog puzzle. It is a recipe that can be illuminated by looking at web-based documents describing the process in more detail.

1. With **dog.pde** loaded in a *Processing* sketch window, select the *File* menu, then *Export Android Project*.

This will create a new directory with everything needed to make a distribution package.

2. Open a *Windows* command prompt (terminal window) or the Mac equivalent, and change to the directory that was just created. It will be called *Android* or *Android.mmmm* for some number *mmmm*. Call this the **package** directory.

```
cd /d C:\dog\Android
Volume Serial Number is 1A0D-4EB9

Directory of C:\Jim\Writing\Books\Processing Game Book\Chap
t0\Android.131201.1657

12/01/2013  05:17 PM    <DIR>          .
12/01/2013  05:17 PM    <DIR>          ..
12/01/2013  05:17 PM                598 AndroidManifest.xml
12/01/2013  04:57 PM                42 ant.properties
12/01/2013  04:57 PM    <DIR>          assets
12/01/2013  04:57 PM                639 build.xml
12/01/2013  04:57 PM    <DIR>          libs
12/01/2013  04:57 PM                46 local.properties
12/01/2013  04:57 PM    <DIR>          project.properties
12/01/2013  04:57 PM                108 res
12/01/2013  04:57 PM    <DIR>          src
5 File(s)      1,425 bytes
6 Dir(s)   102,223,900,032 bytes free
```

3. Enter the following command:

```
keytool -genkey -v -keystore dog2-release-key.keystore -alias
jim -keyalg RSA -keysize 2048 -validity 10000
```

```
Re-enter new password:
What is your first and last name?
[Unknown]: Jim Parker
What is the name of your organizational unit?
[Unknown]: MinkHollow Media
What is the name of your organization?
[Unknown]: MinkHollow media
What is the name of your City or Locality?
[Unknown]: Cochrane
What is the name of your State or Province?
[Unknown]: Alberta
What is the two-letter country code for this unit?
[Unknown]: CA
Is CN=jim Parker, OU=MinkHollow Media, O=MinkHollow media,
a, C=CA correct?
[In]: yes
Generating 2,048 bit RSA key pair and self-signed certificate
a validity of 10,000 days
for: CN=jim Parker, OU=MinkHollow Media, O=MinkHollow media,
S=Alberta, C=CA
Enter key password for <jim>
[RETURN if same as keystore password]:
[Starting dog2-release-key keystore]
```

The name **dog2-release-key.keystore** is a file that will be created in the package directory, and the name **jim** can be any name, preferably yours. This creates a key (signature) and saves it in the named file. **Keytool** will ask you for a password. Give it one and do not forget it—you'll need it later. It will also ask you a bunch of questions, and then display your answers and ask for confirmation. Type *yes* and wait for it to complete.

```
-set-release-mode:
[echo] ****
[echo] ***** Android Manifest has debuggable=true ****
[echo] ***** Doing DEBUG packaging with RELEASE keys ****
[echo] *****

[getbuiltno1] Using latest Build Tools: 18.1.1
[echo] Resolving Build Target for dog2...
[gettarget1] Project Target:   Android 2.3.3
[gettarget1] API level:        10
[gettarget1] WARNING: Attribute minSdkVersion in AndroidManifest.xml
[gettarget1] project target API level <10>
[echo]
[echo] Creating output directories if needed...
[mkdir1] Created dir: C:\jim\Writing\Books\Processing Games\dog2\Android.131201.1657\bin\rsObj
[mkdir1] Created dir: C:\jim\Writing\Books\Processing Games\dog2\Android.131201.1657\bin\rsLibs
[echo]
[echo] Resolving Dependencies for dog2...
[dependency] Library dependencies:
[dependency] No Libraries
[dependency] -----
[dependency] API<=15: Adding annotations.jar to the classpath
[echo]

release:
```

4. Now type `ant release;` `ant` will execute, print a bunch of stuff, and (if you have good fortune) end with `BUILD SUCCESSFUL`. (The window on the left has been abbreviated)

5. We now use the **jarsigner**. Enter the command:

```
jarsigner -verbose -keystore dog2-release-key.keystore pack-
age\dog2-release-unsigned.apk jim where package is the complete
directory path for that directory and jim is the name used when running
the keysigner. It will ask you for the password, so enter it.
```

```
C:\Jin\Writing\Books\Processing Game Book\Chapter11-Android
1657>jarsigner -verbose -keystore dog2-release-key.keystore
  'Processing Game Book\Chapter11-Android\dog2\android.131201
  \signed.apk' jin
Enter Passphrase for keystore:
adding: META-INF/MANIFEST.MF
adding: META-INF/JIM-SF
adding: META-INF/JIM-RSA
signing: assets/dog.jpg
signing: assets/dog400.jpg
signing: assets/fescue5x5.jpg
signing: res/drawable/icon.png
signing: res/layout/main.xml
signing: AndroidManifest.xml
signing: resources.arsc
signing: res/drawable-hdpi/icon.png
signing: res/drawable-ldpi/icon.png
signing: classes.dex
signing: processing/opengl/ColorFrag.glsl
signing: processing/opengl/ColorVert.glsl
signing: processing/opengl/LightVert.glsl
signing: processing/opengl/LineFrag.glsl
signing: processing/opengl/LineVert.glsl
signing: processing/opengl/MaskFrag.glsl
signing: processing/opengl/PointFrag.glsl
signing: processing/opengl/PointVert.glsl
signing: processing/opengl/TextureFrag.glsl
signing: processing/opengl/TextureVert.glsl
```

6. Now check the archive using jarsigner again, using the command:

```
jarsigner -verify package\dog2-release-unsigned.apk
```

With luck, the response will be jar verified

```
1657>jarsigner -verify "C:\Jin\Writing\Books\Processing Game
  Book\Chapter11-Android\dog2\android.131201.1657\bin\dog2-release-unsigned.apk"
jar verified.
C:\Jin\Writing\Books\Processing Game Book\Chapter11-Android
```

7. Finally, use zipalign:

```
zipalign -v 4 package/bin/dog2-release-unsigned.apk dog2.apk
```

```
50 META-INF/MANIFEST.MF <OK - compressed>
873 META-INF/JIM-SF <OK - compressed>
1722 META-INF/JIM-RSA <OK - compressed>
2912 assets/dog.jpg <OK>
97172 assets/dog400.jpg <OK>
222360 assets/fescue5x5.jpg <OK>
28572 res/drawable/icon.png <OK>
27455 res/layout/main.xml <OK - compressed>
287279 AndroidManifest.xml <OK - compressed>
288368 resources.arsc <OK>
289340 res/drawable-hdpi/icon.png <OK>
292512 res/drawable-ldpi/icon.png <OK>
294398 classes.dex <OK - compressed>
563466 processing/opengl/ColorFrag.glsl <OK - compressed>
564083 processing/opengl/ColorVert.glsl <OK - compressed>
564733 processing/opengl/LightVert.glsl <OK - compressed>
566133 processing/opengl/LineFrag.glsl <OK - compressed>
567031 processing/opengl/LineVert.glsl <OK - compressed>
568325 processing/opengl/MaskFrag.glsl <OK - compressed>
569047 processing/opengl/PointFrag.glsl <OK - compressed>
569664 processing/opengl/PointVert.glsl <OK - compressed>
579572 processing/opengl/TextureFrag.glsl <OK - compressed>
572329 processing/opengl/TextureVert.glsl <OK - compressed>
```

This will create the final result, the file **dog2.apk**, in the **package** directory. This is the *Android* file ready for downloading.

To download this to your phone, place it on the Web. Make certain that the settings on the target phone allow the option **install from unknown sources**. This puzzle game can be downloaded from www.minkhollow.ca/dog and is the file **dog2.apk**.

Sound in *Android*

You may recall from Chapter 3 that sound is not a part of *Processing*, and that an additional library was needed to play sounds. The *Minim* package was used. Sadly, as useful as *Minim* is it has not been ported to *Android*, although it may be at some time in the future. In order to play audio files we will have to use another package. There are a few to choose from, but probably the simplest one to use is the *APWidgets* package and the provided *APMediaPlayer* class. This package is exceptionally valuable for *Android* programmers as it gives a *Processing* developer access to the common *Android* interface objects (widgets) including *Button*, *RadioButton*, *EditText*, and sound and video display. We're interested in the sound part only at this point, but the others may be valuable in the future.

Let's download and install *APWidgets* and then do a quick example.

Installing *APWidgets*

The first step is to download the current version of *APWidgets* from the Internet; the revision at the time of writing was 44 and it was found at:

http://apwidgets.googlecode.com/files/apwidgets_r44_for_Java1.5.zip.

Unzip this in its own directory, following the suggestions on the website

How to Install a Contributed Library

<http://forum.processing.org/one/topic/how-to-install-a-contributed-library.html>

Finally, critical step is to add the code to the *Android* sketch. Select the *Sketch/Add File* menu from the *Processing* sketch window and navigate to the directory where you installed *APWidgets*. Inside that directory within **apwidgets/library** is the file **apwidgets.jar**, which is the one you need to add. Select it and you should be ready to go.

Using the *APMediaPlayer*

APWidgets contains a lot of code, but we're interested only in the *APMediaPlayer* at this time, which is the class that plays sound files. To explain the use of this class let's try to write a program that will play a music file. In addition, it will display a progress bar showing how much of the song has been played. As we did when using *Minim*, we have to import the *APWidgets* package at the beginning of the program:

```
import apwidgets.*;
```

Next we'll need an instance of the `APMediaPlayer` for each sound we wish to play, in this case just the song. We'll also declare a variable that tells when the music is playing:

```
APMediaPlayer song;
boolean playing = true;
```

The `setup()` procedure must create an instance of the player, connect it to the file (called **song.mp3**) and start it playing. It will also set the volume to maximum:

```
void setup()
{
    song = new APMediaPlayer(this); // create new player object
    // Don't forget to copy the mp3 file into the sketch!
    song.setMediaFile("song.mp3");
    // set the file  song.start();
    // start play back
    song.setVolume(1.0, 1.0);    // Volume, both channels.
}
```

This code shows how to open the MP3 file and play it `setMediaFile()` and `start()`. It's most of what we need. The `draw()` procedure will create a rectangular region with colors showing how much of the song has been played and how much remains. The function `getCurrentPosition()` within `APMediaPlayer` tells us where we are in the file, and `getDuration()` tells us how big the MP3 file is altogether; thus, `getCurrentPosition() / getDuration()` tells us what fraction of the song has been played and can be used to draw the progress bar. The `draw()` function also prints those values out as numbers on the screen:

```
void draw()
{
    int d=0;

    background(30, 128, 180); //set background color (blue)
    fill (0, 200, 0);         // Box to show progress of song
    rect (100, 300, 200, 30);
    fill (200, 0, 0);
    d = (int) (200*(float)song.getCurrentPosition() /
                (float)song.getDuration());
    rect (100,300,d,30);      // The part that is played
    fill (0);                 // Text description
```

```

    text("Music position is "+song.getCurrentPosition(),
        100, 260);
    text(" out of a total "+song.getDuration(), 233, 260);
}

```

The *Android* system is relatively small and likes us to clean up after our code has completed. The code that does this is simple, and can be used as a template for more complex programs:

```

public void onDestroy()
{
    super.onDestroy();
    if(song!=null) song.release(); // Do this for each sound.
}

```

Every instance of `AMediaPlayer` that has been created needs to be released in this way. The call to `super.onDestroy()` deals with the cleanup is the super class, and needs to be called only once.

Finally, let's introduce a little interaction. When the lower half of the screen is touched the music will pause, or if already paused it will resume playing. This lets us test the `pause()` function. If the upper part of the screen is touched then the music will stop playing and the program will end. As you may find out, terminating an *Android* sketch is tricky in that the emulator keeps running independently of the *Processing* system. You should always have a way to terminate a sketch during the development phase.

The code that implements this interaction is placed within the procedure `mousePressed()`:

```

void mousePressed()
{
    if (mouseY < 400)          // Upper part of screen
    {
        onDestroy();           // Cleanup and exit.
        exit();
    } else
    {
        if (playing)           // Lower part of the screen
        {
            song.pause();      // If playing, then pause.
            playing = false;
        } else
        {

```

```
    song.start();      // If paused, then play.  
    playing = true;  
}  
}  
}
```

Before this program can be successfully executed, the MP3 file has to be included in the sketch, using the **Sketch/Add Files** menu as we did with images. Figure 11.10 shows the *Android* emulator screen while this program is executing and playing a song.

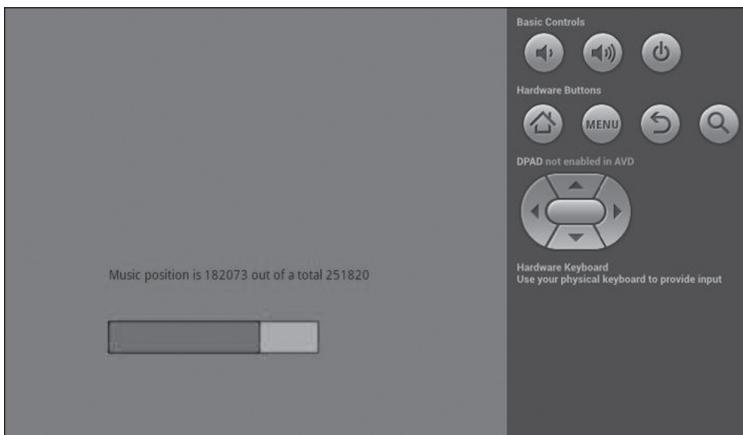


FIGURE 11.10 The sound player application being executed in the emulator.

Summary

Android is an open-source operating system based on *Linux* system that runs on the majority of cellular telephones and many other portable devices. *Android* programs are called apps (applications) and are usually coded in *Java*. There are differences in the way that standard *Java* sketches and *Android* sketches work in Processing, including that the touch screen events translate into mouse operations, screen sizes and orientations vary, and some packages (like *Minim*) are not compatible.

Developing an *Android* sketch begins by using an emulator that executes on your PC, then by executing the sketch on a real device connected to the PC by a USB cable, and finally by creating a *signed package* that can be uploaded to a mobile device. Sound can be displayed using the `APMedia-Player` object from the `APWidgets` package.

Exercises



The exercises below will apply your knowledge of the material in this chapter, and sometimes require that you do some more research before you are able to complete them. Solutions to selected exercises are available on the companion disc.

8. How can the variables **mouseX** and **mouseY** be used in an *Android* sketch? Either experiment by coding a few simple sketches to figure it out, or locate a definitive document on the Internet.
9. The variables **pmouseX** and **pmouseY** contain the X and Y value of the mouse coordinates in the previous frame. Use these to recognize simple *gestures*: that is, a sweep of the finger across the screen either left, right, up, or down. These are gestures basic to touch screen processing.
10. Build and test a sketch that begins with the bouncing yellow ball and adds a sound effect (e.g. a click or thud) whenever the ball bounces.
11. When using the emulator, typing **7** on the number pad will change the orientation (on a Mac, Ctl-F12 or Ctl-Fn-F12). Use the variables `displayWidth` and `displayHeight` to determine orientation in a sample sketch, and to set a variable `orientation` to either `LANDSCAPE (=0)` or `PORTRAIT (=1)`.
12. The emulator allows a real keyboard. In the AVD Manager there is a box that selects this; do so and try to process simple character input by moving a square across the screen under control of the **WASD** keys. If you have a mobile device with a keyboard, try to execute this program on that device.

Resources

The *Android* SDK: <http://developer.android.com/sdk/index.html>

Getting Started with *Processing* for *Android*: <http://createdigitalmotion.com/2010/09/getting-started-with-processing-for-android/>

Basic Mobile *Processing* tutorial: http://ericmedine.com/tute_mobileProcessing
Sound for *Android*:

- <http://realmike.org/blog/2010/12/11/sound-playback-in-processing-for-android/>

- <http://stackoverflow.com/questions/17386928/working-with-mp3-sound-in-processing-for-android-need-to-stop>

Building a signed .apk from *Processing*: <http://www.akeric.com/blog/?p=1352.html>

AP Widgets Download: http://apwidgets.googlecode.com/files/apwidgets_r44_for_Java1.5.zip

Source code for *APMediaPlayer*: <http://code.google.com/p/apwidgets/source/browse/trunk/apwidgets/src/apwidgets/APMediaPlayer.java?r=44>

Bibliography

Darcy, Lauren, and Shane Conder. *Teach Yourself Android Application development*. Indianapolis, IN: Sams Publishing, 2010.

Hall, Steven. *Getting Started with iPhone SDK, Android and Others: Mobile Application Development - Create Your Mobile Applications Best Practices Guide*. Newstead, AUS: Emereo Publishing, 2008.

Khan, Aatif. *What Is Zipalign In Android And How To Make Apps Zipaligned*. November 9, 2010. <http://www.addictivetips.com/mobile/what-is-zipalign-in-android-and-how-it-works-complete-guide/> (accessed 2014).

Open Web Application Security Project. *Signing jar files with jarsigner*. September 21, 2008. https://www.owasp.org/index.php/Signing_jar_files_with_jarsigner (accessed 2014).

Sadun, Erica. *The iPhone Developer's Cookbook*. Boston, MA: Addison-Wesley, 2009.

Sauter, Daniel. *Rapid Android Development: Build Rich, Sensor-Based Applications with Processing*. Pragmatic Bookshelf, 2013.

TeacherBen. *Installing a Processing sketch on your Android device*. April 12, 2012. <http://teacherben.com/?p=223> (accessed 2014).

ABOUT THE CD-ROM

The disc contains example programs and illustrative code, references, supplementary material, and all of the figures in the book as color images. In directories named for each chapter are subdirectories containing source code and assets for most of the example in the book. Here are the subdirectories and a brief description of their contents. All of this material can be navigated by using the web interface beginning at the file ‘index.html’ in the main directory. When this file is double clicked the main web page will open, and will look like this:

Clicking on any of the links, such as Chapter 1, will take you to the page for that chapter.

Each of these pages contains links to figures, example code from that chapter, resources, and references. An index at the top of each page has links to each section, but the page can be scrolled as well.

Processing examples from each chapter are as follows:

Chapter 1

No code in this chapter.

Chapter 2

- chapter0201 Display an image
- chapter0202 Display an image many times, as a 2D grid
- chapter0203 Draw three ellipses
- chapter0204 Draw text in four different sizes
- chapter0205 Read an image and swap red for green in each pixel, then display.
- chapter0206 Draw a circle into a PGraphics object, display the PGraphics object twice.

Chapter 3

- chapter0301 Play an MP3 file.
- chapter0302 Generate and play a mix of three signals.
- chapter0303 Monitor sound playing and display left and right volume bars.
- chapter0304 The ‘simpleAudio’ system and an example of positional audio. Using wasd keys you can move the listener around the source.
- chapter0305 Play a note (A440). Use ‘simpleAudio’ to allow the user to move the listener as before, and change orientation using arrow keys.
- chapter0306 As in chapter0305.pde, but the keyboard controls the source position instead of the listener.

Chapter 4

In the **hockey** directory are four sub-directories named *proto0*, *proto1*, *proto2*, and *proto3*. These contain the prototype versions of *Hockey Pong* with *proto3* being the final version.

Chapter 5

- a0triangle Draw a triangle by drawing 3D lines.
- a1triangle Draw a triangle by specifying three 3D vertices.
- a3triangle Draw a triangle in 3D and move it closer and more distant.
- a4triangle Draw a triangle in 3D and allow the user to change the view point using the WASD keys.
- a5triangle As in chapter0504, but the AD keys change orientation, not the X coordinate.
- a6quad Draw 4 sides of a prism, allow user to walk about it in 3D.
- a7prisms Draw seven prisms, allow user to walk about it in 3D.

- a10buildings Draw many prisms and texture map them to look like buildings. The user can walk through using WASD.
- a11object Read in and display a 3D model of a motorcycle. Warning: quite slow.

Chapter 6

collision1D The 1D collision simulation described in the chapter.

Chapter 7

No code in this chapter.

Chapter 8

The directory *prototype02* contains source code and assets for the SMV Rainbow game.

Chapter 9

NOTE *These programs are executed through a browser, and must be done through a web server. Either install your own or place these directories online.*

- example00 First simple processing.js example, draw a circle and rectangle. The HTML file and processing.js library are in the same directory.
- example03 Simple examples of JavaScript. Example03.html opens a page to Google. Example03a.html opens a selection box allowing the user to choose to go to Google.
- example04 Open a new tab when the mouse is clicked.
- proto0 First prototype of *Hockey Pong* for the web.
- proto3 Final version of *Hockey Pong* for the web.

Chapter 10

balloon	Animation of the balloon popping.
billiards	Pool table animation
control	Multiple animations placed in 2D locations. The control room animation
gait	Animation of a character walking
steam	Animation of steam emitting from a jet, placed in a still frame.

Chapter 11

ball	Android simulation of a ball bouncing.
dog2	Android puzzle: unscramble an image.
sound	Play an MP3 file through Android.

Appendix A

cubic	Cubic interpolation
intersect	Calculate and print the point of intersection between two lines.
matrixMult	Program for a 3x3 matrix multiply.
rotate	Rotation example: a ball will move to strike a bat (when space key is pressed) causing it to rotate.

Appendix B

test00	Set background color to red.
test01	Draw a blue rectangle on a red background.
test02	Draw a blue circle on a red background.
test03	Draw a blue circle on a red background. The circle is drawn at the coordinates of the mouse.

- test04 Draw a blue circle on a red background. The circle is drawn at the coordinates of the mouse. When the left mouse button is clicked, the mouse stays fixed on that location of the screen.
- test05 Small yellow circle moves about the screen and bounces off of the sides of the sketch window.
- test06 Many small yellow circles move about the screen and bounce off of the sides of the sketch window.
- test07 Display a JPG image in a window of the correct size.
- test08 Read and display an image; find yellow pixels and move them to a new location at random, each frame starts over.
- test09 As in test08, but don't start over. The yellow pixels keep moving and eventually occupy random screen locations.

APPENDIX A

MATHEMATICS TUTORIAL FOR GAME DEVELOPMENT

Mathematics is a necessary prerequisite to building a computer game. Why? Because computers are basically calculating devices, and games are essentially simulations of a world whose mathematical properties we understand. This insight comes from Bertrand Russell, who said that “*physics is mathematical not because we know so much about the physical world, but because we know so little: it is only its mathematical properties that we can discover.*” The sad truth about games thus far is that it is only their mathematical properties that we can easily manipulate.

Mathematics is not everyone’s ‘cup of tea,’ as it were. So, as a reference for this book, here is the minimum amount of math that is needed for games. OK, some of it is really physics, but it is expressed as equations, so I think it’s safe to think of it as math.

There are some things about mathematics that you should know, and that few teachers will tell you. You are expected to figure them out by long practice.

Symbols in mathematics are used to avoid long winded verbal

explanations. It makes math look mysterious and complex, but really all they are doing is using a different, more precise, and denser alphabet. The simplest example may be the equal sign ‘=’, which was invented by Robert Recorde in 1557. He did so to avoid writing, over and over again, the phrase “is equal to”.¹

¹. “To avoide the tedious repetition of these woordes: is equalle to: I will settle as I doe often in woorke use, a paire of parallels, or gemowe [twin] lines of one lengthe: =, because noe. 2. thynges, can be moare equalle.” --Robert Recorde, *The Whetstone of Witte* (1557).

Another example, more contrived, is the phrase “the sum of all integer values of x multiplied by itself between 0 and 100” becomes, in math notation

$$\sum_{x=0}^{100} x^2$$

The Greek symbol *sigma* (letter ‘s’) is used to mean *sum*, and the lower and upper bounds of the sum are placed below and above the sigma respectively. What we are summing are the values of ‘ x squared,’ which is shorthand notation for ‘ x multiplied by x .’ The concept of this sum is simple, whereas the notation frightens some people. Any math equation can be translated into English, perhaps losing some precision in the process, but often not.

- 1. Things (concepts, values, operations) that have proven to be valuable in multiple ways are given special names, and sometimes special symbols.** So, for example, the value PI (π) has a special name because it is a very important number. A “dot product” is a relatively simple idea, but has a special name and a symbol to go with it because it is a frequently performed operation. The summation symbol above is another such. In this way, math becomes even more compact and harder to grasp for those learning the symbols. Remember, it is the concepts that really matter.
- 2. Pictures are important.** For any given math problem there are pictures that can explain it, and those pictures often suggest a solution. Draw the problem showing everything that you know as an initial step in getting an answer. Many mathematicians do this in their head and then write the equations down, so you don’t see this step.
- 3. Letters are used to represent values so the solution is general.** A specific solution to the problem “find a point that is a distance 10 from the point (50, 50)” would be (60, 50), or perhaps (50,60). More generally there are many points that are 10 units from any particular point, say (\mathbf{a}, \mathbf{b}) , and they define a circle! Any point on that circle is an answer to the problem. Indeed, it can be generalized even more by specifying the distance as r rather than 10. So the equation of a circle is

$$(x - \mathbf{a})^2 + (y - \mathbf{b})^2 = r^2$$

Where the centre of the circle is (\mathbf{a}, \mathbf{b}) and the radius is \mathbf{r} . This formula is general, and allows us to see *all* solutions for any value of \mathbf{r} , any

point (**a, b**). Algebra is used when there could be many solutions and we might need any or all of them. The letters in algebra are really like variables in *Java* or *Processing*.

- The relationship between mathematics and science, especially physics, is complex, but not rooted in fundamental reality. **We use math to build models of processes**, and if those models accurately predict new situations we are pleased and call them ‘correct.’ These models rarely tell us about ‘how’ things work, and even less often ‘why.’ The equations of physics are observed relationships and are embodied in equations. The equation $\mathbf{f} = \mathbf{ma}$ gives an observed relationship between force and mass; it does not explain why that relationship is true or how the Universe makes it so.

Elementary Trigonometry

The ‘tri’ in trigonometry means triangle; trig is all about triangles, and the relationships between angles and the lengths of sides. It is one of the most useful areas of mathematics.

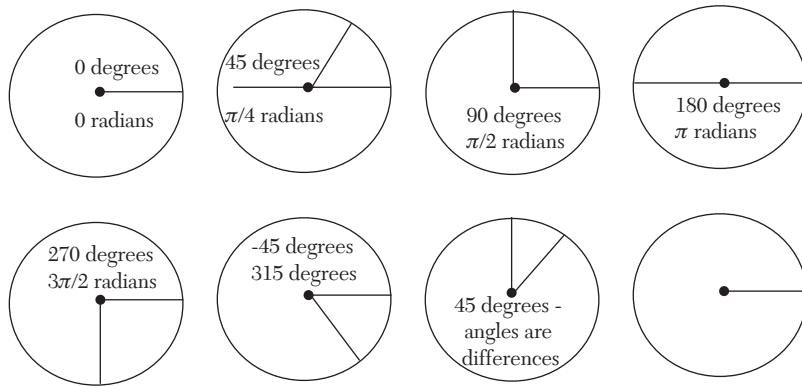


FIGURE A1.1 Angles, degrees, and radians

An angle can be thought of as a difference in direction, as on a compass (Figure A1.1). There is an angle that can be measured between any two lines that intersect, and there are two main measures of angle magnitude: *degrees* and *radians*. Let’s imagine a clock. A horizontal line is one from the clock center to the number 3, and this will be considered to be a baseline. We’ll call it zero, either zero degrees or zero radians. Also, let this be the hour hand of the clock. As is easy to imagine, the angle between this

horizontal line and a moving minute hand will increase as the minute hand advances until it moves all the way around the clock face and meets the 3 o'clock line again, and then it is zero once more. The total number of degrees in a complete rotation around the circle is 360 (because of an old Babylonian number system); the number of radians in the same circle is 2π .

Let's use degrees as a measure from now on, and note that there can be negative angles: -45 degrees is +315 degrees (because $360 - 45 = 315$). Also note that angles are differences between two directions, although sometimes a standard reference is made to a zero angle. That's a lot of words for a simple idea. A lot of people think that trig is hard, and so let's go through this slowly. If you understand then skip to the next section.

The sum of the angles inside of any triangle is a constant -180 degrees. So if we know any two angles, then we can compute the third. On the other hand, the lengths of the sides do not have any similar property: an equilateral triangle can be one inch or one mile on each side, the angles are all still 60 degrees.

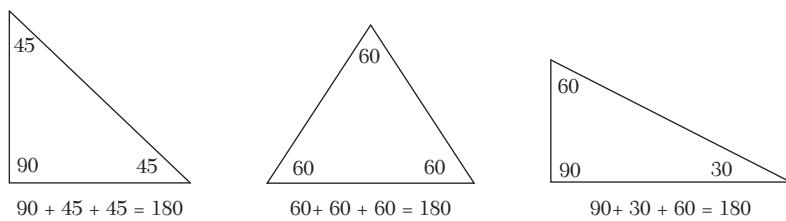


FIGURE A1.2 The sum of the internal angles in a triangle is 180 degrees.

Trig allows us to compute the lengths of sides of a triangle given some of the other sides and angles. To do this we need to know the values of some simple functions, called *trigonometric* functions. The ones named sine, cosine, and tangent are the most important. These functions we look up in a book or, better yet, use a scientific calculator or computer. (We can calculate them using series expansions, but why bother.)

Now, here are some very simple but very important rules to know when using these functions. These rules all apply to any *right* triangle (triangle with a 90 degree angle in it). The *hypotenuse* is the side opposite to the right angle.

1. The sine of any angle in a right triangle equals the length of the side opposite to the angle divided by the length of the hypotenuse.

2. The cosine of any angle in a right triangle equals the length of the side adjacent to the angle (not the hypotenuse) divided by the length of the hypotenuse.
3. The tangent of any angle in a right triangle equals the length of the side opposite to the angle divided by the length of the side adjacent to the angle.

Here come the examples:

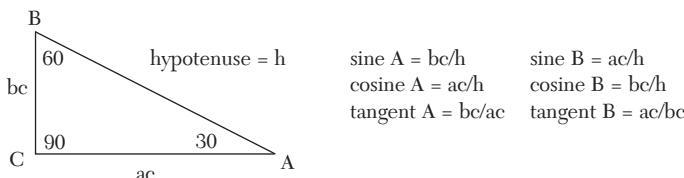


FIGURE A1.3 A definition of the trigonometric functions on a right triangle.

Of course we already know the values of sine, cosine, and tangent (or sin, cos, tan, as they are abbreviated) for any angle – our calculator has these, and before that we used slide rules and tables. What we do is use one of the equations and solve for what we don't know, but want to know. If we have a length we can find the others.

EXAMPLE

The sun is 25 degrees above the horizon. How long is the shadow thrown by a 100 foot tall building? Figure A1.4 illustrates this problem.

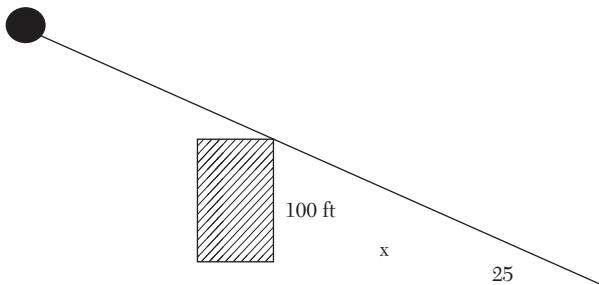


FIGURE A1.4 Compute the length of the shadow.

As usual, most of the problem is in the drawing of it (In math, physics, and computer science, always get a very clear statement of the problem to be solved. This usually does most of the work in solving it!)

So, we know an angle and the size opposite, and wish to know the side adjacent. The trigonometric function that uses the adjacent and opposite sides is the tangent, and the equation here would be

$$\tan 25 = 100/x$$

We solve for x :

$$x = 100/\tan 25$$

My calculator says that $\tan 25$ is about 0.47, so we conclude that the shadow length $x = 100/0.47 = 212$ feet. Does this seem correct? Well, if the sun were higher, the shadow would be shorter. So, if the sun were at 45 degrees we'd have $x = 100/\tan 45 = 100/1.0 = 100$ feet. It is a higher angle by almost double, having a shadow less than half the length of before. Well, seems consistent. This problem is quite typical of the sort that we will encounter in a game. We will know an angle and the length of a side, and will need to compute the length of another side. Depending on which side we wish to compute, we will use a sin, cosine, or tangent function.

That's enough for now. More complex stuff will be explained when we encounter it.

Straight Lines

Most of you will be familiar with the Cartesian coordinate system in a plane, and you can probably see the logical extensions into three and more dimensions. Each point, line, triangle, and object in a game is specified in terms of coordinates in a system of this type. So, a point is expressed as three numbers (X, Y, Z); a line segment is two points, a triangle three points/three segments, and so on.

A common equation for a straight line is the slope-intercept form:

$$y = mx + b$$

where **m** is the line's *slope* ($\Delta y/\Delta x$) and **b** is the *intercept*, or the y coordinate of the point where the line crosses the y axis. Any values of X and Y that satisfy this equation lie on the line. So, if we specify a value of x it is easy to compute the value of y that would specify a point on that line. It is also said that the parameters **m** and **b** *define* the line: there is one line having those parameters. The main problem with this form of the line equation is that it does not work for vertical lines, for which the slope **m** would be infinite.

Either we treat vertical as a special case, or adopt a different line equation. The general form of the line is

$$\mathbf{Ax + By + C = 0}$$

where at least one of **A** or **B** must be non-zero. Now, there is a single line that can be drawn through any given two points; or, in other words, two points also define a line. It is a common process to determine the equation of the line that passes through two points (x_1, y_1) and (x_2, y_2) . Here's how it works:

1. If x_1 and x_2 are different, then the slope $\mathbf{m} = (\Delta y / \Delta x) = (y_2 - y_1) / (x_2 - x_1)$.

The two-point form of the line equation is:

$$\mathbf{y = y}_1 + [(y_2 - y_1) / (x_2 - x_1)](x - x_1)$$

2. If x_1 and x_2 are the same, then the line equation is $\mathbf{x = x}_1$.

Which Side of a Line is a Point on?

Given the equation of a line, any point is either on the line itself or on its left or its right. Assume that the choice of left and right are arbitrary, because we mostly care about figuring out whether two points are on the same side of a line. Turns out that if we substitute the coordinates of a point (r, s) into the equation then the result will be 0 (on the line), negative (on the right) or positive (on the left). So, if two points result in the same numerical sign when plugged into the line equation then they are on the same side.

Distance

The distance between a point $\mathbf{P} = (x_0, y_0)$ and a line \mathbf{L} given by $\mathbf{Ax + By + C = 0}$ is easy to determine. First, we choose the shortest distance, and that means that the line from the \mathbf{P} to the line \mathbf{L} makes a right angle (90 degrees). The formula for this becomes:

$$d = \frac{|ax_0 + by_0 + c|}{\sqrt{a^2 + b^2}}$$

Where Do Two Lines Intersect?

There are two lines, so there are two equations. If the lines are not parallel, then they intersect at the point (x, y) that is a solution to both equations; thus, to find this point we solve the pair of equations simultaneously:

$$A_0x + B_0y + C_0 = 0$$

$$A_1x + B_1y + C_1 = 0$$

A general solution to these has:

$$y = \frac{A_0C_1 - A_1C_0}{A_1B_0 - A_0B_1}$$

$$x = \frac{-B_0y - C_0}{A_0}$$

Two lines are parallel if they have no intersection, and so the system of equations has no real solution in that instance.

What Line Passes Between Two Points?

It seems clear that a unique line passes between any two points that you may select. Given points $P_0 = (x_0, y_0)$ and $P_1 = (x_1, y_1)$, what's the equation of the line through P_0 and P_1 ? A good way to determine this requires that we first notice that Equation 15.2 can be rewritten as:

$$y = -(A/B)x - C/B$$

When we rename $-(A/B)$ as \mathbf{m} and $-C/B$ as \mathbf{b} we get the slope-intercept form of the line of Equation 15.3: $y = mx + b$.

The variable \mathbf{m} represents the slope of the line, or the angle it makes with the x axis. The \mathbf{b} variable is the point on the y axis where it intersects the line. In any case, we now have two constants in the equation, and two points: we can solve this now. That is, solve the following pair of equations for \mathbf{m} and \mathbf{b} :

$$y_0 = m*x_0 + b$$

$$y_1 = m*x_1 + b$$

EXAMPLE

What line passes through the points (1, 1) and (5, 5)?

The equations are: $1 = \mathbf{m}*(1) + \mathbf{b}$ and $5 = \mathbf{m}*5 + \mathbf{b}$. Substituting for \mathbf{b} , we get

$$\mathbf{b} = 1 - \mathbf{m} = 5 - 5\mathbf{m}.$$

Solving for \mathbf{m} gives $\mathbf{m} = 5 - 5\mathbf{m} - 1; 4\mathbf{m} = 4$, or $\mathbf{m} = 1$.

Now take a point on the line (5,5) and the newly discovered value for \mathbf{m} and solve for \mathbf{b} : $5 = \mathbf{m}*5 + \mathbf{b} \rightarrow 5 = 1*5 + \mathbf{b} \rightarrow 5 - 5 = \mathbf{b}$, or $\mathbf{b} = 0$. The equation of the line through (1,1) and (5,5) is therefore $y = 1x + 0$ or just $y = x$.

Important fact: The slope \mathbf{m} is really the change in y divided by the change in x as we sample any two points on the line. So, recalling our trig and looking at the figure below we can see a useful relationship:

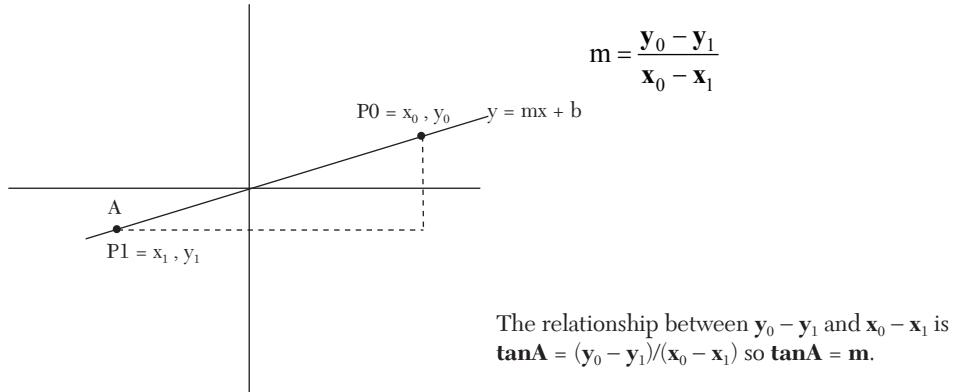


FIGURE A1.5 Point-slope equation of the line.

What is the Angle Between Two Lines?

Lines form angles with each axis, and with each other. It is sometimes important to determine what the angles are, perhaps so we can apply a trigonometric equation or a vector transformation. Let the two lines be $y = \mathbf{m}_1x + \mathbf{b}_1$ and $y = \mathbf{m}_2x + \mathbf{b}_2$. Assign the labels 1 and 2 so that $\mathbf{m}_2 > \mathbf{m}_1$. It can be shown that the angle Z between the two lines is given by

$$\tan Z = \frac{\mathbf{m}_1 - \mathbf{m}_2}{1 + \mathbf{m}_1 \mathbf{m}_2},$$

What is the Equation of a Line Perpendicular to Another?

The trig fact above makes it easy to find the equation of a line perpendicular to some other. The equation of our line is $y = mx + b$, and $\tan A = m$. The line perpendicular to this has inverse slope: that is, its slope will be $1/m$. All we need to do is find b , and to do that we simply select any point on the original line (x', y') and use slope $1/m$, then solve for the new $b' = y' - x'/m$.

Vectors

A vector is a set of values or coordinates that are specified in a given order. The number of values is called the *dimension* of the vector. A point in 2D Cartesian space is a two dimensional vector, and a point in 3D space is a three dimensional vector. Vectors are usually drawn as little arrows with a length and direction, as in Figure A1.6. Since a two dimensional vector has only 2 component values (x, y) , we draw them such that the start of the vector is always $(0, 0)$ and the other end is (x, y) . In actual fact this is just for convenience. Vectors are always relative, and can have an origin where ever you like; for example, at the center of a vehicle or animated object.

The length (also called the magnitude) of a vector $P = (x, y)$ is denoted by $\|P\|$ and is computed as:

$$P = \sqrt{x^2 + y^2}$$

The reason for this is easily seen by looking at Figure A1.6. The x and y values are really two sides of a right triangle, and the length is the hypotenuse. Simple trigonometry will show us that the angle θ of the vector relative to the X axis would be found as:

$$\theta = \tan^{-1}\left(\frac{y}{x}\right)$$

We can make a vector have a length of 1 without changing its direction; this is called *normalizing* the vector. Having a unit vector pointing, say, in the orientation direction of a car happens to be very useful sometimes, and we can make it any length we like by multiplying it by a new length. Anyway, normalizing is done like so:

$$x' = \frac{x}{P}$$

$$y' = \frac{y}{P}$$

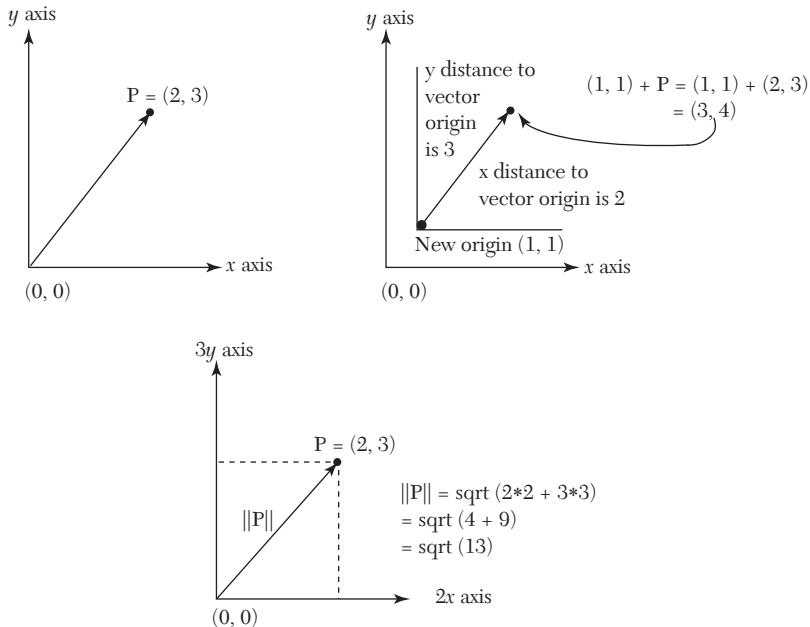


FIGURE A1.6 Vector stuff. (Upper left) A vector drawn at the origin. (upper right) Placing the vector at some other point $(1, 1)$ means adding the point to the vector. (left) The length or *norm* of the vector is the length of the hypotenuse.

Multiplying a vector by a number means multiplying all components of the vector by that number. What does this do to the length? Let's see: if we have a unit vector pointing along the X axis then its coordinates will be $(1, 0)$ and its length is obviously 1. Now multiple this vector by 2, giving us $(2, 0)$, and the length is now $L = (2*2 + 0*0)^{1/2} = 4^{1/2} = 2$. So, it seems that multiplying a vector by a constant increases the length of the result by that constant factor. So, as was said a little earlier, we can give a unit vector any desired length.

We add two vectors by summing the coordinates in each corresponding dimension. For example, the sum of $(1, 0)$ and $(0, 1)$ is $(1 + 0, 0 + 1) = (1, 1)$.

A line through point (a, b) parallel to vector (u, v) is given by

$$(x, y) = (a, b) + t(u, v)$$

where t is any real number. We can think of the phrase ‘parallel to’ as meaning the same thing as ‘in the direction of,’ and so it makes sense that any point that is some multiple of (u, v) offset from (a, b) will be on the line.

Dot Product

The dot product between two vectors is a scalar number. It is computed in a couple of possible ways:

$$A \cdot B = \|A\| \|B\| \cos \theta$$

$$A \cdot B = \sum_{i=1}^n A_i B_i$$

Why is this a useful thing? Well, first we can quickly calculate the angle between two vectors by computing the inverse cosine of the dot product divided by the products of the norms. More importantly in the game domain, it is used in *back face culling*. The dot product of the normal of a polygon, even a triangle, with the vector from the camera to one of the polygon's vertices is greater than zero if the polygon is facing away from the camera. Polygons that face away from the camera cannot be seen, so this calculation is a simple visibility algorithm.

Amazingly, the dot product is invariant under rotation: two vectors keep the same dot product.

Cross Product

A cross product, or *vector product*, between two vectors A and B results in a vector that is perpendicular to both A and B. If we know all of the components of A and B then the cross product $C = A \times B$ is computed as

$$C_x = A_y B_z - A_z B_y$$

$$C_y = A_z B_x - A_x B_z$$

$$C_z = A_x B_y - A_y B_x$$

Why is this useful? Well, for one thing, we can find a normal vector to a polygon (a triangle in particular) by finding the cross product between two adjacent edges. This is, in turn, used in illumination algorithms and shading. Another use for a cross product is the computation of rotational forces in physics (*torque*).

The Plane

A plane is a line that has another dimension, giving it area. This is opposed to a 3D line, which is a line *within* another dimension. More on this later.

An equation that defines a general plane is:

$$Ax + By + Cz + D = 0$$

Notice that this is the equation of a line, but with an extra variable z . In fact, most of the properties of a line are also properties of a plane: they have slopes (with respect to an axis), they can meet at an angle, and so on. A plane divides space into two parts, as a line divides a plane, and we can determine which side of a plane a point is on by substituting the point's coordinates into the equation of the plane, as we did with the line, and look at the sign of the result. There are, however, a couple of interesting new things.

Planes are essential to games because a polygon (quad or triangle) is a plane in 3D space, or at least a part of a plane. So everything that we draw is a plane, and properties of planes are used to render our images.

Normal to a Plane

A normal to a plane is a line or vector that is perpendicular to the plane in all orientations. The vector (A, B, C) is a normal to the plane $Ax + By + Cz + D = 0$; at least it points in the direction of the normal, which is the important part. As we saw in the section on cross products (above), we can compute the cross product of any two adjacent edges of any polygon that lies in the plane, and that will be a normal to the plane too.

Back-face and culling

Just as a point (x, y) was on one side or the other of a line depending on the sign of the result of a substitution of (x, y) into the line equation, a point (x, y, z) can be located on one side or the other of a plane surface according to the sign of the plane equation. If $Ax + By + Cz + D < 0$, then I say the point (x, y, z) is North of the plane surface; if $Ax + By + Cz + D > 0$, then the point (x, y, z) is South of the plane.

I briefly mentioned back face culling when talking about the dot product. In general, half of all polygons in a scene will not be visible by the camera. Eliminating these will obviously improve the performance of the scene renderer.

What I said before was that the dot product of the normal of a triangle with the vector from the camera to one of the polygon's vertices is greater

than zero if the polygon is facing away from the camera. We get the plane's normal using the cross product, and computer the dot product to a vector from the camera's position to a point on the plane. If the angle is between 90 and 270 degrees, then the polygon is facing the camera - the dot product will be negative. Otherwise, the triangle is not facing the camera, and can be ignored.

Transformations

The transformations we are about to discuss are essentially geometric.

Reflection

Given the functional equation $y = f(x)$, it is possible to reflect the function about the Y axis by changing the sign on the x variable; that is,

$$y_{\text{reflected}} = f(-x)$$

So, consider the equation of a line: $Ax*By*C = 0$ This can be converted into the form $y = f(x)$ using some simple algebra, and we get

$$y = f(x) = -(Ax+C)/B$$

The reflection of this line about the y axis is $f(-x)$, which is to say:

$$y = f(-x) = (Ax - C)/B$$

Reflection about the x axis means changing the sign on the y variable, such as

$$y = -f(x)$$

Scaling

A change in size or scale is done by multiplying the scaled coordinates by a scale *factor*, a real number indicating what the magnitude of the change will be. In addition, the scale factor can be different for each coordinate, meaning that scaling, like reflection, is actually dependent on the axis.

Scaling parallel to the x-axis is accomplished by multiplying the x variable by the scale factor:

$$y = f(\text{factor} * x)$$

It is possible to scale in each axis by a different amount, but it is not often useful to do so.

Translation

Translation involves changing the positional coordinates or variables. For a shape, this means that the object seems to change position. Each axis, again, likely has a different translation, and the translation itself is accomplished by adding a distance to the variable. So, moving a function or object along the **X** axis by **a** units is done by:

$$y = f(x + a)$$

This can be undone by adding $-a$ to all x values, obviously. To perform a translation along the y axis we add the translation to y ; that is:

$$y = f(x) + a$$

Basic Matrix Operations

A matrix is a two dimensional array of numbers that usually represents a set of linear equations. There is a specific way of manipulating matrices that is not always intuitive for a beginner. This is a problem in games work because there are many useful algorithms that use matrices: rotation of graphical objects, shortest path through a graph, and most splines are examples. In fact, when you think about it an image is just a huge matrix of pixel values.

So, a matrix has a certain number of rows and a certain number of columns, and is defined by those values: a rotation can be accomplished using a 3×3 matrix, for instance, specifying 3 rows and 3 columns. Here how it looks:

$$m = \begin{bmatrix} 1 & 2 & 0 \\ 3 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Elements in a matrix are indexed by using two integers: the row, then the column. Thus, in the matrix **M** above, the value of the element at **M(1,2)** is 2, and the value at **M(2, 1)** is 3.

Adding matrices together can only be done if the two matrices have the same number of rows and the same number of columns. Then the sum is simply the sum of the corresponding elements:

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

Multiplying matrices is much more difficult than adding them, and much less intuitive. Basically, an element in the product is the sum of the products of elements in a row of one matrix and a column of the other. The simplest case is a 1 row matrix multiplied by a 1 column matrix. The result is a 1x1 matrix, and the calculation is:

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = [1 \times 1 + 2 \times 2 + 3 \times 3] = [14]$$

This same pattern is repeated in a general NxM matrix multiply for each element in the product. That is, if we multiply matrices A and B giving C ($C = A * B$) then:

$$C_{i,j} = \sum_{k=1}^N A_{i,k} B_{k,j}$$

The number of columns of A and the number of rows of B must be equal for the multiply to be possible. Note that this is a vector dot product of a row or A with a column of B. The pattern of row X column is illustrated in Figure A1.7. In that figure, the shaded row and column on the left of the '=' multiply and sum to yield the shaded cell on the right of the '=' sign.

It is also important to note that AxB is not the same as BxA when A and B are matrices. So, order matters. In particular, since matrices can be used to represent scaling, translating, and rotating an object, you must remember to pay attention to the order. The expression $RxTxSxA$, where A is an object, R is a rotation matrix, T is a translation matrix and S is a scale matrix, performs the scale first, then translates, then rotates. This is quite a different thing from $SxTxRxA$.

Matrices are often represented as two dimensional arrays in programming languages. A typical example would be:

```
float rotationMatrix[][] = new float[4][4];
```

For a game the bounds are nearly always known at the outset, but if not they can be generated dynamically. A function that multiplies two matrices together is (**matrixMult.pde**):

```

float [][] matrixMultiply (float x[][], float y[][])
{
    float res [], z;
    int n, m, kk;

    kk = x[0].length;
    if (x[0].length != y.length)
    {
        println ("Incompatible.");
        return null;
    }
    n = x.length; m = y[0].length;

    res = new float[n][m];
    for (int i=0; i<n; i++)
        for (int j = 0; j<m; j++)
    {
        z = 0.0;
        for (int k = 0; k<kk; k++)
            z = z + x[i][k] * y[k][j];
        res[i][j] = z;
    }

    return res;
}

```

The function returns a matrix of the correct size if the multiplication was completed, otherwise it returns **null**.

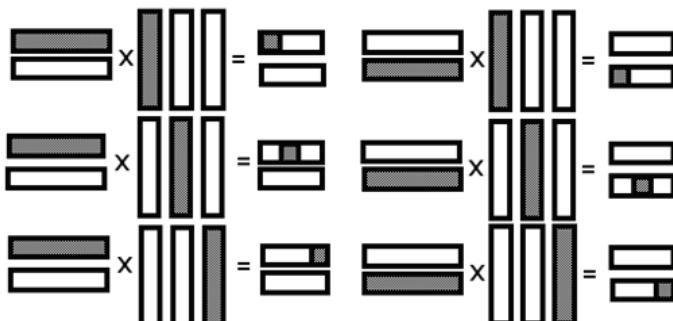


FIGURE A1.7 How rows and columns combine in matrix multiplication.

Interpolation

Interpolation is the act of determining what numerical values belongs at a particular point between some known values, assuming that there is a known behavior of the surface involved. For example, if at time $T = 0$ we measure a voltage of 2.0 volts and at time 1.0 we measure 3.4 volts, what was the voltage at time $T = 0.5$? Well, we don't actually know, but if we assume that the voltages follow a linear function (a line) then a good guess would be 2.6 volts. If you see that right away, you probably already have an insight into interpolation. If you do not, read on.

Figure A1.8 shows the interpolation example that we just did. There are two measurements that we know of, and we connect them with a straight line segment. Then to determine the voltage at $T = 0.5$, move along the line on the horizontal (T) axis to $T = 0.5$ and read off the height of the line at that point. In fact, we can read off the value at any time between 0 and 2, under the assumption that the line is an accurate representation of the real situation. The mathematical statement of this is: given two data points y_0 and y_1 that correspond to x values x_0 and x_1 , assume that $x_1 - x_0 = 1$

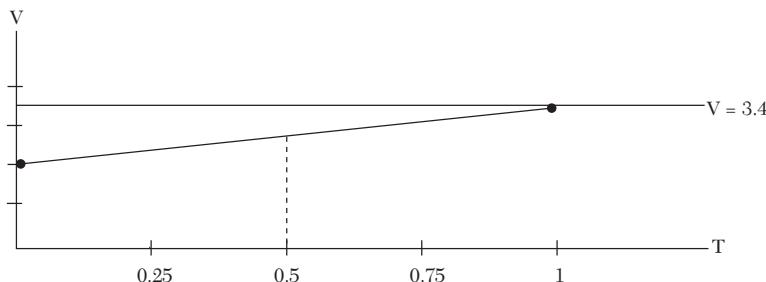


FIGURE A1.8 Linear Interpolation. Between the two points whose values are known, draw a line. Read off the value V from the line for any value of T you like.

(if it is not, we can scale it so it is by dividing by $x_1 - x_0$). Now any point between x_0 and x_1 can be expressed as a value $0 < \mu < 1$. For any such μ , the function value y at that point is given by:

$$y_\mu = (1 - \mu)y_0 + \mu y_1$$

We get as many interpolated samples as we like between any two points x_0 and x_1 by making μ as small as we like: the samples are at $x_0, x_0 + \mu, x_0 + 2\mu, \dots, x_1$. The *Processing* function that does this interpolation is:

```
float linearInterpolate (float y1, float y2, float u)
{
```

```

    return ( y1*(1-u) + y2*u );
}

```

If a line is not correct, we can use any other function we choose as an *interpolation function*. Linear interpolation is usually a fair first approximation, but it is quite jagged when there are many lines connecting multiple points (*piecewise linear* they call it). Using a polynomial can result in smoother curves, though.

There are many other interpolation functions you could choose, so there is no way to show you all of them. Possibly the most useful, other than the linear one, uses a cubic polynomial is used to approximate the data. A big advantage with cubic interpolation is that there is a high degree of continuity possible between adjoining segments. However, to get this benefit we need more data points with which to build the interpolation: four points, in fact. We never use cubic interpolation if we have only a very few data points.

What are the four points? Well, assume that y_0 and y_1 are as before, and we scale the distance between them again to be 1.0, so μ is as before too. Now also take the point that follows y_1 and call it y_2 ; also use the point before y_0 , and call it y_{-1} . The interpolation between y_0 and y_1 is computed as follows:

```

float cubicInterpolate(float y0, float y1,
                      float y2, float y3, float u)
{
    float a0,a1,a2,a3,u2;

    u2 = u*u;
    a0 = y3 - y2 - y0 + y1;
    a1 = y0 - y1 - a0;
    a2 = y2 - y0;
    a3 = y1;

    return(a0*u*u2+a1*u2+a2*u+a3);
}

```

How do we get the first and last segments interpolated? Standard practice is to guess! Add an extra point before the first one so that the slope agrees with that of the first segment, and do the same at the end. Either that or use a linear interpolation at the beginning and end. Figure A1.9 shows a sample set of data and interpolations, both linear and cubic. There are eight data

points to be modeled. Note that the cubic curve moves smoothly through and around the data points, whereas the linear interpolation changes shape abruptly.

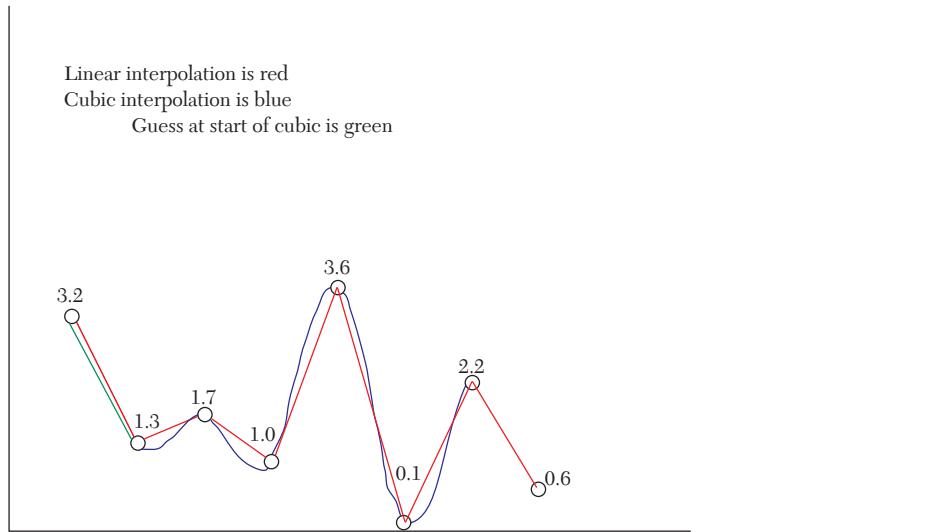


FIGURE A1.9 Eight data points (equally spaced X values) interpolated using linear and cubic interpolation.

Basic Physics

Some would define physics as the science that describes how the Universe operates. From the smallest objects (perhaps quarks) to the largest (galaxies), physics can be used to describe how objects interact, how things move, how the Universe changes as a function of time. However, it is important to remember that physics is based on observation. There is no way to derive the laws of physics from first principles, we must watch, describe, and verify or falsify. I say this to underscore the difference between mathematics and physics. Einstein once said that “As far as the laws of mathematics refer to reality, they are not certain; and as far as they are certain, they do not refer to reality.” We will look at some basic physics that we can use in a game, and try not to look too far past to the (really interesting) details underneath.

For a computer game, a key equation in physics is

$$\mathbf{F} = \mathbf{m}\mathbf{a}$$

Force equals mass multiplied by acceleration. Seems simple enough, but it describes almost everything we want to simulate in this book.

There is, naturally, more to it. What we need to do is to find an equation that defines the magnitude and direction of every force that acts on an object, in our case usually a vehicle. We then resolve the forces, treating them as vectors, so that we have one force remaining. This force acts on the vehicle, and since the vehicle has a mass it will accelerate according to the equation $\mathbf{F} = \mathbf{ma}$. This is a simplification, but will do for this example.

So, consider a car that is accelerating forward because its engine is moving it. There is a set of forces slowing it down: friction with the road, wind resistance, engine friction. There is one force pushing it forward. These forces are in simple opposition, making it simple to resolve them: subtract the frictional forces from the motive ones and compute a resultant. This is the remaining force, and will serve to accelerate the car forward.

Forces and accelerations are actually vectors. A force acts in a specific direction, and this would be indicated by a vector of three components, each representing a force in the three cardinal directions x, y, and z. Resolving the forces is simply a matter of adding the vectors, and the sum is the resultant net force.

Torque

Most forces, the simple ones at least, can be treated as if they act on the center of mass of the vehicle. This gives us a common point to act as the origin of the force vectors. On the other hand, some forces do not behave like this. If the car is struck by another car, the forces do not share the same origin. Consider that if a minivan is hit on the passenger front fender by a Fiat at 90 degrees to the direction of motion, the force applied to the front of the van causes it to rotate.

This is another way to apply a force that causes a different response. The force is applied at an offset to the center of mass. The result is rather like a lever or a crank, where a small force is amplified by the distance from the fulcrum or shaft - which here is the center of mass. This kind of force is called a *moment*, and it results in a new kind of force label (*torque*) and a rotational motion rather than a linear force. *Torque* is a measure of how much rotation will be caused by a force acting on an object. An example is shown in Figure A1.10

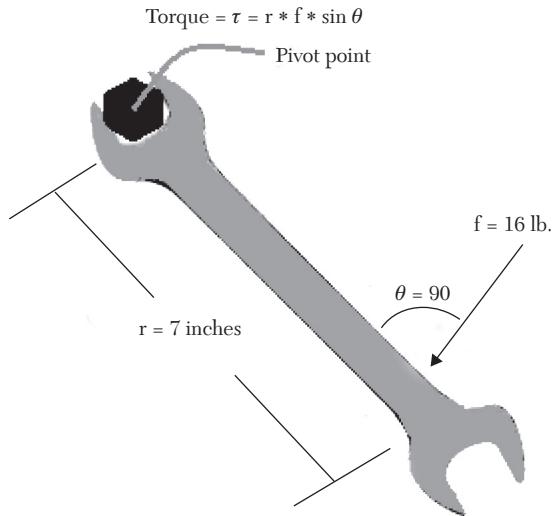


FIGURE A1.10 A pictorial example of torque. A force acting off the axis of the object at a distance r applies a stronger rotational force at greater values of r . In this case $\tau = 9.3$ foot-pounds.

The longer is *the moment arm*, the distance from the pivot point to the force labeled **r** in the figure, the greater is the torque. This fact is well-known to anybody who has removed a stubborn bolt or changed a tire. The object rotates about an axis, the **pivot point**. Sometimes this point is fastened to something, other times it will be the object's center of mass. The force acting on the object is **F**. The torque is normally a vector, but the formula given in the figure shows its magnitude.

What does this do for us? In the case of collisions of vehicles, it permits fairly precise calculations involving rotations; unless struck head-on, cars tend to spin after being hit by another vehicle. The same is true for any off-axis collision. Precise calculations can be done (for example, See: Keifer, 2005) but in the majority of cases we simply need to make it *look* right. So, the rotation involved in a collision will be proportional to the force and the distance of the collision from the center of rotation.

The sketch **rotation.pde** on the disk does precisely this. It draws a ball and a suspended rod. The 'w' and 's' keys move the ball up and down, and when space is pressed the ball speeds off to the right. It will strike the rod at some point along its length, resulting in a rotation about the fixed pivot

point. The rotation is greater when the ball hits the rod far from the pivot, as indicated by the code:

```
dt = -(bally-b1y) / frameRate;
```

where `dt` is the change in angle in each frame and `bally-b1y` is the value of the moment arm \mathbf{r} . What is correct here is that the rotational speed is a function of the torque, but the precise value of that speed is not correct; it can be adjusted simply by multiplying `dt` by a factor until it looks right.

References

- Fletcher Dunn and Ian Parberry (2011). *3D Math Primer for Graphics and Game Development, 2nd Edition*. A K Peters/CRC Press.
- Richard Feynman, Robert B. Leighton, and Matthew Sands (1977). *The Feynman Lectures on Physics, Volumes 1-3*. Addison-Wesley, Reading MA.
- Keifer, O., Reckamp, B., Heilmann, T., and Layson, P. (2005) A Parametric Study of Frictional Resistance to Vehicular Rotation Resulting from a Motor Vehicle Impact, SAE Technical Paper 2005-01-1203, 2005. <http://www.aegiforensics.com/library/pdfs/SAE%202005-01-1203.pdf>
- Joseph H. Kindle (1950). *Theory and Problems of Plane and Solid Analytic Geometry*, Schaum's Outline Series, McGraw-Hill, NY.
- Eric Lengyel (2011). *Mathematics for 3D Game Programming and Computer Graphics*. Delmar Learning; 3 edition
- James M. Van Verth and Lars M. Bishop (2008) *Essential Mathematics for Games and Interactive Applications: A Programmer's Guide*. Morgan Kaufmann; 2 edition
- Online physics tutorials: <http://www.rodedev.com/tutorials/gamephysics/>

APPENDIX

B

A PROCESSING PRIMER

This book is based on an assumption that the reader has some knowledge of the *Processing* language, a computer language designed for artists to create generative art. If you don't know anything about *Processing* that's OK; it is quite easy to learn, especially if you have experience with *Java* or *C++*. This Appendix will tell you much of what you need to get started, and is intended for people who have some experience with *Java*, *C*, or *C++*.

First you need to get a copy of the program. *Processing* is downloaded from the site <http://processing.org/download/?processing>, and is a single archive file that unpacks into a directory. That directory contains the *Processing* compiler and many other things, and does not need to be installed like most other software – the compiler can be executed simply by clicking on *processing.exe*. A *Processing* program is called a *sketch* and is a file ending in the suffix '*pde*'. A sketch file named, for example, "test.pde" must reside in a directory named "test" or the compiler will not accept it.

Basic Code Organization

A *Processing* sketch is organized into two parts, both residing in the same file for the most part. The first part is a function named `setup` which deals with initializations that occur just one time. This can include reading data and image files, allocating arrays and providing values for arrays, creating a window for drawing in, and like operations. `Setup` is called by the *Processing* system once at the beginning of sketch execution.

The second part is a function named `draw` that gets called every time a new picture is to be drawn; by default this is 30 time per second, but can

be changed. The purpose of a *Processing* sketch is to create a visual, an image consisting of pixels but that expresses something imagined by the artist. That image or *frame* can change every fraction of a second resulting in a moving image, or an animation. The rate at which images can be drawn can be adjusted by a function named `frameRate(n)` that is provided by the language, so film animation rates of 24 frames a second can be specified, as can more human speeds like 1 frame per second if high rates are not required.

A simple sketch would be:

```
void setup ()  
{  
    size (300, 200);  
}  
  
void draw ()  
{  
    background(255, 0, 0);  
}
```

This program calls two functions that are built into *Processing*. The first, named `size` and called in `setup`, specifies the size of the window drawn on the screen into which the artist's image will be drawn. In this case it is 300 pixels wide and 200 pixels high. The second, named `background` and called in `draw`, specifies the color that will appear in the background of the window. In this case the color is (255, 0, 0), which are the RGB coordinates for red.

So, this sketch will create a 300x200 window on the screen and fill it with red pixels. Figure B2.1 shows the window containing the code and the resulting graphics window.

Comments can be added to the code that will be ignored by *Processing*. The symbol “//” means that all text that follows to the end of the line is a comment. Otherwise the symbol “/*” begins a comments, and all text up to the symbol “*/” will be ignored. Adding English descriptions to the code helps other programmers read and understand it.

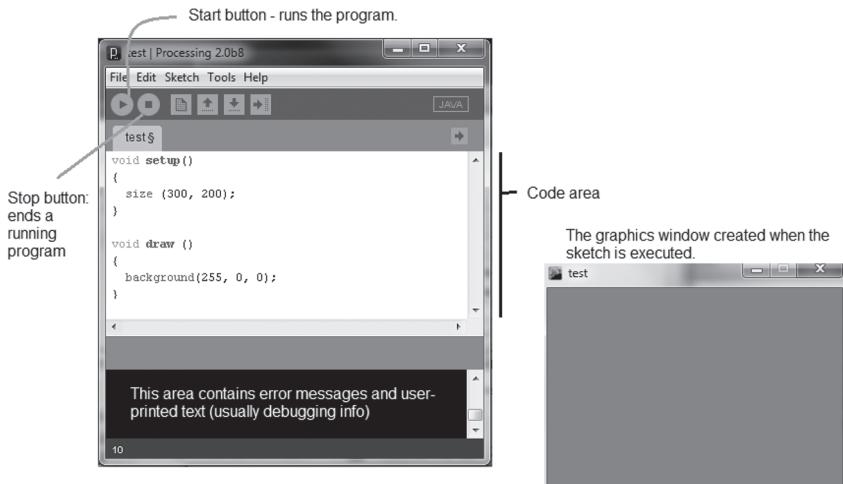


FIGURE B2.1 The programming and execution environment of a simple *Processing* program.

Declarations, Variables, and Types

Types are those of *Java*, and the most used are `int` (integer), `float`, and `boolean`. Declarations are simply a type name followed by a list of variables:

```
int i, j, k;
float x=0.0;
```

The `float` declaration above has an initialization, which sets the variable to a value at the outset; this is always a good idea. Assignments can be made to variables in obvious ways using the assignment operator “`=`”.

Processing has arrays. An array must be declared, allocated, and initialized. So, an example could be:

```
int list[] = new int[12];
```

The “`int list[]`” part is the declaration; the “`= new int[12]`” part allocates 12 array elements to the array, thus setting its size. A declaration such as:

```
int list[] = {1, 2, 3, 4, 5};
```

allocates 5 array elements and initializes them to the values 1, 2, 3, 4 and 5 respectively.

The `char` type represents a character; constants are in single quotes. So, for instance:

```
char ch;  
.  
. .  
ch = 'x';
```

The `String` type is much more complex, and is really a class instance (see: Class section below). Constants are characters between double quotes, and usually represent something to be printed or characters typed by a user. For instance:

```
String s;  
.  
. .  
s = "Hello";
```

Most important String operations are performed by functions belonging to the class.

A set of declarations can be specified at the beginning of the program, before the `setup` function, and these variables are visible anywhere in the program. In fact declarations can be placed anywhere, but that's a poor idea as they can be hard to find. Declarations can also be placed within functions; again, put them at the beginning of the function.

Statements

We can assign the result of an expression or function call to a variable using the assignment operator “`=`” also. Expressions are the usual thing, with standard precedence rules. Processing is careful of converting types, so be alert to message to that effect. Sample assignments from expressions are:

```
int i,j,k;  
float x, y;  
String s;  
.  
. .  
i = 12; j = 10;  
k = i-j;  
x = 1.2;  
y = PI/2.0; // PI is a constant 3.14159 ... provided for us  
j = y; // Causes an error: can't convert from float to int.  
j = round(y); // This is fine: rounding a float gives an int.
```

```
x = sin (y); // Basic math function are provided.
```

Flow of Control

All statements agree with Java.

Conditionals:

```
if (logical expression)           if (logical expression)
{                                statement;
    // Group of statements
} else {                         else
    // Group of statements
}
```

Example:

```
if (a < b) a = a + 1;
else b = b + 1;

switch (i)
{
case 1: statements;
break;
case 2: statements;
break;
. . .
default: statements;
}
```

Loops:

```
for (i=0; i<imax; i= i+increment)
{
    statements;
}
```

This loop begins with the variable *i* being given the value 0 (in this case). If *i*<*imax* the statements are executed, then the value increment is added to *i*. If the new value of *i* is less than *imax* the statements are executed again, and so on until *i*>=*imax*.

```
while (a < b)
{
    statements;
}
```

The statements in this loop execute repeatedly until the condition `a < b` becomes `false`. The condition is tested only at the top of the loop, and if `false` at the outset the statements are never executed.

```
do
{
    statements;
} while (a < b);
```

This loop executes the statements first, then checks the condition; so long as the condition is true (in this case `a < b`) the statements execute repeatedly.

Functions, Procedures, Methods

In a computer language a function calculates a value based on some values that are passed as parameters and returns that value as a result. It has the form of a block of statements with a heading that defines the function type and parameters. A simple example is a function that returns the square of its parameter:

```
// Function is named square and returns a float (real)
float square (float x) // parameter is named x and is real
{
    return x*x; // Return a value that is x * x;
}
```

We can invoke this function by using its name and giving it a value:

```
x = square(x); or y = square (2.0*PI);
```

The parameter passed can be any expression so long as it is floating point. Functions can have multiple parameters. Here is a function that returns the minimum of two values passed to it:

```
// Function is named minimum and returns a real number
float minimum (float x, float y) // Take two real parameters
{
    if (x < y) return x; // Return x or y, whichever is smaller.
    return y;
}
```

Again, we call this using its name and passing two values:

```
x = minimum (2.0, x);
```

A procedure is a function that returns no value; the name for ‘no value’ is `void`. Some things a procedure can do is draw something in the window, do input or output, update data tables. Here is a procedure that sorts an array of integers:

```
void isort (int []arr, int N)
{
    int i,j,k,t;
    for (i=0; i<N; i++)
    {
        k = i;
        for (j=i; j<N; j++)
            if (arr[j] < arr[k]) k = j;
        t = arr[k];
        arr[k] = arr[i];
        arr[i] = t;
    }
}
```

The return type is `void`. The procedure declares some variables at the beginning (called *local variables*) for use in the procedure only. When execution is complete the array `arr` has values within it appearing in ascending order. Some procedures manipulate variables declared outside of the procedure, like those declared at the beginning of the sketch. This is allowed and is not always a bad thing.

A *method* is the name for a function or procedure that is coded within a class. There is no real need for a new name, but some people have been taught to use *method* instead of *function* and you should be aware of it.

Classes

A `class` is a way to encapsulate all (or most) aspects of some object being implemented. The name of a `class` can be used as a type name, and instances of classes can be created. Again, *Processing* classes are the same as *Java* classes in syntax and semantics.

A simple class might contain only data, but that would be unusual. Let’s make a class that represents a point in three dimensional space. It will have

an X, Y, and Z coordinate, and it will have a function to compute the distance between it and any other point. Here's the class:

```
class point
{
    float x, y, z;

    point (float xx, float yy, float zz)
    {
        x = xx; y = yy; z= zz;
    }

    float distance (float xx, float yy, float zz)
    {
        float d;
        d = (x-xx)*(x-xx) + (y-yy)*(y-yy) + (z-zz)*(z-zz);
        return sqrt(d);
    }

    float distance (point p)
    {
        return distance (p.x, p.y, p.z);
    }
}
```

This class can be used as a type; declare a point as follows:

```
point p1, p2;
```

Creating an instance is done with `new`; when that happens the class constructor, which is the function having the same name as the class, is called. This class is passed three real values when created: the coordinates.

```
p1 = new point (10.0, 20.0, 30.0);
p2 = new point (10.0, 10.0, 10.0);
```

Using the distance function can be done in two ways; one version of distance is passed an x, y, and z coordinate against which to calculate the distance.

```
p1.distance (10., 20., 30.);
```

which returns the value 22.36068. The other way is to pass another point as a parameter, and the distance between the two points is computed:

```
p1.distance (p2);
```

Both versions of `distance` are declared inside of the `Point` class, and the one called is determined by *Processing* by looking at the parameters.

Output

Processing offers a procedure that writes to the console. This is valuable for debugging – values of variable can be printed to see if they are correct. The procedure is called `print`, and it prints a `String`:

```
print("The answer is 100");
```

`println` is the same but prints an end of line after the string:

```
println ("The answer is 100");
```

`String` operations can be used to print more complex things. Concatenating two `Strings` is done using the “`+`” operator, and `ints` and `floats` convert into `Strings` in `String` expressions; thus, if `i=100` is an `int` then:

```
println ("The answer is "+i);
```

results in “The answer is 100” being printed.

Graphical Procedures

There are many procedures that draw to the graphics screen. Most important are:

<code>point (x, y)</code>	Draw a single point
<code>line (x1,y1, x2,y2)</code>	Draw a line
<code>ellipse (x, y, width, height)</code>	Draw an ellipse (circle)
<code>triangle (x1,y1, x2, y2, x3,y3)</code>	Draw a triangle
<code>rect (x1, y1, width, height)</code>	Draw a rectangle

Colors are specified in RGB or RGBA coordinates. The stroke color is used to draw lines:

```
stroke (255, 0, 0); // draw red lines
```

The fill color is used to fill shapes and to draw characters.

```
fill (0, 0, 255); // draw characters in blue.
```

Text (strings) can be plotted on the graphics screen using the procedure

```
text ("Plot this string", x, y);
```

which plots the text string parameter at the coordinate (x, y) on the screen.

And more ...

Processing is a complex language possessing very useful libraries and a skilled and fascinating user base, many of whom contribute to the open-source system. A good summary of the language is given in the book *Learning Processing* by Daniel Shiffman (<http://www.learningprocessing.com/>). Online documentation is at <http://www.processing.org/reference/>.

And, as always, the best way to learn a language, or how to program in the first place, is to write a lot of code.

INDEX

- 3D game example, 217–235
2D rendering versus, 233–235
assets, 221–233
 bridge, 225–226
 cargo containers, 222–224
 cargo vessel, 224–225
 controlling the submarine, 230–232
 heads-up map display, 226–227
 illumination, 232
 radiation sensor, 228
 sound, 232–233
 the indicators, 229–230
3D graphics, 123–125
 storing and drawing polygons, 125–127
 using polygons to build objects, 124–125
- A**
- Active list, 176
Android, 291
Android handheld devices, 289–311
 an Android puzzle: touch events, 300–302
 key things that change when Android is targeted, 301–302
 sound in Android, 308–311
 installing APWidgets, 308
 using the APMediaPlayer, 308–311
 playing a game on a real device, 302–307
 exporting an Android game, 304–307
 targeting Android from processing, 292–300
 getting set up, 293–295
 Getting Through the Installation Issues, 295–299
 Your First Android Program, 299–300
 how cell phones work, 290–292
- Animation, 263–286
 ambient, 279–283
 animation math, 269–274
 3 moves in the environment, 270
 motion equations, 271–274
 character animation, 283–284
 creating elementary, 264–269
 cut scenes or full motion video (FMV), 284–286
 reactive, 274–279
 using real images, 277–279
- A Prism, 139–145
 geometry: rotation, 144–145
 geometry: scaling, 143–144
 geometry: translation, 141–143
- Artificial intelligence, 10, 17–18, 159
- Axis Aligned Bounding Box (AABB), 175
- B**
- Basic Autonomous Control, 190–196
 avoidance behavior, 193–195
 cruising behavior, 192–193
 how to control a car, 191–192
 waypoint representation and implementation, 195–196
- Back-face culling, 181
- Bots, 159
- Browser, 240
- C**
- Canvas, 245
Center of projection (COP), 132
Client side processing, 240
Cognitive simulation, 159
Collision detection, 161–184
 broad phase, 166–183
 geometric tests, 167–169
 object-oriented bounding box (OOBB), 177
 operational methods, 166–167
 ray/triangle intersection, 181–183
 space subdivision, 178–180

sphere versus plane collision, 170–175
 using bounding boxes, 175–177
 using enclosing spheres, 169–170
 narrow phase, 180–181
 one-dimensional collisions, 161–164
 packages, 183–184
 ColDet, 184
 I-Collide, 184
 Rapid, 184
 Swift, 184
 Vclip, 183–184
 two-Dimensional Collisions, 164–166
 Colors, shading, and textures, 145–153
 texturing, 146–150
 theory, 150–153
 texture mapping to a triangle, 152–153
 Computer game, 10
 Cruising state, 197

D

Delay, 197
 Doppler effects, 8

E

Embedded system, 291

F

Finite State Machines, 196–205
 FSA in practice, 198–200
 start, air, damaged, attacking, defending,
 searching, patrolling, skidding, stopping,
 201–205
 state and the ‘what do we do now’
 problem, 200–201
 Fonts, 256
 Friction, 270

G

Game architecture, 9–22
 game state, 18–22
 broadcast-listener, 20–21
 global state, 18–19
 managers, 19–20
 push/pull (client server), 19

shared and global entities, 21–22
 playing the game by the rules, 16–18
 most of a computer game is hidden,
 16–17
 presenting the virtual universe, 10–15
 comments on optimization, 15
 geometric level, 13–14
 object level, 13
 rasterization, 15
 the graphics system, 11–13
 the audio system, 15–16
 Game board, 16
 Game design document (GDD), 87
 Game mechanics, 4
 Genre, 2
 Graphics Interchange Format (GIF), 43
 Graphics pipeline, 13
 Graphics and sound, 5

H

Hockey Pong, 87–120, 249–256
 Hockey Pong game design document,
 106–120
 How the Web Works, 240–245
 HTML and HTML5, 241–243
 HTML5, 245–246
 JavaScript, 243–245

I

Implementing the game using prototypes,
 88–105
 prototype 0, 88–89
 prototype 1, 89–94
 buttons, 91
 end screen, 94
 options screen, 92–93
 play screen, 93
 screens, 90–91
 start screens, 91–92
 prototype 2, 94–100
 ambience, 95–96
 artificial intelligence, 97
 collisions, 98
 impacts, 96–97
 other minor features, 100

- random bounces, 97–98
- right paddle, 98, 100
- sound, 95
- user control, 94
- prototype 3, 100–105
 - a timer class, 101–102
 - game state 0–5, 102–104
 - the scoreboard, 104–105
- Integers in JavaScript, 251–252
- Interesting games, aspects of, 2–4
 - conflict, 3–4
 - venue, 2–3
- Interface, 6–9
 - fidelity/accuracy, 9
 - pace/scale, 8–9
- J**
- Jarsigner, 304
- Joint Photographic Experts Group (JPEG), 43
- K**
- Keytool, 304
- M**
- Making a Game Function on a Computer, 22–32
 - an example - Hockey Pong, 22–24
 - control of objects, 28–31
 - front-end/back-end, 31–32
 - high concept design, 24
 - game design document, 24
 - the control of time, 26–28
 - the game loop, 24–26
- Mip-mapping, 12
- More JavaScript, 257–258
- Multiple fixed steps, 27
- N**
- Next event method, 26
- Node, 207
- O**
- Object models, 153–156
 - Pshape, 155
- Object-oriented bounding box (OOBB), 177
- OpenGL graphics, 127–139
 - projecting the image of a scene onto a plane, 131–136
 - hidden object/surface removal, 135
 - triangles, 127–130
 - viewing, 130–131
 - viewpoint, 136
- P**
- Packets, 240
- Pathfinding, 205–211
 - A*Search, 207– 211
 - squence of steps, 206–207
- Persistence of vision, 263
- Perspective projection in, 133–134
- PIImage, 39–47
 - getting and setting color values, 45–47
 - image files, 43–45
 - pixels and colors, 45–47
 - the PGraphics type or graphics context, 55–57
 - rendering to a PIImage, 57
- Playing Sounds, 67
- Playtesting, 105
- Polygonization, 125
- Portable Network Graphics (PNG), 44
- Preloading, 252
- Processing, 37–57
 - drawing shapes, 47–51
 - Hockey Pong, 49
 - lines and colors, 48–49
 - text and fonts, 50–51
 - internals of the type PIImage, 51–56
 - accessing pixels, 53–55
 - image size, 52–53
 - review, 37–39
- Processing.js, 246–249
 - development tips, 258–259
 - making a program work, 247–249
 - sound in, 253–256
- Perspective transformation, 12
- Props, 5–6

Q

Quantization, 12

R

Rapid Application Development (RAD), 221

S

SDK (Software Development Kit), 293

Sample Game Design Document Hockey Pong: High Concept, 32–34

Sandboxes, 3

Searching for trouble, 204

Similar triangle, 133

SMV (Sub Marine Vessel) rainbow, 218–221
screens, 219–220

game environment, 220–221

Sound, 61–83

a processing game audio system, 75–80
example: 2D positional sound, 78–80
positional audio, 76–78
basic audio concepts, 62–65
analog to digital (A to D) converter or ADC, 65

volume or loudness, frequency, and period 63

basic things in computer games, 61

introduction to Minim, 66–75

declaring and initializing minim, 66–67
display of synthesized sounds, 70–72
file, 69–70

Minim and Java sound support, 74–75

monitoring output, 72–73

recording sounds to a file, 73–74

sample, 68–69

snippet, 67–68

using Minim, 67–68

The simpleAudio Class, 80–83

using simpleAudio, 81–83

Start state, 197

Stochastic navigation, 211–213

Superstructure, 224

T

Tagged Image File Format (TIFF), 43–44

Tessellation, 125

Testing, game, 105–106

Touch screen, 292

Tweens, 266

U

USB interface, 6

V

Vertices, 125

Video game, 1

Viewing plane or projection plane (PP), 132

Viewing transformation, 130

Z

Z-buffer algorithm, 135

Z-fighting or Z-flicker, 136

Zipalign, 305